Alexandre Coelho Brasil

Crystalline: Uma arquitetura transparente para cache em aplicações multicamadas

Rio de Janeiro
2007

Alexandre Coelho Brasil

Crystalline: Uma arquitetura transparente para cache em aplicações multicamadas

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Informática, Instituto de Matemática e Núcleo de Computação Eletrônica da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Informática.

Maria Luiza Machado Campos
Ph.D.

Paulo de Figueiredo Pires
D.Sc.

Rio de Janeiro
2007

Alexandre Coelho Brasil

Crystalline: Uma arquitetura transparente para cache em aplicações multicamadas

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Informática, Instituto de Matemática e Núcleo de Computação Eletrônica da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Informática.

Aprovada em 30 de junho de 2006 por:

_____
Prof. Maria Luiza Machado Campos, Ph.D., UFRJ (co-orientador)

_____
Prof. Paulo de Figueiredo Pires, D.Sc., UFRN (co-orientador)

_____
Prof. Marta Lima de Queirós Mattoso, D.Sc., UFRJ

_____
Prof. Vanessa Braganholo Murta, D.Sc., UFRJ

_____
Prof. Flávia Coimbra Delicato, D.Sc., UFRN

Rio de Janeiro
2007

À minha família.

Sem o apoio de vocês, nada seria possível.

“Things should be made as simple as possible, but no simpler”
Albert Einstein

Resumo

Atualmente, a utilização de um mecanismo de cache é condicionada à dependência de uma plataforma específica ou à alteração do código-fonte da aplicação em questão. Mecanismos para a aplicação de um sistema de cache de forma transparente, que mantenham a aplicação independente de plataforma de desenvolvimento são raros ou inexistentes. Este trabalho apresenta a definição e a implementação de um *framework* que fornece um serviço de cache de forma transparente, que tanto pode ser aplicado à camada de persistência como à camada de negócio de sistemas de informação. A arquitetura deste *framework* prevê três pontos focais para o serviço de cache: dados dos componentes de persistência, resultados de métodos de negócio e resultados de métodos de busca.

Os resultados dos testes aplicados sobre este *framework* demonstram que o mesmo é capaz de fornecer de forma satisfatória os serviços de cache de dados de componentes de persistência e de cache de resultados dos métodos de negócio. A implementação do serviço de cache de resultados de métodos de busca, por sua vez, mostrou-se extremamente ineficiente.

Abstract

Nowadays, the usage of a cache mechanism is conditioned to a dependency between the application and a specific platform, or to changing the source code of the application in question. Mechanisms that apply a cache system in a transparent way, and that maintain the application's independency from development platforms are rare or inexistent. This work presents the definition and implementation of a framework that supplies a cache service in a transparent way, which can be applied to both the persistence and business layers of information systems. This framework's architecture defines three focal points for applying the cache service: persistence component's data, results from business methods and results from finder methods.

The results from the tests applied to this framework demonstrate that it's capable of providing a cache service for both persistence component's data and business methods' results with satisfactory levels. The implementation of a cache service for finder methods, on the other hand, showed extremely inefficient results.

Lista de siglas

AOP	Aspect Oriented Programming
API	Application Programming Interface
BMP	Bean Managed Persistence
CMP	Container Managed Persistence
COM	Component Object Model
CORBA	Component Object Request Broker Architecture
DCOM	Distributed Component Object Model
EJB	Enterprise Java Beans
J2EE	Java 2 Enterprise Edition
TPC	Transaction Processing Performance Council
XML	eXtensible Markup Language

Sumário

1 Introdução

Os sistemas de informação tiveram sua curva evolucionária afetada diretamente pelo advento das redes de computadores. Antes da adoção das redes, os sistemas de informação eram desenvolvidos como ilhas auto-suficientes, contendo toda a lógica de negócio e os dados em um único local. Este tipo de sistema tinha como principal desvantagem a incapacidade de partilhar as informações com outros computadores, isolando os dados de forma que o único meio de acessá-los simultaneamente era através de múltiplos terminais acoplados a um mainframe.

A disseminação das redes de computadores nos ambientes corporativos tornou possível o compartilhamento de uma mesma fonte de dados entre computadores distintos simultaneamente. Com a possibilidade de se compartilhar dados surgiu o modelo de desenvolvimento cliente-servidor, no qual os clientes devem acessar os dados através de um servidor central. A evolução deste modelo levou ao desenvolvimento da tecnologia de componentes, que são pequenos artefatos de *software* projetados de forma a aderir a especificações pré-existentes, e que podem ser acessados remotamente (MCINNIS, 2000).

A popularização desta tecnologia levou ao modelo de desenvolvimento baseado em componentes, que foi amplamente adotado por muitas empresas de desenvolvimento de *software*. Este modelo de desenvolvimento prega que os softwares devem ser decompostos em componentes, que são conectados para formar uma aplicação completa. Este modelo é atualmente o padrão adotado para desenvolvimento de sistemas corporativos no mundo todo.

Atualmente existem quatro modelos dominantes no mercado: CCM[1], EJB[2], Microsoft® COM+[3] e Microsoft® .NET. Estes modelos definem tipos similares de componentes: de sessão e de entidades[4]. Os componentes de sessão foram criados para encapsular as regras de negócio de uma aplicação, podendo ser utilizados para a criação da camada de controle de uma aplicação. Os componentes de entidade são aqueles que irão encapsular a lógica de

---

[1] CORBA (Common Object Request Broker Architecture) Component Model

[2] Enterprise Java Beans

[3] Component Object Model.

[4] Os modelos da Microsoft® não oferecem suporte aos componentes de persistência.

acesso aos dados, bem como a lógica de negócio concernente a eles como, por exemplo, a sua validação.

Todos os modelos acima estão associados a serviços que podem ser utilizados pelos componentes, desonerando a tarefa dos desenvolvedores. Ao conjunto formado pelos serviços adicionais e pelo modelo de componentes dá-se o nome de plataforma. Todos os modelos de componentes descritos estão associados a plataformas específicas, que oferecem basicamente os mesmos serviços, tais como gerência de ciclo de vida dos componentes, gerência de transações, controle de acesso aos componentes, etc. Para os modelos apresentados acima, temos as plataformas J2EE[5] (que abrange a especificação EJB), CORBA[6] (para o modelo CMM), e COM/DCOM/COM+ e .NET (para os modelos homônimos). À implementação das especificações que definem uma plataforma (modelo de componentes mais serviços associados) dá-se o nome de servidor de aplicação.

O modelo de componentes EJB permite que os componentes de entidade[7] gerenciem sua persistência diretamente ou que esta seja gerenciada pelo servidor de aplicação. Na persistência gerenciada pelo próprio componente (*Bean Managed Persistence – BMP*), o desenvolvedor deve codificar diretamente no componente o código de acesso aos dados. Ao delegar a gerência de persistência dos componentes para o servidor de aplicação (*Container Managed Persistence – CMP*), é necessário fornecer ao mesmo as meta-informações sobre como relacionar os dados do repositório a estes componentes.

O baixo desempenho dos componentes de persistência decorre da ineficiência de acesso ao repositório de dados. O modelo EJB permite a modelagem de componentes de granulosidade fina, o que torna o número de acessos ao repositório de dados necessários para carregar grandes quantidades de dados excessivamente alto. Embora este problema ocorra

---

[5] Java 2 Enterprise Edition

[6] Common Object Request Broker Architecture

[7] Na plataforma J2EE, os componentes são chamados de *beans*, e dividem-se entre *Session Beans* (componentes de sessão), *Entity Beans* (componentes de entidade) e *Message Driven Beans* (componentes de sessão ativados por mensagens assíncronas).

tanto no esquema BMP como no CMP, os servidores de aplicação J2EE – assim como a própria plataforma[8] – somente provêem mecanismos de otimização para o mecanismo CMP.

O problema de desempenho observado na camada de persistência no modelo EJB levou os desenvolvedores a reconsiderarem a sua adoção. A utilização deste tipo de componente passou a ser repensada e, muitas vezes, desencorajada, levando ao surgimento de novas tecnologias para a camada de persistência. Junto com o surgimento destas novas tecnologias, pôde-se observar o surgimento de uma nova tendência: a minimização do uso dos componentes.

Após anos de massiva utilização de tecnologias de componentes no mundo corporativo os desenvolvedores notaram que, em muitos casos, a utilização de tecnologias de componentes como EJB e .NET acrescentava complexidade ao desenvolvimento sem gerar muitos benefícios para as aplicações. A partir desta constatação ocorreu uma retomada da utilização dos chamados objetos planos – objetos que não seguem nenhuma especificação de componentes – tornando as aplicações mais simples.

Deste movimento, apenas na camada de persistência podemos destacar o surgimento das tecnologias Java Data Objects (JDO), da Sun Microsystems (2003b), e do Hibernate (BAUER, KING, 2004), *framework* para mapeamento de esquemas de dados relacionais em um modelo de objetos formados por objetos planos. O surgimento do JDO foi importante para sinalizar que, mesmo dentro da própria Sun, havia desenvolvedores insatisfeitos com a especificação EJB liderada pela própria empresa. A larga adoção do Hibernate, por sua vez, sinaliza o fortalecimento do movimento de retomada da utilização de objetos planos no desenvolvimento de aplicações corporativas.

Em muitos casos a substituição do modelo EJB por objetos planos nas aplicações corporativas resulta em uma melhoria no desempenho geral destas aplicações. Infelizmente esta melhoria de desempenho obtida pela simplificação da arquitetura das aplicações ainda é limitada por fatores simples como a escassez de recursos computacionais tais como capacidade de processamento dos servidores e largura de banda, ou mesmo pelo repositório de dados.

---

[8] Atualmente em sua versão 2.1, a especificação EJB ainda não cobre serviços como cache e notificação, essenciais para sistemas de alto desempenho (The Middleware Company, 2002).

Uma técnica largamente utilizada para otimizar a gerência dos recursos computacionais disponíveis e melhorar o desempenho das aplicações é a utilização de sistemas de cache. Atualmente, existem propostas para camadas de cache de componentes de apresentação (ANTON *et al*., 2003), para componentes de sessão (POHL, GÖBEL, 2003; POHL, SCHILL, 2003 e PFEIFER, JAKSHITSCH, 2003), e para o simples cache de dados (BORTVEDT, 2001; JBOSSCACHE, 2003; TANGOSOL, 2001). Infelizmente as propostas existentes ou são dependentes de plataforma ou são extremamente invasivas.

Estratégias de cache dependentes de plataforma têm como problema óbvio a impossibilidade de serem adotadas por aplicações que não tenham sido desenvolvidas para a plataforma em questão. Desta forma, sistemas de cache para componentes EJB não podem ser utilizados para cache de objetos planos, por exemplo, o que torna a utilização deste tipo de estratégia extremamente limitada.

As estratégias de cache invasivas, por outro lado, têm como grande problema a necessidade de alteração do código-fonte da aplicação para implementação do serviço de cache. Um sistema de cache de dados genérico pode ser utilizado junto a qualquer aplicação, desde que os desenvolvedores a projetem para funcionar com o mesmo. Em outras palavras, os desenvolvedores passam a ter como requisito extra a manutenção dos dados em cache, tornando o desenvolvimento mais custoso. Além disso, por estar intimamente ligado ao código-fonte das aplicações, a substituição do sistema de cache utilizado torna-se onerosa, pois é necessário reescrever todo o código-fonte que utilize o sistema de cache.

Acreditamos ser possível criar um sistema de cache que seja ao mesmo tempo independente de plataforma e totalmente não-invasivo, podendo ser utilizado em uma gama maior de aplicações sem afetar a forma como as mesmas são desenvolvidas. Tal sistema poderá ser utilizado para aplicar uma camada de cache a aplicações pré-existentes sem que seja necessário modificar o código-fonte destas aplicações, de forma totalmente transparente. Para tanto, podem ser utilizadas as tecnologias de programação orientada a aspectos (KICZALES *et al.*, 1997) e anotações (SUN, 2004).

O objetivo deste trabalho é o desenvolvimento de uma arquitetura de cache com estas características e de uma implementação de tal arquitetura que disponibilize um serviço de cache sem sacrificar sua portabilidade e transparência. Esta implementação de referência foi batizada Crystalline, em uma alusão à sua transparência.

O restante desta dissertação está organizado da seguinte forma: no capítulo 2 discorremos sobre os sistemas de cache existentes atualmente, sobre os trabalhos recentes na área de cache de componentes, suas vantagens e desvantagens; no capítulo 3 iremos apresentar as ferramentas que possibilitaram a injeção do serviço de cache de forma transparente; no capítulo 4 apresentamos a arquitetura de nossa proposta para um sistema de cache não-invasivo e os principais módulos do *framework* resultante; no capítulo 5 descrevemos como essa proposta foi desenvolvida; o capítulo 6 descreve os testes de desempenho efetuados para validar a proposta deste trabalho e os resultados obtidos; finalmente, no capítulo 7 explicitamos nossas conclusões.

2 Cache em sistemas baseados em componentes

Sistemas de cache são utilizados desde o principio da computação, sempre com o objetivo de aumentar o desempenho de sistemas computacionais, sejam na forma de *hardware* ou de *software*. Sistemas de cache são utilizados em microprocessadores, discos-rígidos, sistemas de gerenciamento de bancos de dados e em diversos outros sistemas aonde o desempenho é um fator crítico (FRANKLIN, CAREY, LIVNY, 1997).

O conceito de cache é simples: armazenar os dados resgatados em uma localização mais próxima – com acesso mais veloz – para reutilizá-los quando necessário. Microprocessadores são equipados com diversos sistemas de cache, que permitem que uma pequena quantidade de dados seja armazenada localmente após a sua obtenção da memória RAM, significativamente mais lenta do que a memória cache interna. Sistemas de gerenciamento de bancos de dados obtêm dados do disco rígido e os armazenam na memória RAM para tentar diminuir o tempo de resposta para seus usuários.

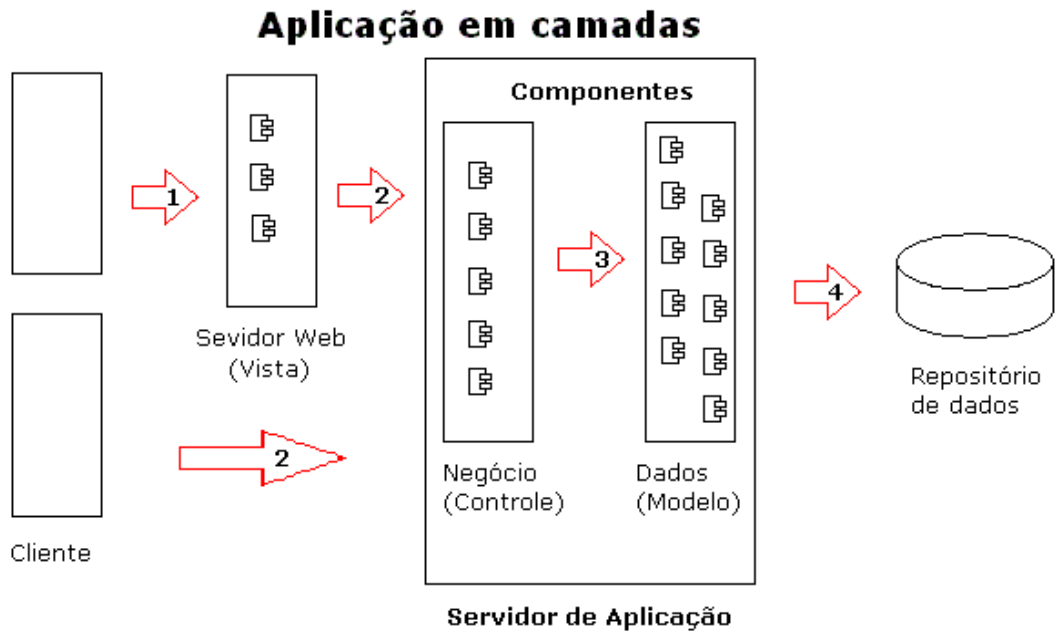

Figura 2.1 – Aplicações de cache em sistemas multi-camadas

Para aplicações multicamadas, existem atualmente diversas propostas para implementação de cache, indo desde o cache de páginas web ao cache dos dados obtidos no

repositório. A figura 2.1 representa a arquitetura de uma aplicação multicamada típica, podendo ser utilizada por clientes finos[9] (e.g. aplicação *web*) ou grossos[10] (e.g. aplicação *desktop* típica). Cada uma das setas na figura 2.1 representa um ponto de aplicação de cache.Entre um cliente *web* e o servidor *web* pode ser usado um sistema de cache de componentes de apresentação (figura 2.1, número 1), utilizados para páginas *web* estáticas (servidores *proxy*). Entre o servidor *web*/cliente *desktop* e o servidor de aplicação podem ser utilizados sistemas de cache de componentes de sessão (figura 2.1, número 2), que armazenam os resultados de invocações aos métodos dos mesmos. Os sistemas de cache utilizados pelos servidores de aplicação se encaixam entre os componentes de sessão e os componentes de entidade (figura 2.1, número 3), e entre os componentes de entidade e o repositório de dados (figura 2.1, número 4).

Embora sistemas de cache sejam utilizados em diversas áreas da computação, as aplicações de sistemas de cache mais significativas para este trabalho são as de cache de dados e cache de componentes de negócio, conforme veremos nas seções a seguir.

2.1 Cache de dados persistentes

O uso de um sistema de cache para dados persistentes é um dos usos mais comuns para esta técnica dentro da área de *software*. Com freqüência, aplicações que utilizam um repositório de dados persistentes armazenam temporariamente os dados recuperados, objetivando evitar novas consultas ao repositório para obtenção destes mesmos dados. Ao evitar novos acessos ao repositório de dados, é possível diminuir o tempo de resposta aos usuários e, ao mesmo tempo, aumentar o número máximo de usuários que podem ser atendidos simultaneamente.

A diminuição do tempo de resposta para os usuários decorre da eliminação do acesso ao repositório de dados ao obter os dados do cache em memória, resultando na eliminação do tempo que a aplicação espera por uma resposta deste repositório de dados. O aumento da capacidade de usuários simultâneos, por sua vez, decorre da desoneração do repositório de dados. O uso de cache de dados por parte dos clientes implica um número menor de usuários

[9] *Thin-clients* (clientes finos) são aqueles que não possuem inteligência (clientes “burros”), obrigando o servidor a realizar todo o processamento da aplicação.

[10] *Thick-clients* (clientes grossos) são aqueles que possuem alguma inteligência, desonerando a carga de processamento do servidor ao realizar o processamento (total ou parcial) no próprio cliente.

necessitando realizar consultas ao repositório de dados. Assim sendo, sem alterar a capacidade de atendimento do repositório de dados, um número maior de usuários pode ser atendido simultaneamente pela aplicação (ALMAER, 2003).

Em sistemas distribuídos a utilização de sistemas de cache locais é potencialmente perigosa devido à possibilidade de se trabalhar com dados obsoletos. Quando o mesmo dado é armazenado em caches locais em clientes distintos, existe a possibilidade de um usuário realizar uma alteração neste dado e tornar inválidas as cópias armazenadas nos demais clientes. Para solucionar este problema diversas soluções foram propostas, utilizando técnicas tais como a replicação e a invalidação de dados, que podem ocorrer de forma síncrona e assíncrona.

A utilização da replicação síncrona de dados garante a consistência dos dados entre todos os clientes. Além disso, essa técnica possibilita um aproveitamento maior dos dados em cache (*hit ratio*), pois os dados obtidos por um cliente passam a alimentar o cache de todos os outros. Em contrapartida, a necessidade de garantir que todos os nós receberam as atualizações corretamente aumenta as chances de as transações dos usuários falharem.

Apesar dos benefícios trazidos por esta técnica, sua utilização em aplicações com um número elevado de clientes pode ser problemática. Em aplicações com um nível elevado de alterações de dados a replicação destes traz um *overhead* de comunicação que pode ser alto, devido à necessidade de um nó se comunicar com todos os outros nós da aplicação e transmitir suas informações. Dependendo da quantidade de nós presentes em uma aplicação utilizando esta técnica, o uso de cache pode vir a tornar seu desempenho pior do que sem a utilização de um sistema de cache (GRAY *et. al.*, 1996).

É possível diminuir o *overhead* de comunicação através da utilização de comunicação *multicast*, em detrimento das mensagens *unicast*. Na comunicação *unicast* as mensagens são enviadas a apenas um único nó, o que gera a necessidade de retransmissão de uma mesma mensagem (n-1) vezes, sendo n o número de nós do sistema. Para aperfeiçoar este tipo de comunicação foi criado o protocolo de comunicação *multicast*. Em comunicações *multicast*, é possível enviar uma mensagem para diversos nós simultaneamente, permitindo economizar tempo e largura de banda de transmissão (KUROSE, ROSS, 2001). Segundo os estudos de Jimenez-Perez *et. al.* (2001), a utilização da replicação síncrona através de um protocolo *multicast* elimina grande parte do *overhead* de comunicação.

A substituição da replicação pela invalidação de dados permite reduzir o *overhead* de comunicação ainda mais, pois sistemas utilizando esta técnica não replicam os dados alterados para os outros nós, mas enviam mensagens para invalidar estes dados nos demais nós. Quando um nó recebe uma destas mensagens, o dado ao qual ela se refere é descartado, sendo removido do cache caso esteja nele armazenado. Como estas mensagens costumam ser menores do que as de replicação, elas provocam um *overhead* menor para a aplicação. Esta técnica permite manter os dados sincronizados entre todos os nós da aplicação com um custo menor para a aplicação.

Em aplicações nas quais os dados não precisam estar sincronizados o tempo todo, é possível utilizar técnicas de replicação ou invalidação assíncrona. Na replicação assíncrona as alterações nos dados são armazenadas localmente no nó que as originou, e somente são transmitidas para os demais nós quando um número pré-determinado de alterações locais tiver ocorrido, ou em intervalos de tempo pré-definidos. Esta técnica diminui o *overhead* de comunicação na aplicação em relação ao modelo síncrono ao aproveitar a abertura de um canal de comunicação para transmitir um número maior de alterações. O tempo de resposta para os usuários – em relação aos métodos síncronos – também é reduzido, pois a aplicação não precisa mais esperar o término da replicação/invalidação para enviar a resposta para o cliente.

A invalidação assíncrona, por sua vez, só permite que os dados permaneçam em cache por um período de tempo pré-determinado, sendo descartados do cache ao final deste período. Em aplicações nas quais a versão atual da informação não seja tão importante – tais como jornais *on-line* e sistemas de consultas de cotações de ações (não voltadas para o comércio) – esta técnica pode ser utilizada sem prejuízo para seus usuários.

A utilização da invalidação assíncrona em aplicações cujos usuários necessitem da versão atual dos dados – tais como sistemas de compra e venda de ações – é desencorajada. Quando um sistema de cache distribuído é utilizado em uma aplicação destas, torna-se necessário a utilização de uma das técnicas síncronas descritas acima para garantir que todos os nós trabalhem sempre com a versão atual dos dados.

2.2 Cache de componentes de negócio

Conforme visto na seção 2.1, sistemas de cache são largamente utilizados para armazenar dados obtidos de repositórios, posicionando o sistema de cache entre a camada de persistência e o repositório físico de dados. Outro ponto de aplicação para sistemas de cache é entre as camadas de vista e de negócio, armazenando os resultados das invocações dos métodos de negócio para evitar futuras invocações.

Enquanto a utilização de cache para dados persistentes propicia um aumento de desempenho através da desoneração do repositório de dados e da redução do tempo de acesso aos dados, sua utilização na camada de negócio da aplicação objetiva economizar tempo de processamento. Em aplicações em que os clientes acessam a camada de negócio remotamente, esta técnica permite eliminar o atraso provocado pela transmissão de invocações remotas.

Os principais trabalhos de pesquisa encontrados nesta área são os de Pohl, Göbel e Schill (2003) e os de Pfeifer e Jakschitsch (2003). Estes dois trabalhos são focados na aplicação de cache na invocação remota de métodos de componentes de negócio, na plataforma EJB (SUN, 2003a) sobre o servidor de aplicação JBoss.

Pfeifer e Jakschitsch aplicaram o sistema de cache através da manipulação dos *proxies* dos componentes remotos para injetar o sistema de cache de forma transparente para seus usuários e desenvolvedores. A geração dos *proxies* com o sistema de cache embutido é realizada através de uma ferramenta semelhante a um compilador, que encapsula os *proxies* comuns dentro dos *proxies* do sistema de cache. Desta forma, quando uma requisição não puder ser atendida pelo sistema de cache, ela é repassada ao *proxy* original.

Quando um método remoto é executado, o sistema de cache pode interceptar esta requisição e verificar se ela pode ser atendida utilizando os dados armazenados localmente. Se o resultado para esta requisição não estiver armazenado localmente, o sistema permite que a chamada percorra seu caminho natural e, ao retornar, o interceptador a armazena em cache automaticamente. A ligação das aplicações com os *proxies* gerados por este trabalho é realizada através de uma implementação proprietária da interface JNDI (SUN) `InitialContext`, que deve ser utilizada para obter as referências aos componentes remotos.

O trabalho de Pfeifer e Jakschitsch utiliza um mecanismo de invalidação de dados baseado em um modelo – representado em um descritor XML – que deve ser fornecido pelos desenvolvedores. Neste descritor estão enumerados os métodos de leitura e os métodos de escrita de cada componente remoto utilizado, sendo chamados de métodos de leitura aqueles que não causam alteração nenhuma ao sistema (em dados persistentes ou no estado interno de algum componente), e de métodos de escrita aqueles que causam. Além da classificação dos métodos, este modelo descreve a dependência entre os métodos de leitura e os métodos de escrita.

Sempre que um método de leitura for invocado, o sistema irá tentar atende-lo através dos dados armazenados em cache. As invocações de métodos de escrita, por sua vez, são sempre repassadas ao componente remoto e, no seu retorno, os dados de todos os métodos de leitura relacionados com este método de escrita são invalidados e removidos do cache.

Apesar de esta abordagem de invalidação de dados funcionar corretamente para um sistema de cache local, este sistema não está preparado para funcionar em uma aplicação que permita que um mesmo conjunto de informações seja utilizado em nós diferentes. Para funcionar corretamente nesta situação, é necessário definir uma forma para invalidar os dados em múltiplos clientes remotos sempre que um método de escrita for invocado por um deles. Sem este sistema de invalidação síncrona, alguns clientes poderão trabalhar com informações que não correspondem mais à situação atual.

Em seu trabalho, Pohl, Göbel e Schill, ao invés de criarem novos *proxies* para aplicar seu sistema de cache, adicionaram novos interceptadores à cadeia de interceptação dos *proxies* dos componentes remotos. No servidor de aplicação JBoss cada componente possui duas cadeias de interceptadores: uma do lado do cliente e uma do lado do servidor. Quando uma invocação é feita a um método remoto, as duas cadeias de interceptadores são percorridas na ida e na volta. Um interceptador instalado em uma destas cadeias pode ler e alterar os parâmetros passados, impedir que a requisição prossiga na cadeia de interceptação (abortando a execução do método) e alterar os valores retornados pelo método remoto.

Quando uma invocação de um método remoto é realizada, o interceptador do lado do cliente verifica se é possível atende-la utilizando os dados em cache. Se o método já tiver sido invocado anteriormente e seus resultados ainda puderem ser considerados válidos, a invocação remota é abortada e o valor obtido no cache é retornado para o cliente. Todo

resultado armazenado em cache é associado a um prazo de validade, que é utilizado para verificar se os dados devem ser considerados válidos ou não. Quando a requisição não pode ser atendida através dos dados armazenados em cache, o interceptador do lado do cliente anexa uma lista com os dados cujos prazos de validade tenham expirado.

Quando a invocação é repassada ao servidor, o interceptador de cache deste lado permite que ela seja repassada ao componente remoto diretamente. Ao retornar, este interceptador obtém a lista de dados expirados enviada pelo cliente e verifica se as versões dos dados do cliente ainda são válidas. Os dados cujas versões não mais são válidas têm seu prazo de validade expirado, mas as versões atualizadas destes dados não são enviados para os clientes. Ao invés disso, ao receber o retorno do servidor, o interceptador do lado do cliente remove os dados inválidos do cache.

Este trabalho é mais bem equipado para trabalhar com múltiplos clientes do que o de Pfeifer e Jakschitsch apresentado anteriormente, uma vez que ele utiliza um mecanismo de invalidação dos dados distribuído. Apesar disso, uma vez que esta invalidação é realizada assincronamente, ele não é recomendado para aplicações que não possam trabalhar com dados obsoletos.

Apesar de nosso trabalho também lidar com um serviço de cache transparente que possa ser utilizado a frente da camada de negócio, ele diverge destas pesquisas em diversos pontos. Enquanto ambos os trabalhos apresentados nesta seção usem um sistema de cache do lado do cliente de uma aplicação distribuída, nosso trabalho não estabelece uma localização específica para o uso do cache, podendo ser utilizado em aplicações distribuídas – tanto no lado do cliente como no lado do servidor – como também em aplicações centralizadas. Nossa proposta também inclui um serviço de cache para a camada de persistência, e não é limitado por nenhuma plataforma de desenvolvimento, sendo definido como uma arquitetura abstrata que pode ser implementada em qualquer plataforma, em qualquer linguagem de programação.

3 Mecanismos não-intrusivos para injeção de serviços

Neste capítulo iremos discorrer sobre as ferramentas utilizadas para injetar serviços de forma totalmente transparente em uma aplicação: programação orientada a aspectos e anotações. Estas duas tecnologias foram utilizadas para que a implementação do Crystalline mantivesse sua transparência e pudesse ser aplicada às aplicações pré-existentes sem que fosse necessário alterar o código-fonte das mesmas.

3.1 Anotações

Anotações são artefatos de código que permitem que metadados sejam anexados às classes e recuperados posteriormente em tempo de execução ou através da análise do código fonte. O conceito mais amplo de anotações surgiu na plataforma .NET da Microsoft, sob o nome de atributos. Nesta plataforma, as anotações são utilizadas para diversos objetivos, indo desde a formatação de código pela ferramenta de edição até a definição do modelo de *threads* utilizado por componentes. As anotações podem ser associadas a classes, métodos e campos.

O escopo das anotações pode ser de código-fonte ou de tempo de execução. No primeiro caso, as anotações só existem no código-fonte, e são descartadas pelo compilador. Anotações com este tipo de escopo são utilizadas por ferramentas que lêem o código-fonte e extraem informações, tais como geradores de documentação e geradores de código-fonte. Anotações com escopo de tempo de execução devem ser preservadas pelo compilador, e podem ser lidas pelas próprias aplicações através de operações de introspecção e reflexão em seus objetos.

É possível ainda definir campos para as anotações, permitindo personalizar seu uso através de valores diferentes para estes campos. Estes valores podem ser utilizados posteriormente por ferramentas específicas ou pela própria aplicação, que pode lê-los em tempo de execução e tomar decisões de acordo.

Na plataforma Java, as anotações foram inicialmente introduzidas por ferramentas como o JavaDoc – para geração de documentação da API de classes Java – e o XDoclet (2002) – para geração automática de código. Nestas duas ferramentas, as anotações (ou atributos) eram utilizadas dentro de comentários nos arquivos de código-fonte, e somente eram acessíveis através destes arquivos. O conceito completo de anotações passou a ser suportado pela linguagem Java somente a partir de sua versão 5.0 (SUN, 2004).

3.2 Programação orientada a aspectos

Mesmo que a maioria das classes de um modelo de objetos forneça uma funcionalidade única e específica, elas normalmente também contêm a implementação de requisitos secundários em seu código. Quando isso ocorre, dizemos que estes requisitos secundários são transversais aos requisitos principais. Um exemplo típico desta situação é a necessidade de manter um log da invocação de vários serviços diferentes, em camadas diferentes, o que normalmente implica a duplicação do código de execução do serviço de log ao longo da aplicação.

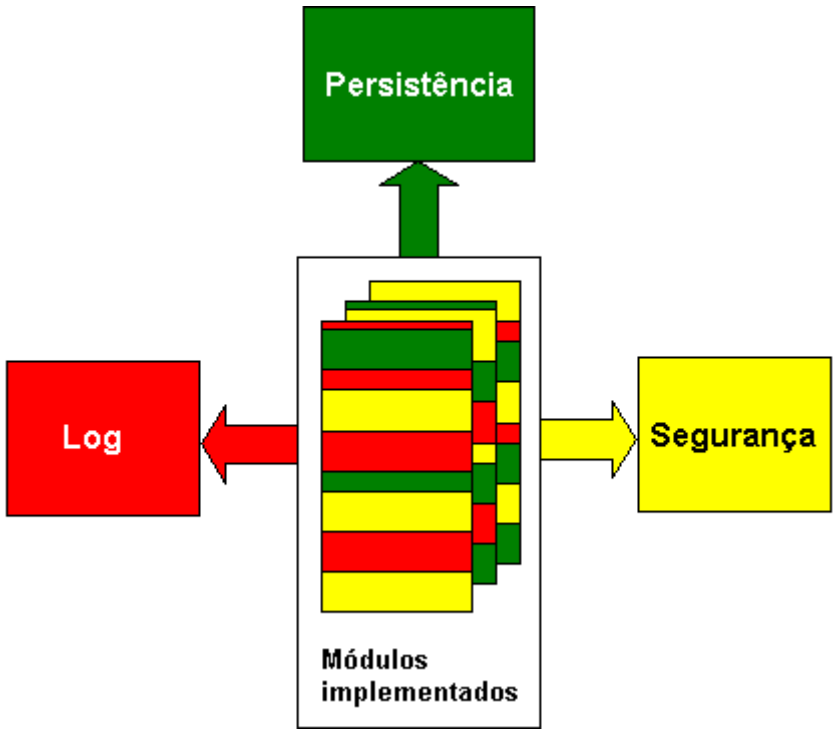

Figura 3.1 – Módulos de um sistema como um conjunto de requisitos

A figura 3.1 demonstra como os requisitos transversais (*crosscuting concerns*) se misturam ao código das aplicações, entremeando a implementação das regras de negócio com a implementação dos requisitos transversais. Segundo Kiczales *et. al.* (1997) e Laddad (2002) as principais implicações desta abordagem são:

- Obscurecimento da correspondência entre um requisito funcional e sua implementação, resultando em um mapeamento pobre entre os dois;

- A implementação de diversos requisitos simultaneamente desvia o foco dos desenvolvedores do requisito principal para os periféricos, levando a uma baixa produtividade;

- Menor reuso de código, devido à necessidade de duplicar o código da chamada aos módulos dos requisitos secundários;

- Baixa qualidade do código, que se torna confuso ao misturar a implementação dos vários requisitos em um único local;

- Evolução do sistema torna-se mais difícil devido à replicação de código e ao possível espalhamento da implementação de diversos requisitos transversais.

Uma das formas existentes para resolver estes problemas é através da programação orientada a aspectos (*Aspect Oriented Programming* – AOP), que permite que os requisitos transversais sejam modularizados. Este paradigma de desenvolvimento – criado para ser um paradigma complementar ao desenvolvimento orientado a objetos – prega que as funcionalidades ortogonais ao modelo de objetos sejam implementadas separadamente em um único artefato de *software* (LADDAD, 2002).

O desenvolvimento utilizando AOP engloba três fases distintas, representadas na figura 3.2. Na primeira fase, os requisitos são decompostos para que os requisitos transversais sejam identificados e separados dos requisitos funcionais do sistema. Na fase seguinte todas as regras de negócio são implementadas separadamente, sem adicionar nenhuma referência aos requisitos transversais.

Ainda durante esta fase de desenvolvimento, os requisitos transversais também são implementados e encapsulados em módulos individuais completos e totalmente independentes. Neste ponto do desenvolvimento o aplicativo está totalmente modularizado, e cada um dos módulos contém a implementação de apenas um requisito do sistema – funcional ou não-funcional – possibilitando um código limpo e enxuto.

Na fase final – recomposição dos aspectos – as regras de interação entre os módulos principais e os requisitos transversais são especificadas. Ao conjunto formado por um módulo que implementa um requisito transversal e pelas regras que definem onde e como o mesmo será utilizado ao longo da aplicação, dá-se o nome de aspecto.

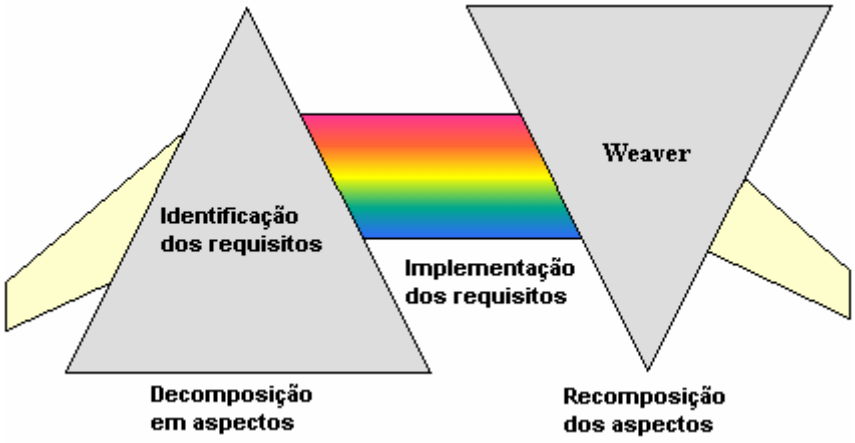

Figura 3.2 – Estágios do desenvolvimento AOP

Os métodos que compõem um aspecto são chamados de *advices*, e as regras de utilização destes são chamadas de pontos de junção (*join point*), que são pontos da execução do programa bem definidos tais como, por exemplo, a execução de um determinado método ou a leitura e/ou escrita de um determinado campo. Esses pontos de junção são especificados através da utilização de linguagens específicas de cada *framework* AOP existente.

Para evitar que os desenvolvedores tenham que enumerar cada um dos pontos de junção manualmente – o que seria um processo cansativo e suscetível a erros, as linguagens AOP utilizam um mecanismo chamado *pointcuts*, que são expressões que permitem especificar diversos pontos de junção de uma única vez. Os *pointcuts* podem ser expressos através de uma lista de pontos de junção específicos, de uma expressão lógica, da utilização de expressões regulares, operadores booleanos, etc. Um exemplo da expressividade destas linguagens pode ser observado na figura 3.10.

```
execution(* MinhaClasse+.get*(..) ) || get(*MinhaClasse+.*)
```

Figura 3.3 – Exemplo de definição de um *pointcut*

A figura 3.3 exemplifica a criação de um *pointcut* utilizando a linguagem fornecida pelo *framework* AspectJ (LADDAD, 2003). O *pointcut* em questão define os seguintes pontos de junção:

- Execução de todos os métodos da classe “MinhaClasse” e suas subclasses cujo nome comece por “get”, independente do tipo e quantidade de parâmetros e do tipo de retorno;

- Leitura de todos os campos da classe “MinhaClasse” e suas subclasses, independente de seu nome, tipo e visibilidade.

O último passo do ciclo de desenvolvimento AOP é a criação do executável da aplicação, através da união de todos os módulos e aspectos. A esse processo dá-se o nome de entrelaçamento, em uma alusão ao processo no qual um tecelão une inúmeros fios desconexos em uma única teia coerente e firme, produzindo um tecido. De forma análoga ao processo de tecelagem, o *weaver* é responsável por unir os módulos e os aspectos em uma aplicação funcional.

A execução do *weaver* faz com que todos os módulos e aspectos do sistema sejam combinados em um produto final – seja este um único arquivo executável, um conjunto de bibliotecas e/ou classes – no qual os módulos individuais estão misturados aos aspectos. Este procedimento pode ser realizado de diversas formas tais como a compilação do código fonte das classes e dos aspectos ou a alteração das classes já compiladas para uni-las aos aspectos. Independente da técnica escolhida é importante ressaltar que apesar de o produto final deste processo tornar-se parecido com a figura 3.8, a separação é mantida em seu código-fonte.

3.2.1 JBossAOP

Este *framework* AOP foi desenvolvido pelo projeto JBoss, tornando-se parte integrante deste servidor de aplicação a partir de sua versão 4.0. A principal motivação para seu desenvolvimento foi possibilitar que objetos Java simples (*Plain Old Java Objects* – POJOs) pudessem usufruir dos serviços oferecidos pelo servidor de aplicação, tais como gerência de transações, segurança, cache e etc. (JBOSSAOP, 2003). Desde a sua fase inicial este *framework* fornece aspectos para incorporar estes serviços aos POJOs.

Essa característica do *framework* permitiu que aplicações simples pré-existentes passassem a executar dentro do servidor de aplicação e a utilizar os serviços oferecidos pelo mesmo sem que seu código-fonte precisasse ser alterado. Além disso, o JBossAOP passou a permitir que POJOs fossem disponibilizados para invocação remota de forma transparente, sem a necessidade da implementação de um EJB que servisse de ponte[11].

[11] Padrão de projeto *Bridge* (GAMMA *et. al.* 1995)

Esse *framework* utiliza a linguagem de programação Java para a definição dos aspectos, ao contrário do AspectJ (LADDAD, 2003), que define uma extensão da linguagem Java para este fim. Dentre as vantagens do uso da linguagem Java, podemos destacar a menor curva de aprendizado e a maior facilidade para entendimento de aspectos escritos para este *framework*. No JBossAOP os aspectos são definidos através de interceptadores específicos, escritos como classes Java simples.

A única regra imposta pelo JBossAOP para a criação dos aspectos está na definição dos *advices*. Os métodos dos interceptadores que funcionarão como *advices* devem ser declarados conforme exibido na figura 4.1.

```
Object nomeDoMétodo(Invocation object) throws Throwable
```

Figura 3.4 – Modelo de cabeçalho de método para um *advice*

O tipo específico do parâmetro `Invocation` irá depender do tipo de *advice* sendo definido, que pode ser leitura de um campo, escrita de um campo, invocação de um método, criação de um objeto, etc.

Além da transparência oferecida aos desenvolvedores, esse *framework* também oferece um alto nível de flexibilidade e facilidade na definição dos *joinpoints* ao utilizar um arquivo XML simples para tal. Sua linguagem de definição dos *pointcuts* é extremamente poderosa, permitindo definições amplas como, por exemplo, a interceptação de todos os métodos que recebam como parâmetro um objeto cuja classe tenha sido marcada com uma anotação pré-determinada.

Ao ler as configurações dos aspectos e seus *joinpoints* nesse arquivo XML, o JBossAOP utiliza o padrão de projeto *Chain of Responsability* (GAMMA *et. al.*, 1995) para criar cadeias de interceptadores com os aspectos. Sempre que a aplicação passar por um dos *joinpoints*, a corrente de interceptadores associada a ele é percorrida, e cada um dos seus interceptadores é acionado. Todo interceptador tem o poder para alterar os dados da operação, seu retorno e ainda definir se a invocação deve ou não ser realizada e lançar exceções. Por causa destas características, um interceptador pode realizar o tratamento de dados sendo escrito em um campo, verificando a validade do novo valor, e interrompendo a execução da escrita caso este valor seja inválido.

No arquivo de configuração, além da declaração dos aspectos e *joinpoints*, é possível definir a introdução de implementação de interfaces e a introdução de anotações em classes pré-existentes. A introdução dinâmica de anotações é uma característica muito importante para nosso trabalho, e foi o principal fator para a decisão de usar este *framework* AOP em detrimento de outros. A possibilidade de inclusão de anotações em classes pré-existentes sem a necessidade de alterar seu código-fonte é muito importante para a manutenção da transparência na utilização de nosso *framework*.

Outra vantagem oferecida por esse *framework* é a possibilidade de se aplicar os aspectos em tempo de execução, sem a necessidade de se pré-compilar as classes anteriormente. Para que esta característica possa ser utilizada, porém, é necessário que a aplicação tenha controle sobre o carregador de classes principal, para substituí-lo pelo carregador fornecido pelo JBossAOP. Quando o uso desse carregador não for possível, é necessário executar o compilador AOP fornecido pelo JBossAOP sobre as classes já compiladas para permitir o uso dos aspectos.

O Crystalline foi desenvolvido utilizando a programação orientada a aspectos, e o JBossAOP foi o *framework* utilizado para tal finalidade. O conjunto de *pointcuts* foi definido utilizando a linguagem disponibilizada por este *framework*.

4 Arquitetura para o framework de cache

Ao discorrer sobre as pesquisas na área de cache de dados e de cache de componentes – capítulo 2 – pudemos observar algumas deficiências nas propostas apresentadas, tais como falta de transparência ou aplicabilidade restrita a uma plataforma específica.

Segundo Borsato (2004), a transparência de um sistema de persistência pode ser medida pela quantidade de referências a esse sistema que estão presentes no código-fonte da aplicação principal. Segundo esta classificação, para que um sistema seja considerado totalmente transparente, não pode existir nenhuma referência a ele no código da aplicação principal. Podemos aplicar os conceitos básicos desta classificação em qualquer subsistema que preste outros serviços secundários, tais como log, controle de acesso, cache, etc.

Outra forma de classificação de um sistema auxiliar é através de sua aplicabilidade. Para que um sistema desse tipo possa ser aplicado em um número grande de situações diferentes é necessário que ou ele tenha sido desenvolvido visando ser genérico ou que essas diversas situações tenham sido consideradas em seu desenvolvimento. O nível de aplicabilidade de um sistema é diretamente proporcional ao número de situações em que ele pode ser aplicado.

Utilizando essas duas diretivas – transparência e aplicabilidade –, podemos dividir os sistemas auxiliares existentes atualmente em:

- sistemas intrusivos e sistemas transparentes, e;
- sistemas especialistas e sistemas gerais.

Os sistemas de cache transparentes normalmente estão associados a plataformas específicas, tais como servidores de aplicação e o *framework* Hibernate[12] (BAUER, KING, 2004). Nesses casos os sistemas de cache são totalmente transparentes para os desenvolvedores, mas seu uso é limitado a tipos específicos de componentes (sistemas especialistas).

Sistemas de cache com alto nível de aplicabilidade, por outro lado, costumam ser intrusivos, forçando os desenvolvedores a referenciar sua API no código de suas aplicações,

[12] O Hibernate é um *framework* utilizado no mapeamento objeto-relacional que abstrai o trabalho do desenvolvedor de criar a camada de persistência de suas aplicações.

tornando-as dependentes de tais sistemas. Esta dependência explícita implica não só em acoplamento entre as aplicações e o sistema de cache, mas também uma mistura entre a implementação das regras de negócio e de requisitos secundários (desempenho), o que torna o código mais confuso e, conseqüentemente, mais difícil de ser mantido.

Esta aparente incompatibilidade entre transparência e aplicabilidade pode ser explicada pela necessidade de se prever as situações nas quais o sistema de cache poderá ser aplicado. Quando um sistema de cache é projetado para uma aplicação ou plataforma específica, torna-se simples definir as situações em que o mesmo será utilizado, o que permite que o sistema aplique suas regras sem a interferência dos desenvolvedores. Já nos casos de sistemas de cache que não estão voltados para nenhuma aplicação ou plataforma específica, não há como determinar todas as situações em que poderá ser aplicado. Por este motivo, a forma de utilização do sistema de cache é deixada a cargo do desenvolvedor, que deverá implementar o acesso ao sistema de cache dentro de sua aplicação.

A proposta deste trabalho é implementar uma camada de cache para componentes de persistência e de negócio que seja totalmente transparente para seus desenvolvedores, e com um alto grau de aplicabilidade. Doravante neste trabalho o termo “componente” será utilizado para designar todo e qualquer objeto que possa ser reutilizado, podendo ou não ser aderente a um modelo de componentes específico. Esta generalização do termo “componente” visa aumentar o número de situações em que a arquitetura aqui definida poderá ser utilizada, não vinculando a mesma a nenhuma plataforma ou linguagem de desenvolvimento de software.

No restante deste capítulo iremos enumerar os principais requisitos do *framework* de cache para, em seguida, detalhar a arquitetura do sistema de cache proposto neste trabalho.

4.1 Características do framework proposto

Nas próximas seções serão apresentadas as características que devem ser atendidas para que o *framework* proposto neste trabalho alcance um alto grau de aplicabilidade e, ao mesmo tempo, mantenha a sua transparência para os desenvolvedores.

4.1.1 Transparência para o usuário

A principal característica da camada de cache aqui proposta é ser totalmente transparente para os desenvolvedores, segundo a classificação de Borsato (2004). É necessário permitir que os desenvolvedores se preocupem em implementar somente os requisitos

funcionais de suas aplicações, e permitir que os requisitos transversais sejam gerenciados por subsistemas externos de forma transparente.

A garantia de transparência permite tanto que os desenvolvedores continuem implementando seus componentes da mesma forma como sempre o fizeram, como também permite que eles utilizem a camada de cache junto a componentes de terceiros. Quando uma aplicação que possui componentes externos utiliza um sistema de cache intrusivo, é necessário ou alterar o código-fonte desses componentes ou criar um Adaptador (GAMMA *et al*., 1995) para acessar a camada de cache quando esses componentes forem utilizados.

Outra vantagem obtida pela transparência é a possibilidade de uso da camada de cache em aplicações pré-existentes. A transparência da camada de cache permite que a mesma seja utilizada sem que seja necessário realizar alterações no código-fonte de tais aplicações. Outro ponto positivo de se utilizar uma camada transparente de cache é a compatibilidade com os padrões existentes, permitindo que tal camada seja utilizada tanto em componentes que aderem a uma especificação qualquer como em componentes que não aderem a nenhuma especificação.

4.1.2 Gerênciamento dos dados dos componentes de persistência

Para evitar acessos desnecessários ao repositório de dados pela aplicação, é necessário armazenar os dados dos componentes de persistência localmente. Uma vez disponíveis no cache, é possível utilizá-los para recarregar uma instância específica do componente, sem que seja necessário acessar o repositório de dados. Esta estratégia possibilita uma economia de tempo e de recursos preciosos do sistema como, por exemplo, conexões com o repositório de dados e poder de processamento do servidor que o hospeda.

Uma vez que os dados estejam disponíveis no cache, é necessário mantê-los sincronizados, o que implica descobrir quando os dados de um componente sofrem alterações e replicá-las para o cache. Em um componente de persistência, existem quatro eventos básicos que devem ser interceptados: criação, carga, persistência e exclusão.

Os eventos de criação e carga de dados devem ser interceptados pela camada de cache para que as informações obtidas do usuário e do repositório, respectivamente, sejam armazenadas localmente para reuso futuro, evitando acessos desnecessários ao repositório de dados. O evento de persistência é solicitado por um componente quando ele deseja salvar as

alterações em seus dados no repositório. A interceptação deste evento é importante para que a consistência entre os dados locais e os dados do repositório seja mantida. O evento de exclusão é solicitado por um componente quando este deseja excluir os dados do repositório persistente. Neste momento, os dados em cache relativos a esta instância devem ser excluídos, para preservar a consistência entre o cache local e o repositório permanente.

4.1.3 Gerência de relacionamentos de componentes de persistência

Na maioria das aplicações do mundo real, existem inter-relações entre seus diversos componentes. Estas interdependências existem entre dois componentes de persistência, entre dois componentes de negócio e entre um componente de negócio e um componente de persistência. Estas inter-relações existem para permitir uma melhor organização das responsabilidades, isolando comportamentos comuns em componentes específicos e disponibilizando-os através de sua API.

Um sistema de cache para componentes deve levar em consideração os inter-relacionamentos com componentes de persistência, evitando carregar dados desnecessários. Um componente deve ser carregado rapidamente, para aumentar o desempenho geral das aplicações. Desta forma, se um componente faz referência a um ou mais componentes de persistência, deve-se permitir que estes componentes sejam carregados apenas quando forem necessários. Esta abordagem permite um ganho maior de velocidade e uma otimização do uso da memória ao não carregar objetos desnecessários.

A falha em gerenciar adequadamente os relacionamentos entre componentes de persistência pode trazer dois problemas distintos: carga total de uma sub-árvore de componentes e carga infinita de referências circulares.

A primeira situação pode ocorrer quando se tenta carregar um componente que contenha relacionamentos com outros componentes, que por sua vez também contenham outros relacionamentos com componentes. Se o sistema de cache tentar carregar os componentes de forma completa no momento em que este for requisitado, é possível que um grande número de componentes sejam carregados e não sejam utilizados, ocupando espaço em memória em detrimento a outros componentes.

A carga infinita de referências circulares, por sua vez, pode ocorrer ao carregar um grafo circular de componentes. Se o uso de carga sob demanda não for utilizado, corre-se o

risco de criar-se um *loop* infinito na carga destes componentes. Para evitar os problemas advindos destas duas situações, e para aumentar o desempenho geral da camada de cache, é necessário prever o gerenciamento de relacionamentos com componentes de persistência e a carga sob demanda.

4.1.4 Serviço configurável

Com o reuso de componentes torna-se difícil – se não impossível – determinar se um componente deve ou não utilizar um serviço de cache sem saber em que situações o componente será utilizado. Em um sistema de postagem de notícias, por exemplo, o componente que representa a entidade “notícia” efetuará muito mais leituras do que atualizações no repositório de dados, quando o mesmo estiver sendo utilizado na apresentação das notícias para os leitores. Neste cenário, o desempenho da aplicação provavelmente seria beneficiado pelo uso de um sistema de cache por este componente.

Nessa mesma aplicação, por outro lado, o mesmo componente “notícia” utilizado no cenário de postagem, provavelmente efetuará muito mais atualizações do que leituras no repositório de dados. Além disso, uma mesma instância de “notícia” será utilizada por menos usuários neste cenário do que no anterior. Neste caso de uso específico, há uma grande probabilidade de que o uso de um sistema de cache por este componente tenha um efeito negativo no desempenho do sistema, devido ao trabalho extra de atualizar constantemente os dados em cache e à baixa taxa de leitura destes.

Diversas aplicações apresentam cenários como os descritos nos parágrafos anteriores, nos quais os mesmos componentes têm respostas diferentes à utilização de um sistema de cache. Por este motivo, é interessante que a camada de cache possa ser configurada, de forma a ser ou não ativada em determinados cenários, estejam eles na mesma aplicação ou em aplicações distintas.

Para atender ao requisito de transparência (seção 4.1.1), é necessário que a configuração da camada de cache possa ser feita de forma totalmente independente do código. Atualmente a forma mais comum para este tipo de configuração é feita através da utilização de descritores XML, que podem ser facilmente estendidos caso seja necessário.

4.1.5 Gerenciamento dos dados de componentes de negócio

Como visto nas seções anteriores, o cache de dados dos componentes de persistência permite que a aplicação realize menos acessos ao repositório de dados, diminuindo o tempo de resposta da aplicação e, conseqüentemente, aumentando o seu desempenho. Outra aplicação possível para um sistema de cache é na camada de controle, possibilitando a reutilização de resultados de métodos de componentes de negócio previamente retornados.

O uso de cache em componentes de negócio se justifica para métodos complexos – que demandam uma carga de processamento muito alta – utilizados freqüentemente. Armazenar localmente os resultados de tais métodos resulta na eliminação de processamento redundante, o que permite não apenas reduzir o tempo de resposta da aplicação como também economizar recursos do sistema.

Definimos as seguintes regras para classificar os métodos de negócio de acordo com a sua dependência do estado interno do objeto:

- métodos que dependem apenas dos parâmetros fornecidos, tais como o cálculo do n-ésimo número de uma seqüência de Fibonacci e o cálculo da raiz n-ésima de um número;

- métodos que dependem do estado interno do objeto e dos parâmetros fornecidos, tais como o cálculo do frete de um Pedido, baseado no número de itens que este contém;

- métodos que dependem do estado interno do próprio objeto, do estado interno de outros componentes e dos parâmetros fornecidos, tais como o cálculo da média das notas de uma Turma, baseado nas notas de cada um de seus Alunos.

Para realizar o cache de componentes de negócio de forma eficiente, é necessário identificar em qual categoria um método se encontra e, no caso dos métodos que dependem de algum estado interno, de que campos e/ou componentes eles dependem. Os resultados em cache devem estar associados aos seguintes artefatos: componente, método, valores dos parâmetros e, quando aplicável, valores dos campos dependentes – tanto do próprio componente como de outros.

4.1.6 Gerenciamento dos resultados de pesquisas (finder methods)

Muitos padrões de componentes de persistência – tais como EJB (SUN, 2003a) e JDO (SUN, 2003b) – especificam que os componentes solicitam as informações ao repositório de dados através de métodos de busca. Estes métodos definem como os usuários destes componentes podem pesquisar as informações e, normalmente, apresentam-se sob duas formas: métodos que retornam uma única instância de um componente e métodos que retornam uma coleção de componentes que atendem às especificações da pesquisa em questão.

Os métodos que retornam uma única instância de um componente são aqueles que realizam a pesquisa através de alguma chave – tanto por chaves primárias como por chaves alternativas. Exemplos deste tipo de método seriam: busca de Aluno por seu número de registro e busca de Pessoa Física pelo seu CPF.

A outra classe de métodos de busca – aqueles que retornam uma coleção de objetos – normalmente realizam pesquisa por parâmetros mais extensos, tais como: busca de Livros de um Autor, busca por Alunos inscritos em uma Turma e busca de Disciplinas do currículo de um Curso.

Em muitas aplicações, este tipo de método pode ser invocado inúmeras vezes passando-se os mesmos parâmetros de busca, o que implica que um resultado idêntico ao anterior será obtido, exceto quando houver ocorrido alguma alteração nos dados em questão. Nestas situações, pode ser proveitoso utilizar um sistema de cache para evitar que a mesma pesquisa seja enviada repetidas vezes ao repositório de dados, desonerando-o.

Para que seja possível a utilização de uma camada de cache neste tipo de método, é necessário não apenas que os resultados sejam armazenados localmente, mas também que estes resultados sejam gerenciados ao longo da execução da aplicação. Sempre que um novo conjunto de dados for criado, é necessário verificar se eles atendem aos requisitos de alguma das pesquisas armazenadas em cache e, em caso positivo, acrescentar o novo conjunto de dados ao resultado da pesquisa. Ao excluir um conjunto de dados, é necessário verificar a quais resultados eles pertenciam, e excluir suas referências desses. Ao persistir as alterações em um conjunto de dados, é necessário verificar a quais novos resultados os dados pertencem, e se estes ainda pertencem aos resultados a que pertenciam antes das alterações.

Para que esta gerência dos resultados de pesquisa possa ocorrer, é necessário que a camada de cache conheça a estrutura da consulta de alguma forma. Pode ser utilizada uma linguagem de consulta – a exemplo da EJB-QL do modelo EJB (SUN, 2003a) – ou a definição de meios alternativos como, por exemplo, a definição de classes auxiliares.

É preciso avaliar cuidadosamente se o cache de resultados de pesquisas será benéfico para as aplicações, pois a sobrecarga de processamento gerada pela manutenção dos resultados das consultas pode impactar negativamente em seu desempenho em cenários onde os dados sofrem muitas atualizações.

4.2 Arquitetura proposta

A arquitetura proposta neste trabalho foi desenvolvida visando atender a todas as características levantadas na seção 4.1, com seu foco principal na transparência para os desenvolvedores e usuários finais. Tal arquitetura foi concebida sob a forma de quatro módulos distintos, que se comunicam com um sistema central de cache, conforme apresentado na figura 4.1.

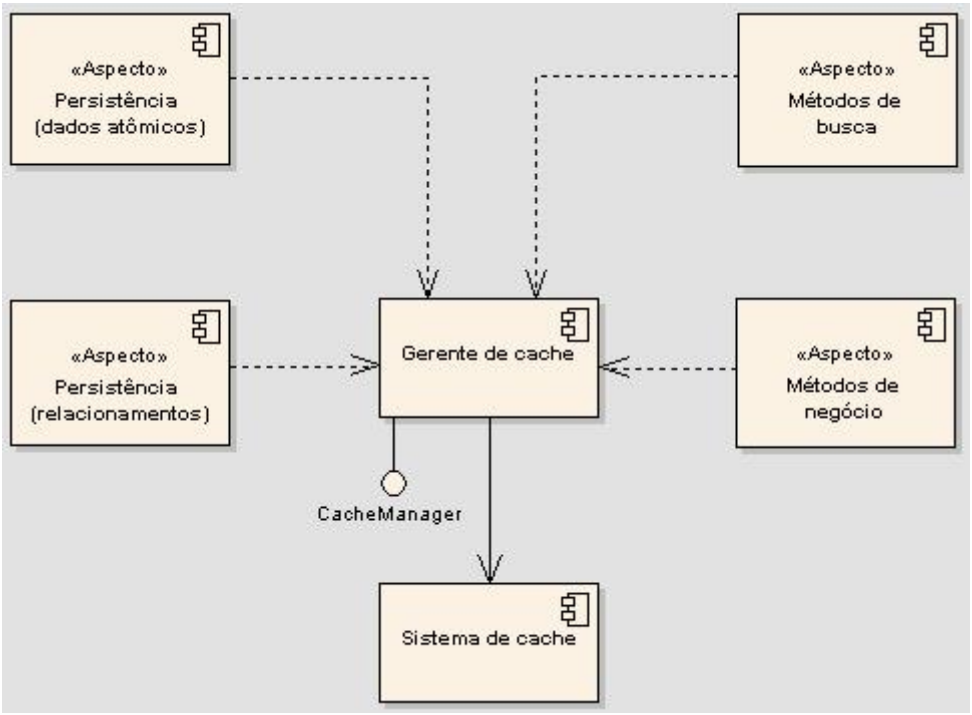

Figura 4.1 – Arquitetura do Crystalline

Quatro módulos básicos foram definidos na arquitetura do Crystalline, cada qual com uma responsabilidade específica dentro do sistema de cache: persistência de dados atômicos,

persistência de relacionamentos entre componentes, métodos de busca e métodos de negócio. Além destes quatro módulos, há ainda um componente responsável pela gerência de cache dentro do Crystalline.

O Crystalline foi projetado para atender às necessidades de uma aplicação multicamada típica, e tais aplicações normalmente são modeladas de forma semelhante ao exposto na figura 4.2. Esta figura servirá para evidenciar onde cada um dos quatro módulos do Crystalline se encaixa ao longo deste capítulo.

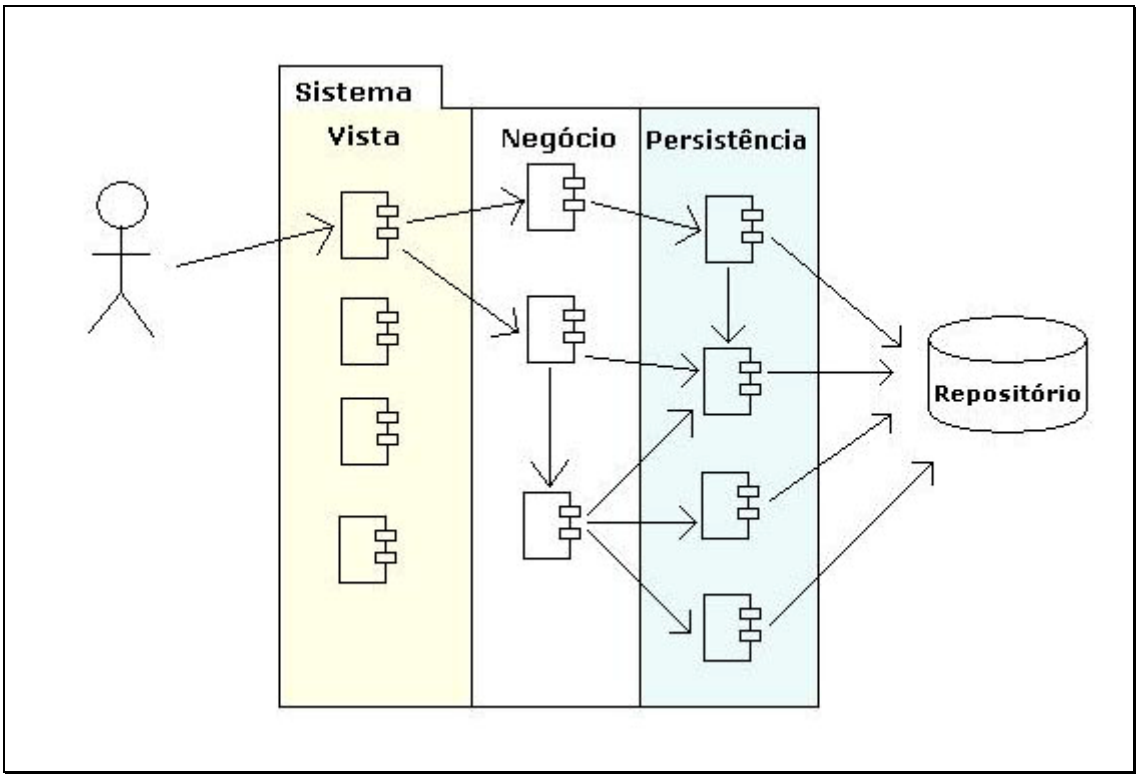

Figura 4.2 – Aplicação multicamadas típica

Em uma aplicação multicamada podemos observar que, de forma geral, os componentes de uma camada só se comunicam com componentes das camadas fronteiriças. Desta forma, a camada de vista apenas se comunica com a camada de negócio, que por sua vez pode se comunicar com ambas as camadas de vista e persistência. Os componentes da camada de persistência são os únicos a se comunicar com o repositório de dados e, da mesma forma, apenas os componentes da camada de vista interagem com os usuários do sistema.

Em uma aplicação que siga este modelo, podemos implementar diversos módulos de cache em posições estratégicas, conforme demonstraremos ao longo deste capítulo. Em cada uma das subseções abaixo iremos descrever os módulos que juntos compõem o Crystalline.

4.2.1 Módulo de cache para componentes de persistência

Utilizamos o termo "componentes de persistência" para designar os objetos responsáveis por transportar os dados entre a aplicação e o repositório de dados. A aplicação do módulo responsável pelo cache deste tipo de componente pode ser observada na figura 4.3, posicionado entre as camadas de negócio e de persistência. Este módulo é o responsável pela gerência dos dados dos componentes de persistência em cache (seção 4.1.2) e por evitar acessos desnecessários ao repositório de dados.

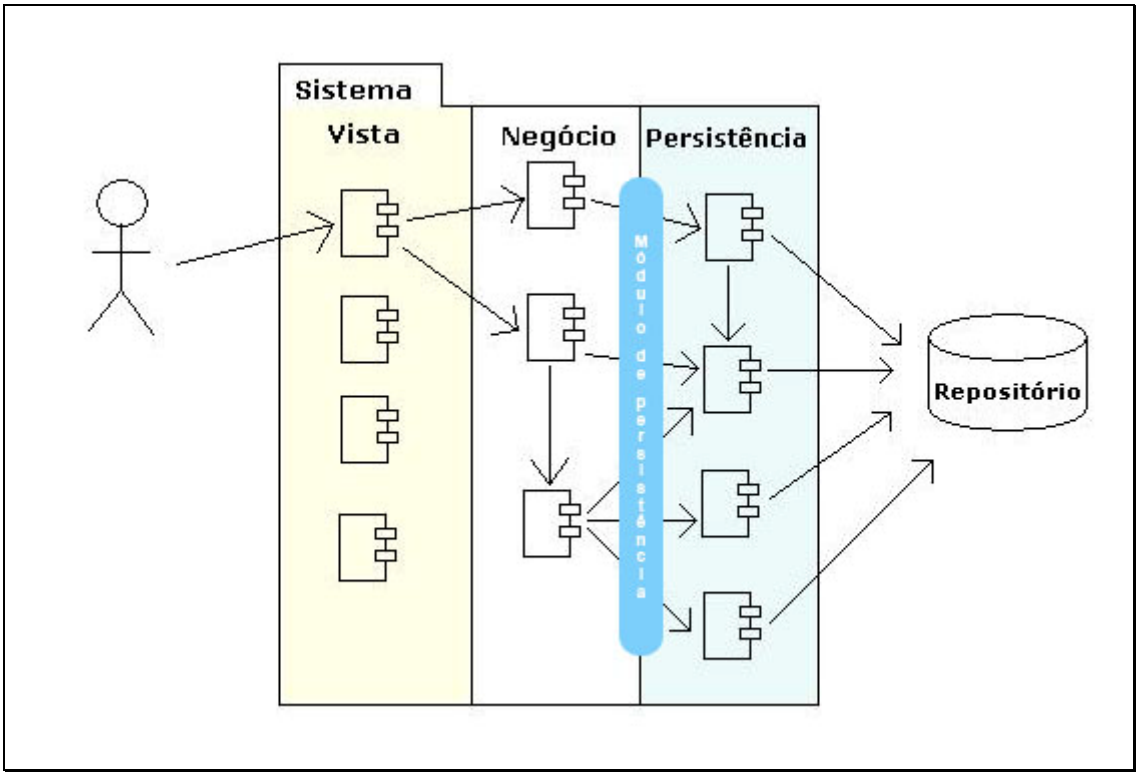

Figura 4.3 – Posicionamento do módulo de cache de componentes de persistência

Todas as chamadas aos componentes de persistência deverão passar por este módulo, para que este possa filtrar os acessos desnecessários ao repositório de dados, acessar os dados obtidos e gerenciar o seu armazenamento em cache. Acessos para criação, persistência e exclusão de dados sempre serão repassados ao repositório, para manutenção do sincronismo entre os dados armazenados em cache e os dados no repositório. A atualização dos dados em

cache deverá ser realizada após a execução destes métodos, de forma que a atualização só ocorra quando a operação no repositório de dados for bem sucedida.

Acessos para leitura, por sua vez, devem ser filtrados por este módulo. Apenas as requisições que não possam ser atendidas utilizando os dados em cache devem ser repassadas para o repositório de dados. A carga de componentes de persistência cujos dados já se encontrem em cache serão respondidas pelo Crystalline, que retornará uma instância do componente de persistência carregada com os dados correspondentes. Quando os dados necessários para carga de um componente de persistência não se encontram em cache, a requisição é repassada para o repositório de dados. Neste ponto, os dados obtidos pela aplicação são armazenados em cache para futuras referências.

O armazenamento dos dados em cache e o posterior filtro das requisições de carga utilizam o conceito de chave primária para diferenciar os diversos objetos gerenciados – termo que doravante denominará um objeto ou uma classe que sejam gerenciados por algum módulo do Crystalline.

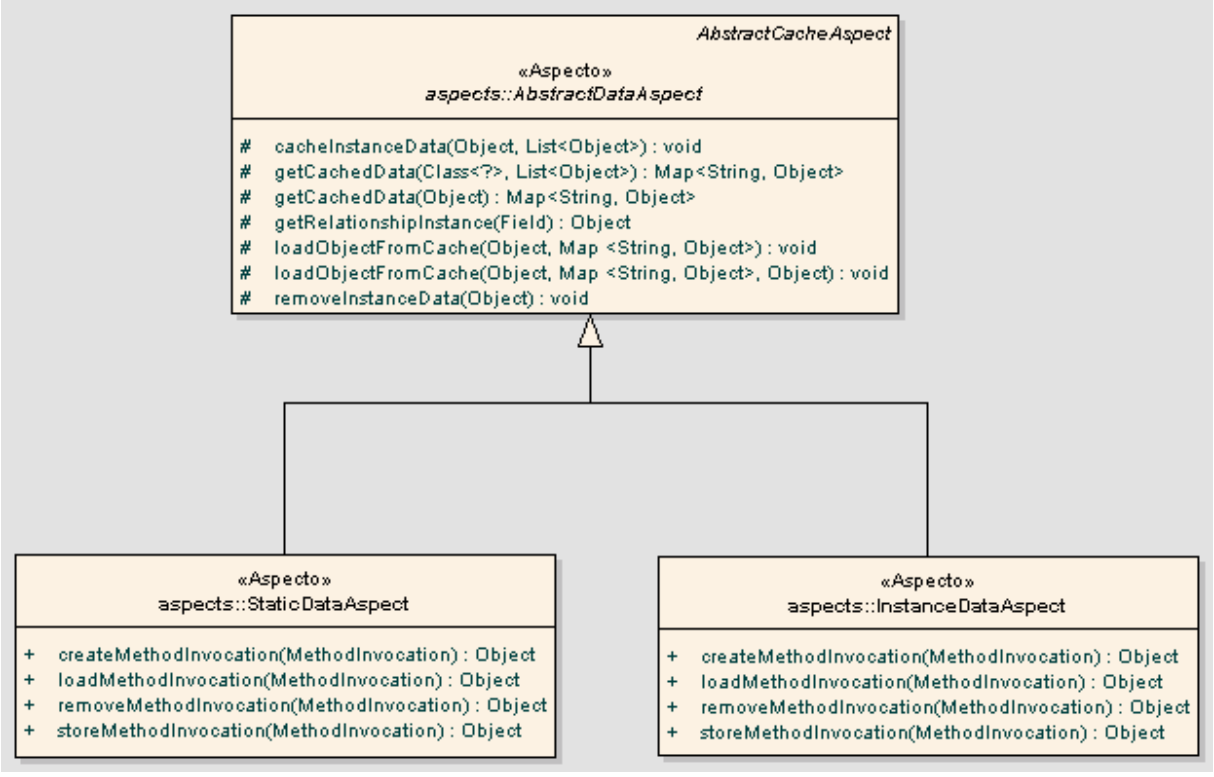

Figura 4.4 – Arquitetura do módulo de cache de componentes de persistência

A figura 4.4 representa o diagrama de classes deste módulo, que foi desenvolvido para atender a duas estratégias de criações de objetos: métodos fábrica (GAMMA, 1995) responsáveis pelo ciclo de vida de um componente de persistência (métodos de classe) e métodos de gerência do ciclo de vida localizados no próprio componente (métodos de instância). As duas classes concretas deste módulo – respectivamente StaticDataAspect e InstanceDataAspect – implementam aspectos para interceptação dos quatro tipos de métodos de ciclo de vida de um componente, e a classe abstrata AbstractDataAspect contém os métodos comuns a essas duas classes, e que podem ser utilizados por futuras implementações de estratégias diferentes para gerência do ciclo de vida dos componentes.

4.2.2 Módulo de cache para relacionamentos de componentes de persistência

O procedimento detalhado ao longo da seção 4.2.1 funciona corretamente para dados atômicos (dados que não representam outros objetos gerenciados), mas referências a outros componentes gerenciados devem ser tratadas de forma diferente. Este tratamento diferenciado é necessário para garantir que os dados de um objeto gerenciado referenciado por outros objetos gerenciados não sejam espalhados por toda a árvore de cache, ocupando memória desnecessariamente e dificultando sua manutenção.

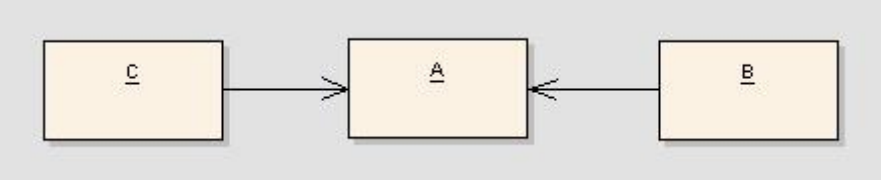

Figura 4.5 – Inter-relacionamento de objetos gerenciados

Conforme demonstrado na figura 4.5, se o mesmo objeto gerenciado A é referenciado pelos objetos gerenciados B e C, é necessário uma forma de garantir que, quando os dados de A forem atualizados, os dados em cache dos componentes B e C não se tornem inconsistentes. Uma forma de se garantir isso seria armazenar uma lista dos objetos que referenciam A e, quando este objeto fosse persistido, os dados em cache de todos os objetos que o referenciam também fossem atualizados. Esta técnica, além de ocupar memória excessivamente com a replicação dos dados de A ao longo de toda a árvore de cache, é ineficiente por efetuar múltiplas atualizações da árvore de cache quando um único objeto gerenciado for persistido.

Uma técnica mais eficiente é implementar o conceito de chaves estrangeiras de bancos de dados relacionais. Sempre que um objeto gerenciado faz referencia a outro, o sistema de cache armazena apenas a sua chave primária junto aos dados do primeiro objeto. No caso do exemplo acima, ao armazenar os dados dos objetos gerenciados B e C em cache, apenas a chave primária de A é armazenada junto com os dados de B e C. Quando o Crystalline for carregar B a partir da árvore de cache, ele obtém a chave primária de A dos dados de B e, quando necessário, a utiliza para carregar o objeto A normalmente.

Um problema comum ao carregar objetos inter-relacionados é a quantidade de objetos carregados. A carga de um objeto do repositório de dados pode disparar a carga de todos os seus relacionamentos, que por sua vez dispararia a carga de todos os objetos relacionados a estes, e assim por diante. A árvore de objetos carregada desta forma pode consumir um espaço considerável de memória e um longo tempo de processamento, e muitas vezes a maior parte dos objetos carregados não seria utilizada.

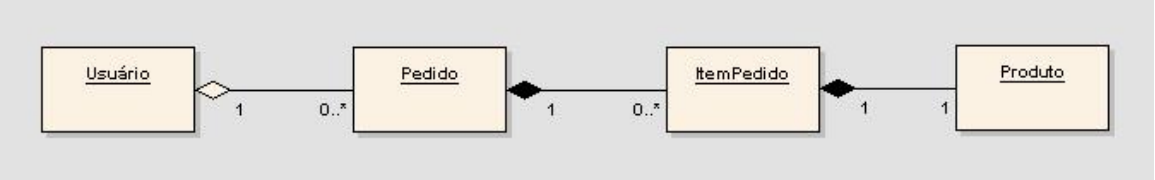

Figura 4.6 – Árvore de objetos gerenciados

Um exemplo deste tipo de cenário é demonstrado na figura 4.6, aonde um usuário de um sistema de compras tem um conjunto de pedidos, que são formados por conjuntos de ítens de pedidos, que representam um produto a venda. Se a carga dos relacionamentos se der de forma indiscriminada, ao carregar um usuário para verificação de login e senha, todos os seus pedidos, itens de pedidos e produtos comprados serão carregados e, muitas vezes, não serão utilizados. Uma das técnicas existentes para resolver este problema é a de *lazy-loading*, que consiste em somente carregar os objetos relacionados no momento em que estes forem utilizados.

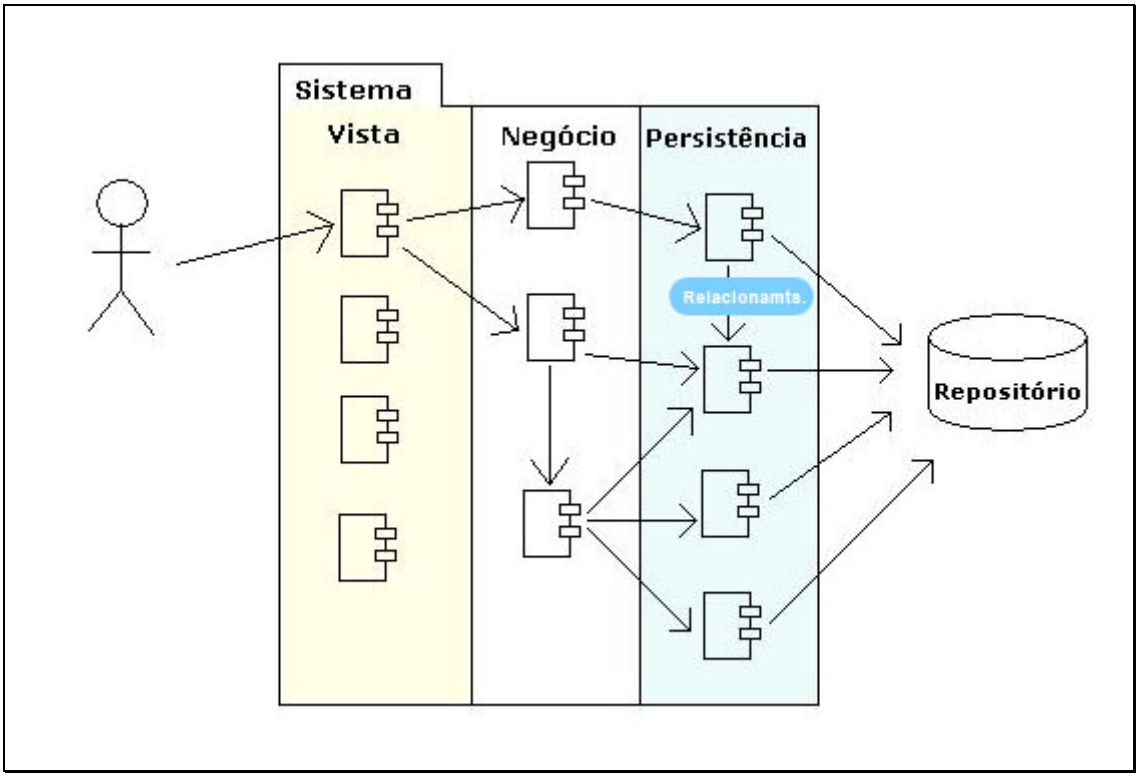

Figura 4.7 – Módulo de gerência de relacionamentos

Para tratar de todos os problemas advindos do relacionamento entre objetos gerenciados foi desenvolvido um módulo a parte para o Crystalline, que é responsável pela carga sob demanda (*lazy loading*) destes objetos e pela gerência de seus dados. A aplicação deste módulo é interna à camada de persistência, sendo ativado sempre que um relacionamento entre componentes desta camada for acessado, conforme exibido na figura 4.7.

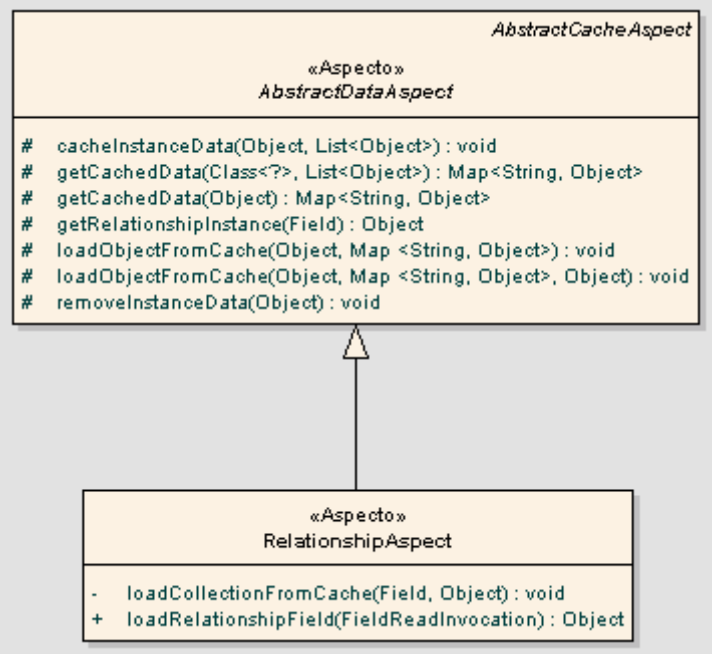

Figura 4.8 – Arquitetura do módulo de cache de relacionamentos entre componentes de persistência

A figura 4.8 representa o diagrama de classes deste módulo, composto por apenas uma única classe: RelatioshipAspect. Esta classe disponibiliza um único aspecto, que é responsável por interceptar as leituras aos campos de um componente de persistência que referenciam outros componentes deste tipo. É este aspecto quem implementa o *lazy-loading* das informações dos relacionamentos, tanto para relacionamentos entre apenas dois componentes (1:1) como para relacionamentos entre múltiplos componentes (1:N).

4.2.3 Módulo de cache para métodos de negócio

Os módulos definidos nas seções 4.2.1 e 4.2.2 atendem a todos os requisitos necessários para um sistema de cache de componentes de persistência. Conforme descrevemos na seção 4.1.5, outra forma de se reduzir o tempo de resposta a uma requisição de um usuário é através do armazenamento dos resultados das invocações dos métodos de negócio dos objetos.

Este tipo de cache pode ser extremamente útil em métodos custosos que sejam utilizados repetidas vezes. Métodos deste tipo incluem aqueles que trabalham com componentes remotos – como, por exemplo, a invocação de um serviço *web* dos correios para calcular o valor de um frete para determinado CEP – ou métodos recursivos – como o cálculo de números de Fibonacci, aonde o n-ésimo elemento da série depende do cálculo dos dois elementos anteriores.

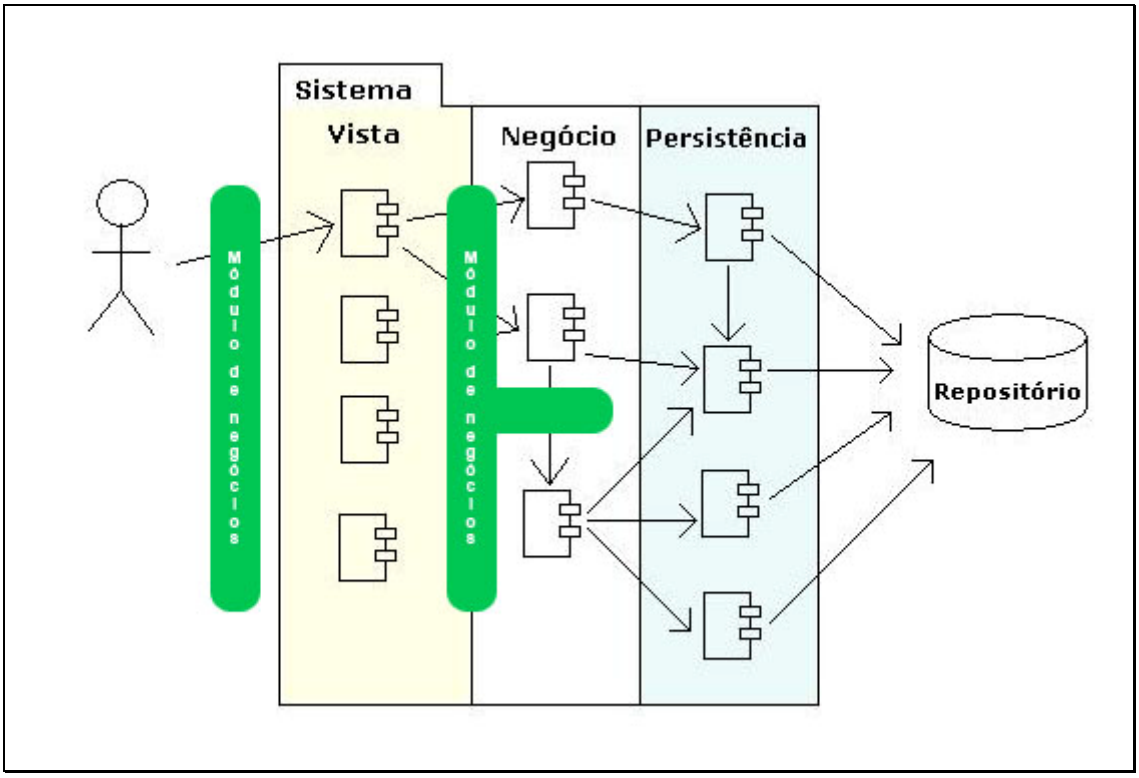

Figura 4.9 - Módulo de cache de métodos de negócio

O cache de métodos de negócio é gerenciado por um módulo a parte dentro da arquitetura do Crystalline. Este novo módulo deverá interceptar as chamadas aos métodos gerenciados (métodos que serão submetidos à camada de cache) antes de sua execução (figura 4.9). Ao interceptar a chamada a um destes métodos, este módulo verifica se o seu resultado está presente nos dados armazenados em cache e, neste caso, poderá retorná-lo diretamente, impedindo a execução do método. Caso o resultado do método em questão não se encontre em cache o mesmo deve ser executado normalmente, e seu resultado armazenado em cache.

Conforme podemos observar na figura 4.9, também é possível utilizar este módulo de cache de métodos de negócio na camada de vista, em ambientes *web*. Nestes cenários, este módulo funcionaria como um servidor *proxy* inteligente, que devolveria os resultados para o usuário de acordo com a página solicitada e os parâmetros fornecidos, por exemplo.

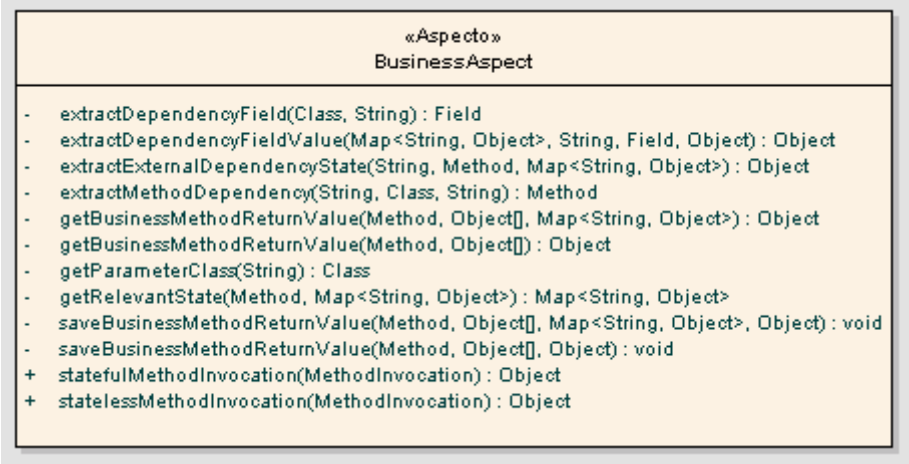

Figura 4.10 – Arquitetura do módulo de cache de métodos de negócio

A figura 4.10 exibe a classe BusinessAspect, que é a responsável pelo cache de métodos de negócio e a única classe existente neste módulo. Esta classe disponibiliza dois aspectos para interceptação dos métodos de negócio, sendo um para métodos que não dependem do estado do componente e outro para aqueles que dependem.

4.2.4 Módulo de cache para métodos de busca

Este módulo do Crystalline permite a realização de caches dos resultados de métodos de busca, conforme delineado na seção 4.1.6. Para adicionar esta funcionalidade ao Crystalline este módulo intercepta as chamadas a estes métodos dos componentes de persistência, e gerencia seus resultados através da interceptação dos métodos de ciclo de vida destes mesmos componentes.

Quando um método de busca for executado pela primeira vez, este módulo armazenará os seus resultados em cache, associando-os com os parâmetros de pesquisa utilizados em sua execução. Na próxima vez que este método for invocado com os mesmos parâmetros, os resultados armazenados serão retornados diretamente ao usuário, sem a necessidade de executar a consulta no repositório de dados.

Para garantir que os resultados das pesquisas armazenados em cache estejam sempre consistentes, é necessário verificar se um componente de persistência pertence a estes resultados sempre que alguma operação é realizada sobre este componente. Desta forma, este módulo precisa interceptar a execução dos métodos de ciclo de vida (criação, alteração e exclusão) e atualizar os resultados de métodos de pesquisa armazenados, incluindo ou excluindo o componente dos resultados conforme necessário.

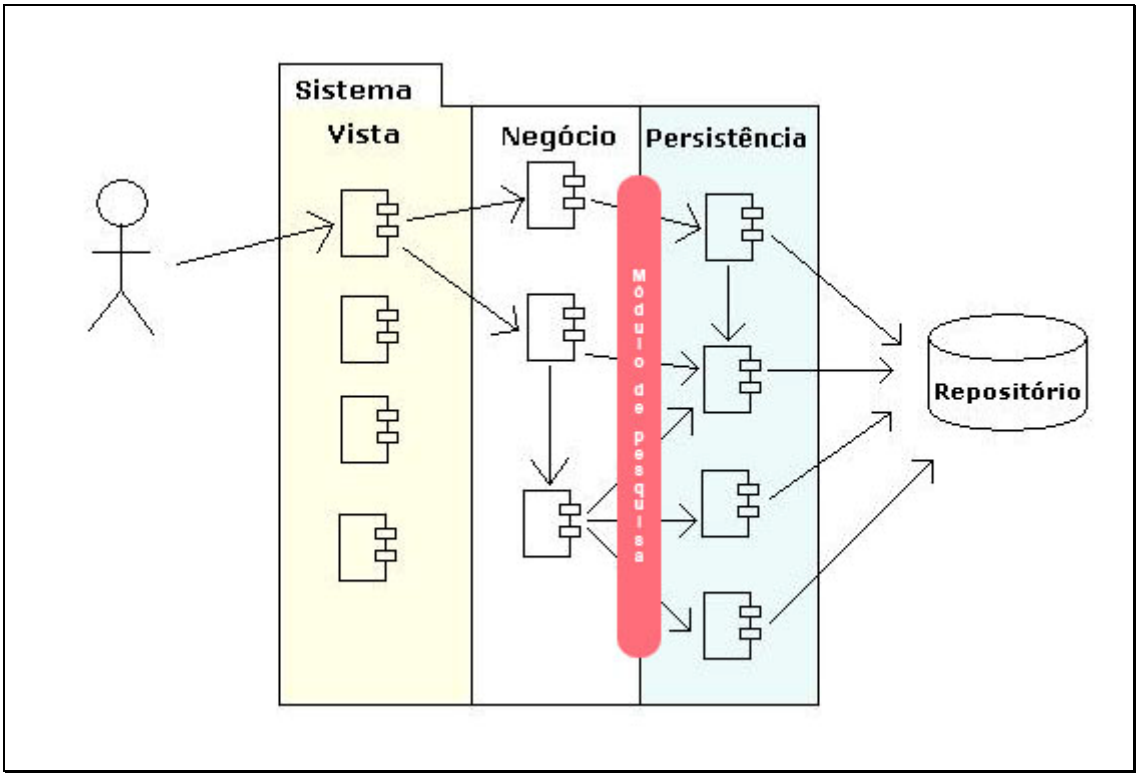

Figura 4.11 – Módulo de cache de métodos de busca

Conforme podemos observar na figura 4.11, o módulo de cache para métodos de busca intercepta a execução dos métodos dos componentes de persistência para gerenciar os resultados dos métodos de busca. Comparando esta figura com a figura 4.3, podemos observar que os dois módulos atuam de forma sobreposta, mas complementar, conforme demonstrado na figura 4.12. Esta figura exibe a representação da classe FinderMethodAspect, e nos permite observar que três dos quatro aspectos definidos por esta classe têm a mesma assinatura aspectos de cache de componentes de persistência (figura 4.4). O outro aspecto definido por esta classe é utilizado para interceptar a execução dos métodos de busca dos componentes.

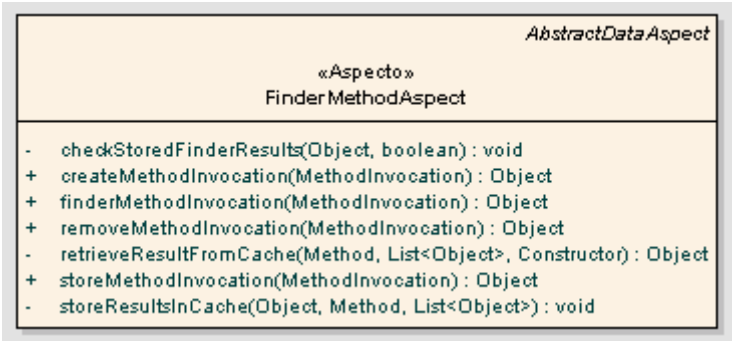

Figura 4.12 – Arquitetura do módulo de cache de métodos de pesquisa

4.2.5 Gerente de cache

O Crystalline foi concebido para ser uma camada transparente de comunicação entre as aplicações e um sistema de cache de terceiros – tal como o EHCache (2005) ou o TreeCache (JBOSSCACHE, 2003) – e o gerente de cache é o responsável por conectar os demais módulos ao sistema escolhido. Sua interface provê métodos para armazenar e recuperar os dados do sistema de cache específicos para cada um dos demais módulos de cache, isolando-os da forma como os dados são armazenados no sistema de cache. O isolamento do acesso ao sistema de cache escolhido é importante para que, futuramente, o Crystalline possa contar com sistemas de cache intercambiáveis, sem que para isso seja necessário alterar os aspectos desenvolvidos.

O acesso ao gerente de cache é fornecido por um método de classe deste componente, que foi implementado como um Singleton (GAMMA *et. al.*, 1995). Outras formas de implementação do acesso a este componente foram consideradas, tais como o uso de JNDI (SUN), mas foram descartadas devido ao fato que estas estratégias imporiam um acoplamento a uma plataforma específica, limitando a gama de aplicações que poderiam se beneficiar do uso do Crystalline.

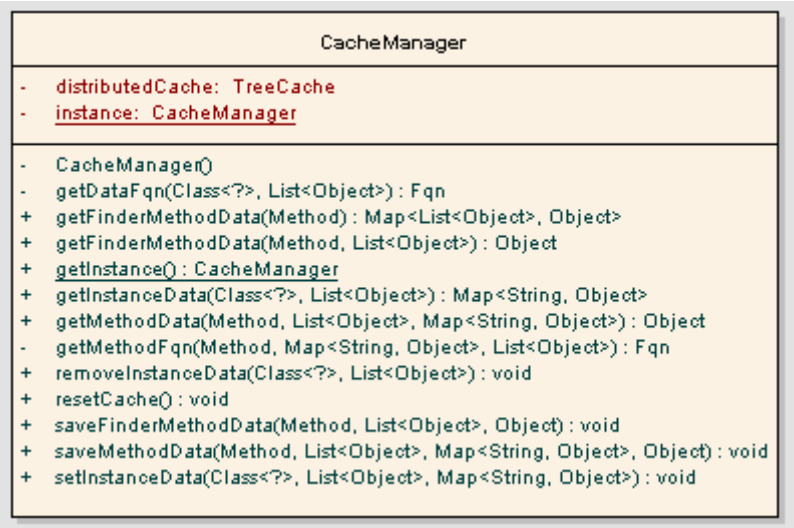

Figura 4.13 – Arquitetura do gerente de cache

Como podemos visualizar na figura 4.13, optamos por utilizar como sistema de cache o TreeCache (JBOSSCACHE, 2003) no Crystalline. Optamos por esta implementação em detrimento de outras por esta apresentar algumas características que julgamos importantes, tais como: suporte a distribuição de dados, suporte a transações (inclusive a transações

distribuídas), API simples, e por ser uma solução *open-source*. Este sistema de cache disponibiliza uma estrutura de dados na forma de árvore, na qual podem ser armazenados dados em todos os seus nós, e não apenas em suas folhas.

4.2.6 Crystalline como um framework transparente

Conforme delineado na seção 4.1.1, o Crystalline foi projetado visando ser totalmente transparente para usuários e desenvolvedores, funcionando como um serviço externo que pode ser integrado às aplicações sem a necessidade de alteração no código-fonte das mesmas. Para tornar o Crystalline totalmente transparente consideramos aplicar as seguintes estratégias: cadeias de interceptadores e programação orientada a aspectos. A utilização de interceptadores possivelmente tornaria o *framework* mais simples, mas o deixaria dependente de algum *middleware*, tal como o servidor de aplicação JBoss. A utilização da programação orientada a aspectos, por outro lado, permitiria que o Crystalline fosse utilizado por qualquer aplicação, de forma totalmente independente de *middleware*.

Outra tecnologia que nos auxiliou na criação do Crystalline como um *framework* totalmente transparente foi a tecnologia de anotações, que nos permite definir os metadados necessários para que o Crystalline possa identificar corretamente os diversos tipos de métodos que devem ser interceptados.

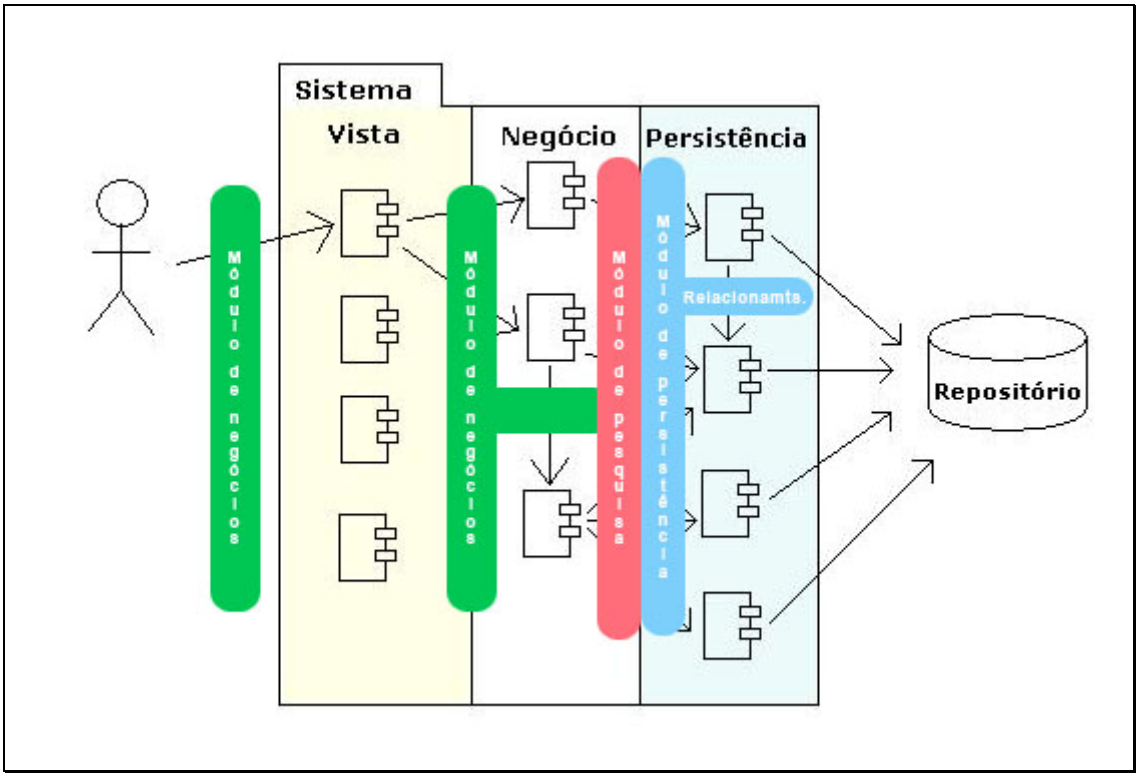

Figura 4.14 – Arquitetura completa do Crystalline

A arquitetura modular proposta para o *framework* Crystalline apresentada na figura 4.1 pode será implementada como um conjunto de aspectos independentes. A própria natureza da programação orientada a aspectos favorece o desenvolvimento do Crystalline desta forma, por tornar trivial a interceptação dos métodos necessários.

As anotações são utilizadas em nosso trabalho para demarcação do escopo de atuação (pontos de junção) de cada um dos quatro módulos do Crystalline. Através de um conjunto de anotações criadas especificamente para este sistema, os desenvolvedores poderão marcar quais classes, métodos e campos devem ser gerenciados pelo sistema de cache, simplificando a especificação dos metadados necessários pelo Crystalline e a própria declaração dos pontos de junção.

A figura 4.14 exibe todos os módulos criados para a implementação do Crystalline e a forma como os mesmos interagem com as aplicações que utilizarem este *framework*. Como podemos observar, a arquitetura do Crystalline nos permite aplicar um sistema de cache em todas as comunicações entre as diversas camadas das aplicações, e até mesmo nas comunicações existentes dentro de uma única camada.

5 Implementação do Crystalline

No capítulo 4 foi definida a arquitetura do Crystalline, formada por quatro módulos distintos, responsáveis pelo cache de componentes de persistência, pelo relacionamento entre componentes deste tipo, cache de métodos de busca e cache de métodos de negócio. Estabeleceu-se também o desenvolvimento destes módulos como um conjunto de aspectos, ligados às aplicações através de *joinpoints* definidos utilizando um conjunto de anotações predeterminadas, fornecido junto com esse *framework*.

Neste capítulo iremos detalhar como os quatro módulos do Crystalline foram desenvolvidos, quais problemas foram encontrados durante seu desenvolvimento e quais as soluções adotadas para sanar estes problemas.

5.1 Implementação da transparência no Crystalline

Como o Crystalline foi desenvolvido como um conjunto de aspectos, os desenvolvedores somente precisariam definir os *join-points* aonde os mesmos deveriam agir para implementação da camada de cache. Infelizmente, a definição de tais *join-points* é extremamente suscetível a erros e, caso um ou mais módulos do Crystalline não fossem ativados, a aplicação poderia passar a trabalhar com dados obsoletos e/ou incorretos.

Visando tornar a utilização do Crystalline mais simples, foi definido um conjunto de anotações sobre as quais foram definidos os *join-points* utilizados. Com os *join-points* previamente definidos, a tarefa dos desenvolvedores passa a ser a definição das classes e métodos que deverão ser anotados, de acordo com a funcionalidade dos mesmos. O problema na colocação das anotações nas classes das aplicações passa a ser a falta de transparência para os desenvolvedores, que passariam a ter que alterar o código-fonte das aplicações para ativar o *framework* de cache.

A solução para este problema foi fornecida pelo JBossAOP (2003), que possibilita a introdução dinâmica de anotações em classes compiladas, o que permite a manutenção da transparência total do *framework* de cache, segundo a definição de Borsato (2004).

Resolvidos os problemas de manutenção da transparência e da simplicidade de uso do *framework*, torna-se necessário apresentar as anotações definidas e utilizadas pela camada de

cache. As anotações citadas na tabela 5.1 serão explicadas mais detalhadamente ao longo deste capítulo.

Tabela 5.1 – Conjunto de anotações disponibilizadas pelo Crystalline

Anotação	Propósito
@CacheClass	Definir as classes que utilizarão a camada de cache. Classes sem esta anotação não serão tratadas pela camada de cache.
@CacheField	Definir os campos atômicos que deverão ser gerenciados pela camada de cache. Campos atômicos sem esta anotação serão tratados como campos transientes.
@CreateMethod	Definir os métodos de criação de objetos no repositório de dados.
@FinderMethod	Definir os métodos de busca de objetos.
@LoadMethod	Definir os métodos de carga de objetos a partir do repositório de dados.
@RelationshipField	Definir um campo que contém uma referência a outro objeto gerenciado pela camada de cache.
@RelationshipMultipleField	Definir um campo que contém uma coleção de referências a outros objetos gerenciados pela camada de cache.
@RemoveMethod	Definir os métodos de exclusão de objetos do repositório de dados.
@StatefulBusinessMethod	Definir métodos de negócio cujos resultados dependem do estado interno do objeto e dos parâmetros fornecidos.
@StatelessBusinessMethod	Definir métodos de negócio cujos resultados dependem apenas dos parâmetros fornecidos.
@StoreMethod	Definir os métodos de persistência de objetos no repositório de dados.

Todas as anotações acima definem *join-points* nos quais os aspectos de cache devem interceptar o processamento normal da aplicação. Os desenvolvedores podem escolher a inserção dinâmica dessas anotações através do JBossAOP ou sacrificar a transparência e utilizá-las diretamente nas classes de suas aplicações. Seja qual for a forma escolhida, é necessário anotar corretamente as classes gerenciadas, para garantir o funcionamento adequado da camada de cache.

5.2 Cache de componentes de persistência

Normalmente, componentes de persistência contêm métodos para gerenciar seu ciclo de vida, que são responsáveis por criar, carregar, atualizar e remover os dados do repositório. Nosso *framework* de cache suporta duas formas distintas para declaração desse tipo de método: métodos de instância e métodos de classe. Quando os métodos de ciclo de vida são definidos como métodos de instância, é esperado que eles trabalhem com o próprio objeto cujo método está sendo invocado. Já nos métodos de classe, espera-se que o objeto alvo da operação seja passado como primeiro parâmetro do mesmo, exceto no método de carga.

A estratégia de se trabalhar com métodos de ciclo de vida de classe e de instância foi implementada para possibilitar que os desenvolvedores utilizem o *framework* de cache com objetos do tipo DAO (Data Access Object) (ALUR, CRUPI & MALKS 2003) ou com objetos de negócio com persistência auto-gerenciada, a exemplo dos EJBs (SUN, 2003a).

Objetos do tipo DAO são utilizados para encapsular os acessos às bases de dados, isolando-os do resto da aplicação. Quando DAOs são utilizados, estes se tornam responsáveis pela gerência do ciclo de vida dos objetos persistidos, disponibilizando métodos separados para cada um destes passos (criação, carga, atualização e exclusão). Apesar de nem todo DAO implementar os métodos de ciclo de vida como métodos de classe – uma vez que isso não é uma requisição desse padrão de projeto – definimos que nesta versão do *framework*, para facilitar o desenvolvimento, esses métodos deveriam ser métodos de classe em objetos deste tipo, funcionando como fábricas de objetos (GAMMA *et. al.*, 1995) persistentes.

5.2.1 Identificadores dos objetos em cache

Para ser possível recuperar os dados armazenados em cache quando necessário é preciso uma forma de identificá-los de maneira única e inequívoca. Para solucionar esse problema, foi utilizado o conceito de chave primária dos sistemas de gerenciamento de bancos de dados. Um objeto gerenciado é identificado pela camada de cache através de sua classe e de sua chave primária. A chave primária de uma classe gerenciada é obtida pela camada de cache através das informações armazenadas na anotação `@CacheClass`.

Essa anotação serve a dois propósitos: definir a chave primária de uma classe e marcar esta classe como uma classe gerenciada. Apenas instâncias de classes que possuam essa anotação serão gerenciadas pela camada de cache. Apesar de resultar em um maior trabalho

para os desenvolvedores, esse requerimento torna a utilização do *framework* de cache mais segura, pois torna mais difícil o esquecimento da declaração de uma chave primária.

A anotação `@CacheClass` define dois atributos opcionais: `type` e `pk`. O primeiro permite determinar se a chave primária é definida através de métodos ou de campos. Em classes cujas chaves primárias sejam definidas através de métodos, dois cuidados devem ser tomados com relação aos métodos que as compõem:

- Eles não podem receber nenhum parâmetro, pois a camada de cache não tem como descobrir os valores que devem ser passados para os mesmos;
- A invocação destes métodos não deve ocasionar a alteração do comportamento/estado interno do objeto em questão, sob o risco de o funcionamento da aplicação tornar-se errático quando o *framework* de cache for utilizado.

Chaves primárias definidas através de campos de um objeto não precisam de nenhum procedimento especial, podendo inclusive declarar os campos em questão como campos sem visibilidade externa (privados). O acesso aos campos ou métodos da chave primária é realizado através dos mecanismos de introspecção e reflexão da linguagem Java, que permitem obter os valores dos campos e os resultados dos métodos dinamicamente, independente da visibilidade dos mesmos. A definição do tipo da chave primária é feita através de duas constantes declaradas em `@CacheClass`: `PK_TYPE_FIELD` e `PK_TYPE_METHOD`. Por default, chaves primárias são definidas através de campos.

Além da definição do tipo da chave primária, é preciso definir os nomes dos campos ou métodos que a compõem. Através do atributo `pk`, é possível definir uma lista com os nomes dos mesmos, separados por ponto-e-vírgula. Se uma classe gerenciada não declarar uma chave primária, todas as suas instâncias serão tratadas como instâncias idênticas, e seus dados serão identificados somente através de sua classe. Este conceito é útil ao se tratar componentes de negócio, como será apresentado mais adiante neste capítulo.

5.2.2 Integração com o sistema de cache

Conforme mencionado na seção 5.2.1, a identificação dos dados em cache é realizada através do uso da chave primária do componente, que é composta por sua classe e por um

subconjunto de seus campos. A figura 5.1 exemplifica como os dados dos componentes de persistência são estruturados em cache, separados através de suas chaves primárias.

O primeiro nível de separação destes dados é feito através da utilização da sua classe (em nosso exemplo, representado pela classe `br.ufrj.nce.cache.classeGerenciada`), separando-os dos dados dos demais tipos de componentes de persistência. Abaixo de sua classe é definido um nó através do uso da string “#data”, que separa os dados persistentes do componente em questão dos dados associados à métodos de negócio e a métodos de busca (seções 5.4 e 5.5). Abaixo do nó “#data” os dados de diversas instâncias do componente são separadas em outros nós representados pelo valor dos campos que formam a sua chave primária. É neste último nível que ficam armazenados os dados persistentes, em um mapa que contém os valores de cada um dos campos persistentes associados ao seu respectivo nome (representado na figura 5.1 através do nó “dados”).

Figura 5.1 – Estrutura utilizada para armazenamento dos dados dos componentes de persistência em cache

5.2.3 Tratamento dos métodos de criação

A gerência de um componente de persistência pela camada de cache é iniciada através de sua carga ou de sua criação, momentos nos quais um objeto é armazenado em cache pela primeira vez. Para que o Crystalline reconheça os métodos de ciclo de vida de objetos persistentes como tal, é necessário anotá-los de acordo. Para tanto, foram criadas anotações para cada tipo de método de ciclo de vida, que serão descritos a seguir.

Conforme citado anteriormente, os métodos de criação correspondem a uma das duas formas de um objeto tornar-se um objeto gerenciado. É esperado que, ao ser executado, esse método insira as informações do objeto no repositório, criando uma nova entrada de dados para representá-lo. Apesar de ser esperado o mesmo comportamento para métodos deste tipo

definidos como métodos de classe e métodos de instância, a forma com que eles obtêm os dados a serem persistidos é diferente.

Quando um método de criação é definido como um método de instância, este deve obter os dados do próprio objeto no qual o método foi invocado. Essa restrição está associada ao funcionamento do aspecto de cache de dados de métodos de instância, que irá obter os dados que devem ser armazenados em cache a partir da instância a qual o método invocado pertence.

Os métodos de criação definidos como métodos de classe, por sua vez, devem retornar o objeto criado, pois o aspecto de cache responsável por métodos de classe espera obter os dados que devem ser armazenados em cache a partir deste objeto. Esta estratégia foi escolhida para dar liberdade aos desenvolvedores com relação à definição dos parâmetros que o método de criação deve receber, possibilitando que estes métodos recebam como parâmetros os valores dos campos do objeto a ser criado, o próprio objeto a ser criado, ou qualquer outro tipo de parâmetro que se julgue necessário.

Os métodos de criação – tanto os de instância como os de classe – são demarcados através da introdução da anotação `@CreateMethod`, que não declara nenhum atributo. Quando os aspectos de cache interceptam a execução de um método de criação, este método é executado antes de qualquer outro processamento. Desta forma, caso alguma exceção seja levantada na execução deste método, os novos dados não são armazenados em cache, o que economiza tempo de processamento e garante que apenas os dados persistidos corretamente no repositório serão introduzidos no sistema de cache.

Após a execução correta do método de criação, a chave primária do objeto criado é extraída, e seus dados armazenados em cache. Caso já exista um objeto em cache com a mesma chave primária do objeto criado, os dados em cache são sobrescritos pelos novos dados. Esta estratégia visa manter a transparência para desenvolvedores e usuários, assumindo que a responsabilidade por reportar um erro ao criar dados repetidos é da camada de persistência da aplicação, e não da camada de cache.

5.2.4 Tratamento dos métodos de carga

A outra forma de uma instância de uma classe gerenciada passar a figurar na árvore de cache é através da interceptação de sua carga. As regras definidas para os métodos de carga de

instância e de classe são análogas às dos métodos de criação: métodos de instância devem carregar os dados no próprio objeto, e métodos de classe devem criar um novo objeto gerenciado, carregá-lo e retorná-lo. Em ambos os casos, é esperado que os campos que compõem a chave primária do objeto a ser carregado sejam fornecidos como parâmetros do método de carga, na mesma ordem em que foram declarados no atributo `pk` da anotação `@CacheClass`.

Quando um método de carga é interceptado por um dos aspectos de cache, a classe do objeto gerenciado é obtida através do retorno do método de carga (métodos de classe) ou do tipo do objeto cujo método foi invocado (métodos de instância). Utilizando essa classe e os parâmetros fornecidos ao método – que formam a chave primária do objeto sendo carregado – os aspectos de cache verificam se os dados desta instância estão presentes na árvore de cache.

Se os dados forem encontrados neste passo, o método de carga não é executado e o objeto é carregado pelo cache. No caso de métodos de instância, os dados são carregados diretamente no objeto alvo. Na interceptação de métodos de classe, um novo objeto é criado utilizando-se o construtor padrão (sem parâmetros) da classe gerenciada, através do mecanismo de reflexão. Desta forma, quando métodos de classe forem utilizados para carregar objetos gerenciados, estes devem obrigatoriamente possuir um construtor padrão, mesmo que este só seja acessível pela própria classe (construtor privado).

Os dados são carregados nos objetos gerenciados através do uso dos mecanismos de reflexão e introspecção, acessando os campos diretamente para configurar seus valores. Desta forma os objetos gerenciados não são obrigados a possuir métodos acessores para todos os campos, mantendo a transparência para os desenvolvedores.

Por outro lado, quando a instância sendo carregada não estiver presente na árvore de cache, o método de carga é executado normalmente. Após a execução do mesmo, o objeto gerenciado é obtido e seus dados são extraídos e armazenados em cache, utilizando o mesmo processo empregado ao interceptar os métodos de criação (seção 5.2.3).

5.2.5 Tratamento dos métodos de persistência

Uma vez que os dados foram carregados na árvore de cache, torna-se necessário gerenciá-los para garantir sua integridade e consistência. Para tanto, é necessário atualizar os dados em cache sempre que os objetos gerenciados forem persistidos no repositório de dados.

Os métodos de persistência são aqueles que salvam o estado atual do objeto no repositório de dados, e são demarcados através da utilização da anotação `@StoreMethod`.

Sempre que um método desse tipo for invocado, os dados em cache correspondentes ao objeto gerenciado em questão são atualizados, garantindo a paridade dos dados em cache com os dados do repositório. Se um método de persistência for definido como um método de instância, é esperado que ele grave os dados do próprio objeto no repositório. Já se o método invocado for um método de classe, assume-se que o objeto a ser persistido seja aquele passado como primeiro parâmetro deste método. Em ambos os casos, é esperado que o objeto a ser persistido seja anotado como `@CacheClass`.

Ao interceptar a invocação destes métodos, os aspectos de cache primeiramente permitem a execução normal dos mesmos. Caso sua execução transcorra normalmente – sem nenhuma exceção sendo levantada – assume-se que os dados foram armazenados corretamente no repositório, e que o cache deve ser atualizado. Para tanto, os aspectos de cache extraem a chave primária e os dados do objeto gerenciado e os armazenam na árvore de cache. Se ocorrer alguma exceção durante a execução do método de persistência ou durante o armazenamento dos dados em cache, o nó contendo os dados desta instância do componente é removido da árvore de cache.

5.2.6 Tratamento dos métodos de remoção

O último passo para garantir a integridade e consistência dos dados em cache é excluir os objetos gerenciados da árvore de cache sempre que os métodos de remoção dos mesmos forem executados. Para demarcar esse tipo de método, usa-se a anotação `@RemoveMethod`, que não declara nenhum atributo. As regras para definição dos métodos de exclusão são as mesmas dos métodos de persistência: métodos de instância devem remover o próprio objeto do repositório, enquanto métodos de classe devem remover o objeto gerenciado passado como primeiro parâmetro do método.

O funcionamento dos aspectos de cache na interceptação de métodos desse tipo também é análogo ao funcionamento dos mesmos para métodos de persistência (seção 5.2.5). Primeiramente, o método de remoção é executado normalmente. Somente se este método for executado com sucesso os dados são excluídos da árvore de cache. Para tanto, a chave

primária do objeto de negócio é extraída e o nó da árvore de cache representado por essa chave é excluído, eliminando seus dados da camada de cache.

5.3 Gerência de relacionamentos de componentes de persistencia

Conforme especificado na seção 4.2.2, o módulo do Crystalline responsável pela gerência de relacionamentos entre componentes de persistência também será responsável por carregar estes relacionamentos sob demanda, somente quando necessário (*lazy loading*). Uma forma de implementar esta técnica é a utilização de *proxies* dinâmicos no lugar dos objetos relacionados. Quando algum método destes *proxies* é invocado pela primeira vez, o objeto que ele representa é carregado e a invocação dos métodos e leitura de campos realizadas no *proxy* são repassadas para este objeto (BAUER, KING, 2004). A desvantagem dessa abordagem está na complexidade de sua implementação.

Uma forma mais simples de implementar *lazy loading* no *framework* de cache é a utilização de AOP. Ao invés de utilizarmos *proxies* para carga dos objetos sob demanda, nós utilizamos a interceptação da leitura de campos para realizar esta carga. Ao carregar um objeto gerenciado do cache, os seus relacionamentos não são carregados imediatamente. Ao invés disso, no momento em que os campos que representam este relacionamento são acessados pela primeira vez, os objetos gerenciados são carregados pelo aspecto de carga de relacionamentos.

Um caso especial de inter-relacionamentos de objetos gerenciados ocorre quando um objeto contém uma coleção de referências a outros objetos gerenciados, como é o caso de usuários e seus pedidos (figura 4.6). A principio esse problema era resolvido simplesmente armazenando uma lista com as chaves primárias dos objetos gerenciados referenciados. O real problema surgiu no momento da carga desta lista, pois muitas vezes não era possível determinar qual a coleção correta a ser utilizada para armazenar os dados no objeto referenciado a partir de sua definição de classe, devido ao uso de interfaces.

Uma opção para solucionar esse problema é a definição de classes *default* para cada tipo de coleção (e.g. lista encadeada para listas, tabela *hash* para conjuntos, e assim por diante). Embora funcional, esta abordagem torna-se invasiva ao possibilitar a substituição da implementação da coleção escolhida pelos desenvolvedores por outro. O impacto dessa

substituição seria sentido ao realizar buscas dentro das coleções, que poderiam passar a custar mais com a utilização do *framework* cache.

A opção encontrada para resolver esse problema foi a utilização de uma coleção do mesmo tipo da presente no objeto gerenciado para armazenar as chaves primárias dos objetos relacionados. Desta forma, no momento da carga, a classe da coleção presente na árvore de cache é utilizada para determinar o tipo da coleção que será utilizada no objeto gerenciado sendo carregado. A utilização dessa estratégia no lugar da definição de tipos de coleção *default* permite um maior nível de transparência e segurança para os desenvolvedores, que podem contar com a garantia de que as coleções definidas por eles serão utilizadas ao longo de toda a execução das aplicações.

Como pode ser observado, três tipos distintos de campos podem estar presentes em objetos gerenciados: campos atômicos e referências univaloradas e multivaloradas a outros objetos gerenciados. Uma vez que cada tipo de campo deve ser tratado de forma distinta pelos aspectos de cache, foram criadas três anotações distintas para demarcar os campos que serão gerenciados pela camada de cache. São elas:

- **`@CacheField`**: demarca os campos atômicos que devem ser gerenciados pela camada de cache. O *framework* de cache armazena estes valores na árvore de cache diretamente.

- **`@RelationshipField`**: demarca os campos que armazenam uma referência a um objeto gerenciado. O *framework* de cache armazena apenas a chave primária deste objeto na árvore de cache.

- **`@RelationshipMultipleField`**: demarca as coleções que armazenam referências a objetos gerenciados. O *framework* de cache armazena apenas as chaves primárias destes objetos na árvore de cache, dentro de uma coleção do mesmo tipo da coleção referenciada no objeto gerenciado. Define o atributo `value`, que armazena o nome totalmente qualificado da classe dos objetos gerenciados armazenados dentro desta coleção.

Um quarto tipo de campo que pode existir dentro de um objeto gerenciado são campos transientes. Campos desse tipo não são gerenciados pela camada de cache, não sendo

armazenados na árvore de cache nem restaurados na carga de um objeto gerenciado. É importante que os desenvolvedores tenham em mente que esses campos não serão inicializados durante a carga de um objeto gerenciado, e que esses objetos devem estar preparados para trabalhar com referências nulas dentro destes campos. Campos transientes são definidos pela ausência das anotações definidas na lista acima.

5.4 Cache de métodos de negócio

Para ser possível aplicar cache nos resultados de métodos de negócio, é necessário descobrir o que influencia a execução desses métodos e pode alterar seus valores. O primeiro fator que podemos ressaltar são seus parâmetros. Um parâmetro de um método serve para afetar o resultado de sua execução, ao permitir que o mesmo trecho de código seja executado com valores diferentes como entrada. Se um parâmetro de um método não afeta o resultado de um método, é porque ele não está sendo utilizado ao longo da execução do mesmo.

É importante ressaltar que existem métodos nos quais um ou mais parâmetros não influenciam o resultado retornado, mas esses métodos não são considerados métodos de leitura, segundo a definição de Pfeifer e Jakschitsch (2003). Como um sistema de cache de resultados de métodos só deve ser utilizado para métodos de leitura, métodos com esse tipo de parâmetro não nos preocupam.

Outro fator que influencia o resultado de um método é o estado interno de seu objeto. Se um método de leitura utiliza os valores armazenados em campos do objeto durante o seu processamento, estes valores provavelmente irão influenciar no resultado retornado pelo mesmo, e dizemos que este tipo de método depende diretamente do estado interno do objeto.

O último fator que influencia o resultado de um método são os demais métodos chamados ao longo da sua execução. Esses métodos secundários, por sua vez, podem depender apenas de seus parâmetros, do estado interno de seus objetos, ou ainda de outros métodos. Dizemos que esse tipo de método depende indiretamente do estado interno do objeto. Esses dois tipos de dependência do estado interno de um objeto não são excludentes, existindo métodos que dependem direta e indiretamente do estado interno de seus objetos.

Analisando os fatores apresentados acima, podemos dividir os métodos de negócio em duas categorias: aqueles cujos resultados dependam apenas dos parâmetros fornecidos e aqueles cujos resultados dependam dos parâmetros fornecidos e – direta e/ou indiretamente –

do estado interno do objeto. Os métodos da primeira categoria são os mais simples de se tratar, pois não é necessário obter nenhuma informação adicional do próprio objeto. Este tipo de método será chamado de *stateless* ao longo deste trabalho.

5.4.1 Tratamento de métodos de negócio *stateless*

Quando o aspecto de cache de métodos de negócio intercepta a execução de um método do tipo *stateless*, ele utiliza a classe, o método e os valores passados como parâmetros para definir se o seu resultado já se encontra armazenado na árvore de cache, conforme mostra a figura 5.2. Ao procurar pelo resultado da invocação de um método, o aspecto de cache verifica se existe um nó na árvore de cache representando a classe do objeto gerenciado em questão, e se abaixo deste nó existe um outro nó representando o método sendo invocado. Abaixo desse último nó são armazenados os resultados da invocação dos métodos, dentro de nós que representam a lista dos valores passados como parâmetros na invocação, na mesma ordem em que estes são declarados no método.

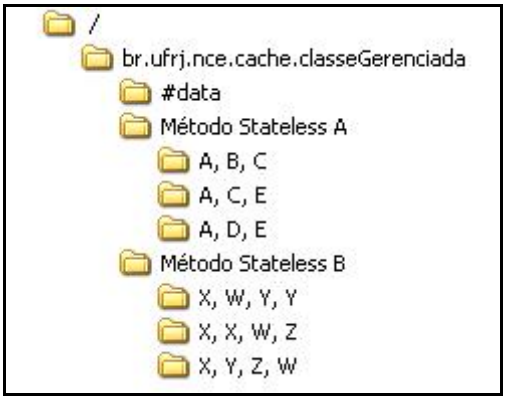

Figura 5.2 – Estrutura da árvore de cache para métodos de negócio *stateless*

Se não existir valor armazenado na árvore de cache para a invocação do método, o aspecto de cache permite que a execução do método transcorra normalmente e, após seu retorno, armazena seus resultados dentro da árvore de cache. Nas próximas invocações deste método, com os mesmos parâmetros, sua execução é impedida pelo aspecto de cache, que recupera seus resultados do cache e os retorna. Como os resultados deste tipo de método não dependem do estado interno do objeto, seus resultados podem ser utilizados de forma independente do objeto no qual o método foi invocado.

5.4.2 Tratamento de métodos de negócio *stateful*

Aos métodos de negócio pertencentes à outra categoria – os métodos cujos resultados dependem do estado interno do objeto – denominamos *stateful*. Os resultados de métodos desse tipo podem depender de valores dos campos do componente em questão, invocação de outros de seus métodos e de outros componentes gerenciados (tanto de seus campos como de seus métodos).

O tipo mais simples de métodos *stateful* são aqueles que dependem apenas de campos atômicos do próprio objeto. Nesse caso, basta adicionar o valor desses campos à lista de parâmetros no nó que armazena o resultado do método. Quando o aspecto de cache de métodos de negócio intercepta um método desse tipo, os valores atuais dos campos são extraídos do objeto gerenciado cujo método está sendo invocado, e unidos aos valores dos parâmetros utilizados nesta invocação. Esta lista completa é então utilizada para verificar se este método já foi invocado nestas condições. Em caso positivo, o valor armazenado em cache é retornado diretamente. Caso contrário, a execução do método *stateful* é realizada normalmente e seu resultado é armazenado na árvore de cache.

Um método *stateful* com dependência mais complexa é aquele que depende de campos de outro objeto gerenciado, que por sua vez deve ser um campo do objeto gerenciado que declara o método. A resolução deste tipo de dependência é idêntica à de métodos *stateful* que dependem apenas de campos atômicos. Os valores dos campos dos outros objetos gerenciados são extraídos normalmente e adicionados às demais dependências, para verificação da existência do valor em cache.

O tipo de dependência mais complexa é aquela dos métodos com dependência indireta de seu estado interno, ou seja, aqueles que dependem dos resultados da execução de outros métodos. A primeira vista, nos parecia que para resolver esse tipo de dependência seria necessário determinar quais parâmetros e campos do objeto são passados como parâmetros para cada um dos métodos secundários que compõem a dependência indireta. Entretanto, ao longo do desenvolvimento, essa premissa se mostrou falsa, uma vez que os parâmetros passados para estes métodos secundários só podem ser os parâmetros passados para o primeiro método, valores de campos do objeto, resultados do processamento desses dois tipos ou os próprios resultados de métodos secundários. Desta forma, os parâmetros passados para

os métodos secundários dependem diretamente apenas dos parâmetros passados para o método principal e dos valores dos campos do objeto.

Assim sendo, podemos transformar as dependências indiretas de um método em dependências diretas, através de um processo de herança de dependências. Esse processo consiste simplesmente em eliminar as dependências a outros métodos, substituindo-as pela própria definição da dependência de outros métodos. Essa premissa parte do principio que se um método depende do resultado de outro, na verdade ele depende dos fatores que afetam o resultado deste método secundário. Um exemplo da aplicação dessa premissa seria o seguinte: se um método M1 depende do campo C1 e do método M2, e o método M2 depende apenas do campo C2, é válido dizer que M1 depende apenas dos campos C1 e C2, além dos parâmetros recebidos.

Desta forma, quando o aspecto de cache de métodos de negócio intercepta a execução de um método com dependência indireta, ele aplica o mecanismo acima para transformar as dependências indiretas em dependências diretas. De posse das dependências diretas do método, o aspecto de cache extrai os valores dos campos dos quais o resultado do método depende e verifica se o mesmo está armazenado na árvore de cache. Da mesma forma que nos casos anteriores, se o resultado estiver em cache, o mesmo é retornado sem a execução do método de negócio. Caso contrário, o método de negócio é executado normalmente e seu resultado é armazenado na árvore de cache.

Da mesma forma que ocorre com os métodos de ciclo de vida de um componente de persistência, é necessário demarcar os métodos de negócio que serão gerenciados pela camada de cache. Os métodos *stateless* devem ser marcados com a anotação `@StatelessBusinessMethod`, que não define nenhum atributo. Os métodos do tipo *stateful*, por sua vez, devem receber a anotação `@StatefulBusinessMethod`, e sua dependência deve ser especificada em seu atributo `dependencies`. Por default, o aspecto de cache de métodos de negócio irá assumir que métodos *stateful* dependem de todos os campos do objeto, e de nenhum de seus métodos.

Para alterar esse comportamento, as dependências devem ser listadas no atributo `dependencies`, sob a forma de uma lista de campos e métodos separados por vírgulas. Quando um método é definido como dependência, sua assinatura deve ser descrita de forma

completa, com os nomes totalmente qualificados de seus parâmetros, conforme apresentado na figura 5.3.

A figura 5.3 exemplifica a declaração de dependência de um método. Nesse exemplo, o método em questão depende do campo C1, dos métodos M1 que recebem como parâmetro, respectivamente, somente um número inteiro e uma string e um número inteiro, e do método M2 – sem parâmetros – do campo C2.

```
@StatefulBusinessMethod(
dependencies="C1, M1(int), M1(java.lang.String, int), C2.M2()")
```

Figura 5.3 – Exemplo de dependência de um método

Vale ressaltar que é desnecessário declarar dependências a métodos do tipo *stateless* uma vez que, ao transformar esta dependência indireta em uma dependência direta, ela simplesmente desaparece. Da mesma forma, se for declarada uma dependência a um método não gerenciado (sem nenhuma das duas anotações de métodos de negócio), o mesmo é tratado como um método *stateless*, e esta dependência é eliminada durante a transformação de dependências indiretas em dependências diretas.

5.5 Cache de métodos de busca

O último ponto para completar a camada de cache formada pelo Crystalline é a possibilidade de se realizar cache de resultados de métodos de busca. Métodos deste tipo podem ser encarados como uma forma especial de métodos de negócio *stateless*, uma vez que ambos dependem apenas dos parâmetros passados. Sua diferença está no tratamento dos resultados obtidos quando os mesmos devem ser armazenados e extraídos da árvore de cache.

O retorno de um método de busca será sempre um objeto gerenciado ou uma coleção destes. Quando um método desse tipo é executado pela primeira vez com determinado conjunto de parâmetros, sua execução transcorre normalmente. Após a sua execução, o aspecto de cache recupera os objetos gerenciados retornados, armazenando-os em cache e guardando a coleção de chaves primárias destes objetos para ser utilizada nas próximas invocações deste método.

Nas invocações posteriores do método com o mesmo conjunto de parâmetros, a coleção de chaves primárias é obtida da árvore de cache e os objetos correspondentes são carregados a

partir dos dados em cache. O problema com o tipo de coleção a ser retornado foi solucionado utilizando a mesma técnica utilizada para coleções de referências a objetos gerenciados (seção 5.3): as chaves primárias são armazenadas em uma coleção do mesmo tipo da coleção retornada pelo método de busca. Assim, ao obter os resultados do método de busca da árvore de cache, uma coleção do mesmo tipo é instanciada, os objetos gerenciados representados pelas chaves primárias obtidas são carregados e armazenados nesta coleção, que é então retornada.

Quando o método – ao invés de retornar uma coleção de objetos gerenciados – retorna apenas um objeto gerenciado, um procedimento parecido com o utilizado para coleções é adotado. Nesse caso, ao invés de armazenarmos a chave primária do objeto gerenciado em uma coleção, a armazenamos diretamente no nó da árvore de cache correspondente. Ao recuperar esse resultado da árvore de cache em execuções futuras deste mesmo método, com os mesmos parâmetros, o objeto é carregado e retornado diretamente.

O problema encontrado no cache de métodos de busca é a necessidade de atualizar os resultados obtidos conforme as informações contidas no repositório de dados são alteradas. Dentre as possíveis estratégias para contornar esse problema, podemos citar a invalidação por tempo de vida e a invalidação por operações nos objetos.

Na primeira opção, ao obter um resultado de um método de busca, o *framework* de cache associaria a ele um prazo de validade, após o qual esse resultado deve ser considerado inválido e descartado da árvore de cache. O problema com essa estratégia está em seu caráter otimista, no qual deve ser assumida a possibilidade de se trabalhar com um conjunto de dados que não mais representa a realidade contida no repositório de dados. Como os métodos de ciclo de vida continuariam mantendo os dados dos objetos armazenados em cache consistentes com o repositório de dados, os métodos de busca passariam a apresentar um comportamento errático ao longo do funcionamento da aplicação.

Um exemplo desse tipo de comportamento errático pode ser observado no seguinte cenário: um usuário requisita um conjunto de livros com valor inferior a vinte e cinco reais. Um outro usuário, antes do prazo de validade deste resultado expirar, altera o valor de alguns destes livros para um valor superior a vinte e cinco reais. Se fossem requisitados novamente os livros com preço inferior a vinte e cinco reais antes do prazo de validade dos resultados

expirarem, seriam retornados para esta consulta livros com preço maior do que o limite superior fornecido, configurando um comportamento errado da aplicação.

A segunda opção – invalidação por operações nos objetos – não sofre deste tipo de problema. Nesta estratégia, sempre que o conjunto de objetos consultado por um método de busca fosse alterado (objetos criados, alterados ou excluídos), os resultados deste método seriam todos invalidados. Desta forma, no cenário descrito acima, sempre que um livro fosse criado, alterado ou excluído, todos os resultados dos métodos de busca de livros seriam excluídos da árvore de cache. Esta estratégia, devido à sua visão generalista, nos pareceu ineficiente para cenários nos quais ocorra um número razoável de atualizações de objetos, pois, mesmo que sejam cenários do tipo *read-intensive*, a quantidade de resultados descartados seria extremamente alta, diminuindo consideravelmente o nível de leituras a partir da camada de cache.

Para reduzirmos a quantidade de invalidações desnecessárias, seria necessário um meio de descobrir quais os métodos afetados pela informação criada, excluída ou alterada. Desta forma, torna-se possível invalidar apenas os resultados afetados por uma atualização pontual no conjunto de informações. A partir desta inferência, podemos observar que, se há uma forma de se descobrir quais resultados de métodos de busca são afetados por determinada alteração nos dados da aplicação, deve ser possível descobrir qual o tipo de alteração deve ser realizado nos resultados, isto é, se o objeto gerenciado que disparou o processo de invalidação deveria ser incluído ou excluído de cada um dos resultados afetados. O problema neste ponto passa a ser como fazer para realizar esta verificação.

Nas especificações de métodos de busca de componentes de persistência EJB (SUN, 2003), os métodos de busca gerados automaticamente pelo servidor de aplicação são especificados através de uma linguagem de consulta chamada EJB-QL (*Enterprise Java Beans – Query Language* ou Linguagem de consulta para EJBs). Nas especificações destes métodos, utiliza-se a EJB-QL para definir as restrições da consulta a ser realizada para obter-se o resultado esperado para o método, tal qual em uma consulta SQL em um banco de dados relacional.

Com o uso de uma linguagem deste tipo, torna-se possível obter as restrições de cada método e, assim, verificar se um objeto deve ser removido ou incluído em seus resultados, para cada conjunto de parâmetros armazenados. Aplicando esta técnica no cenário de

atualização dos preços de livros descrita acima, seria possível atualizar os resultados de forma que, quando o preço de um livro passar a ser de quarenta reais, este livro seja removido de todos os resultados cujo parâmetro que indica o limite superior do preço seja inferior a este valor. Da mesma forma, se neste cenário um livro tem seu valor reduzido de quarenta para vinte reais, este livro deve ser incluído em todos os resultados cujo limite superior para o preço seja superior a este valor.

A utilização de uma linguagem como EJB-QL para fornecer este tipo de inteligência ao *framework* de cache trás como vantagem a possibilidade de se manter os dados de resultados de consultas sempre atualizados na camada de cache, mantendo a mesma totalmente transparente para os desenvolvedores. A única desvantagem observada nesta abordagem está na imposição de uma nova linguagem aos desenvolvedores, o que pode acarretar em um alongamento da curva de aprendizado necessária para utilização do *framework* de cache.

Por limitações de tempo para desenvolvimento desta solução, tornou-se necessário implementar uma outra solução menos complexa, para ser possível testar a utilização de cache de métodos de busca. A solução encontrada foi a criação de objetos auxiliares pelos desenvolvedores, com métodos capazes de descobrir se um objeto gerenciado pertence ou não aos resultados de um método de busca de acordo com os parâmetros fornecidos. Nesta solução, é necessário criar uma classe que tenha métodos homônimos aos métodos de busca, que recebam como parâmetro o objeto gerenciado, seguido da mesma lista de parâmetros do método de busca, se existentes. Além disso, estes métodos devem retornar um valor booleano, que indica se o objeto gerenciado pertence ou não ao resultado sendo consultado.

```
public class Livro{
    private double preco;

    public double getPreco(){
          return preco;
    }

    List<Book> procuraLivrosPorLimiteSuperiorDePreco(
                                              double limiteSuperior){
       // Consulta os livros com preços inferiores ao valor
       // do parâmetro limiteSuperior
    }
```

```
}
```

Figura 5.4 – Código-fonte para classe Livro, em Java

```
public class AuxiliarMetodosProcuraLivro{
    public boolean procuraLivrosPorLimiteSuperiorDePreco (
                                            Livro livro,
                                            double limiteSuperior){
          return livro.getPreco() < limiteSuperior;
    }
}
```

Figura 5.5 – Código fonte da classe auxiliar para métodos de busca de livros, em Java

As figuras 5.4 e 5.5 exemplificam as classes Livro – com seu método de busca `procuraLivrosPorLimiteSuperiorDePreco()` – e a classe AuxiliarMetodosProcuraLivro, utilizada para validar os métodos de busca definidos em Livro. Podemos observar na figura 5.5 que, conforme especificado acima, o método auxiliar tem o mesmo nome do método de busca, retorna um valor booleano e recebe como parâmetros um objeto da classe Livro, seguido dos parâmetros do método de procura original: um valor de ponto flutuante representando o limite superior de preço.

```
public class Livro{
    private double preco;

    public double getPreco(){
          return preco;
    }
    @FinderMethod(finderHelperClass =
                                    "AuxiliarMetodosProcuraLivro")
    List<Book> procuraLivrosPorLimiteSuperiorDePreco(
                                            double limiteSuperior){
       // Consulta os livros com preços inferiores ao valor
       // do parâmetro limiteSuperior
    }
}
```

Figura 5.6 – Código-fonte anotado para classe Livro, em Java

Para possibilitar a gerência dos resultados dos métodos de busca, é necessário demarcar os métodos deste tipo e, de alguma forma, fornecer a classe do objeto auxiliar definido para cada método ao aspecto responsável. Para esta finalidade foi criada a anotação `@FinderMethod`, que define `finderHelperClass` como seu único atributo, que indica o nome totalmente qualificado do objeto auxiliar, responsável pela validação dos objeto junto aos resultados deste método de busca. Desta forma, a classe Livro poderia ser reescrita para suportar a gerência dos métodos de busca conforme a.figura 5.6.

De posse da classe deste objeto, sempre que um objeto gerenciado for criado, excluído ou alterado, todos os resultados de métodos de busca relativos ao tipo deste objeto devem ser processados e atualizados de acordo. No *framework* de cache criado, um método de busca é relativo a uma classe de objetos se o mesmo é declarado nesta classe.

Quando um objeto gerenciado é excluído, ele deve ser removido de todos os resultados em que estiver presente. Como sua remoção dos resultados em cache independe de seus dados atuais, simplesmente removemos a chave primária deste objeto de todos os resultados de métodos de busca relativos à sua classe. Para implantar esta restrição, adicionamos o aspecto responsável pela manutenção dos resultados de métodos de busca à cadeia de interceptadores dos métodos de exclusão.

Em alterações de objetos gerenciados, o procedimento é semelhante ao descrito acima. Sempre que um método de persistência é executado, o aspecto de cache de métodos de busca percorre os resultados dos métodos de busca do objeto gerenciado e este é avaliado pelo objeto auxiliar correspondente. Sempre que o objeto gerenciado for avaliado como pertencente a um método de busca, sua chave primária é adicionada aos resultados do mesmo. Caso contrário, sua chave é removida.

Finalmente, também adicionamos o aspecto de cache de resultados de métodos de busca à cadeia de interceptadores dos métodos de criação. Ao criar um objeto gerenciado de determinado tipo, deve-se percorrer os resultados dos métodos de busca deste tipo e, sempre que o objeto auxiliar determinar que o objeto gerenciado pertença ao resultado em questão, a chave primária do mesmo deve ser adicionada a este resultado.

O problema – não solucionado – encontrado nesta abordagem é relativo à atualização das consultas que retornem resultados ordenados (métodos que retornam listas de objetos): ao

adicionar um objeto em uma lista com os resultados o mesmo será sempre inserido no fim, sem se importar com a ordenação. Uma possível solução é a definição de uma nova anotação que indique o(s) campo(s) dos objetos retornados que deve(m) ser utilizado(s) na ordenação dos resultados.

6 Análise de desempenho

No capítulo 5 descrevemos como o Crystalline foi desenvolvido, qual o seu escopo, como ele atua em conjunto com as aplicações que o utilizam e quais as suas deficiências. Uma vez construído o *framework* de cache, torna-se necessário testar seu funcionamento e seu desempenho, para verificar se a arquitetura concebida para o Crystalline pode ser aplicada em situações reais e oferecer benefícios para os sistemas que a utilizarem.

Para avaliar o desempenho do Crystalline foi criado um conjunto de testes unitários para cada aspecto do *framework* e duas aplicações de teste para sua validação funcional: uma para validar a camada de cache de persistência, e outra para validar a camada de cache de negócios. O conjunto de testes unitários foi criado segundo os preceitos de desenvolvimento baseado em testes (BECK, 2002), e serviu para nortear o desenvolvimento do *framework* de cache, possibilitando validar cada uma de suas classes à medida que estas foram desenvolvidas. As aplicações de teste, por sua vez, são necessárias para possibilitar a avaliação comparativa do funcionamento do *framework* de cache em cenários completos.

Assim como o *framework* Crystalline, as aplicações de teste também foram desenvolvidas utilizando a linguagem de programação Java, em sua versão 5.0. Nas próximas seções iremos descrever os cenários de teste utilizados para validar os módulos de cache de métodos de persistência e de métodos de negócio do Crystalline, e a execução destes testes.

6.1 Módulo de cache de métodos de persistência

Para testar o desempenho dos módulos de cache de métodos de persistência e de cache de métodos de pesquisa, desenvolvemos uma aplicação de testes que simula as atividades de um *site* de uma loja de livros. O desenvolvimento de tal aplicação baseou-se no modelo fornecido pelo *Transaction Processing Performance Council* (TPC) para ambientes *web*, o TPC-W (TPC, 2002), que estabelece regras para um *benchmark* de um sistema de bancos de dados transacional para o ambiente *web*.

Os testes descritos nesta seção foram executados utilizando um servidor de banco de dados SQL Server 2000 (Intel Pentium 2.8GHz Dual Core, com 2GB de memória RAM), enquanto a aplicação cliente foi executada em um computador Intel Pentium 2.8GHz com 1GB de memória RAM, utilizando a máquina virtual Java da Sun, versão 1.5.0_08.

Sobre esta aplicação foram aplicados diversos cenários de teste, para comparar seu desempenho utilizando o Crystalline e outros sistemas de cache existentes no mercado atualmente. A seguir, iremos descrever os cenários e a aplicação de teste em maiores detalhes, para em seguida apresentarmos e analisarmos os resultados obtidos.

6.1.1 Cenários de teste

Para uma correta avaliação da camada de cache que atua sobre os métodos de persistência, é necessário comparar os resultados obtidos na execução da aplicação de testes em diversos cenários. São eles:

- Utilização do Crystalline de forma transparente, sem o suporte aos métodos de busca. Esse cenário permite coletar os dados para posterior avaliação do impacto da ativação dos métodos de busca.

- Utilização do Crystalline de forma transparente, com suporte aos métodos de busca. Esse cenário permite avaliar o impacto da ativação do suporte a este tipo de método no *framework* de cache.

- Utilização de outro sistema de cache, possibilitando avaliar as diferenças de desempenho decorrentes da implementação da camada de cache. Este teste permite avaliar o desempenho do Crystalline comparativamente com outros sistemas de cache pré-existentes.

- Sem utilizar nenhum sistema de cache, possibilitando avaliar as vantagens e desvantagens que os sistemas de cache trouxeram aos quatro cenários anteriores.

Cada um destes cenários será avaliado com duas situações distintas: busca totalmente randômica e busca por valores pré-determinados. No primeiro caso, os valores passados para as buscas de livros serão gerados automaticamente de forma totalmente aleatória, seguindo a mesma especificação para geração dos dados iniciais, fornecida pelo TPC-W (TPC, 2002). Estes testes permitirão avaliar o comportamento e o desempenho da aplicação do *framework* de cache em cenários com baixo índice de reaproveitamento dos dados em cache.

A execução dos cenários utilizando buscas por valores pré-determinados permitirá a avaliação do *framework* de cache em cenários com alto índice de reaproveitamento dos dados em cache, e uma real avaliação dos benefícios trazidos por este trabalho.

6.1.2 Aplicação de testes

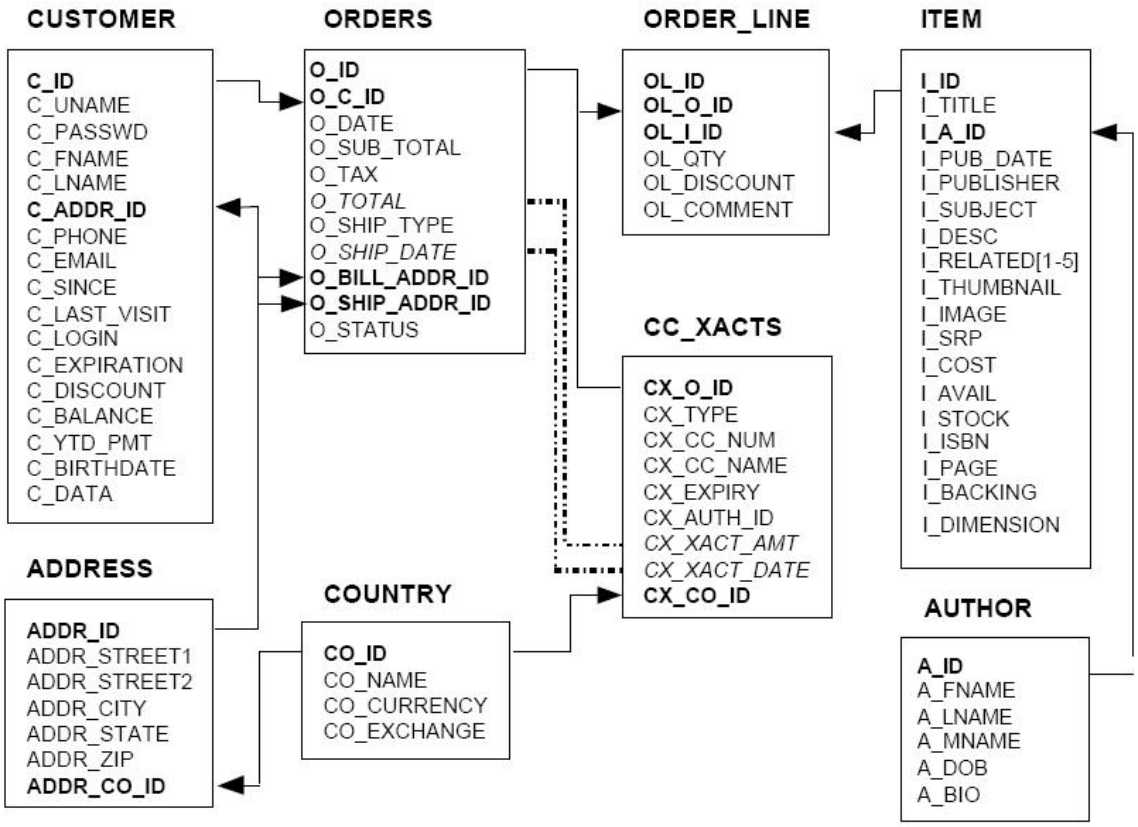

Figura 6.1 – Modelo de dados TPC-W

A figura 6.1 exibe o modelo de dados definido no TPC-W para o sistema de vendas *on-line*. As tabelas deste modelo representam os clientes do sistema (CUSTOMER), seus endereços (ADDRESS), seus pedidos (ORDER e ORDER_LINE), os detalhes da transação de cartão de crédito (CC_XACTS), os livros disponíveis no site (ITEM) e seus autores (AUTHOR), e os países atendidos pelo sistema (COUNTRY). Uma descrição mais detalhada destas tabelas pode ser encontrada na especificação do TPC-W (TPC, 2002).

Utilizamos o modelo de dados da figura 6.1 para construir uma aplicação que simula um ambiente de comércio eletrônico e configuramos a base de dados inicial segundo a especificação do TPC-W para cem mil itens e quarenta clientes simultâneos. Esses parâmetros resultaram no preenchimento das tabelas da base de dados de acordo com o demonstrado na tabela 6.1.

Tabela 6.1 – Preenchimento da base de dados

Tabela	Nº de linhas
COUNTRY	92
AUTHOR	25.000
ITEM	100.000
CUSTOMERS	115.200
ORDERS	103.680
ORDER_LINE	311.983
CC_XACTS	103.680
ADDRESS	230.400

A especificação do TPC-W define que os testes devem ser realizados em um ambiente *web* completo, através de clientes que emulem os navegadores dos usuários e realizem requisições HTTP. A partir deste ponto nossa aplicação de testes passa a divergir desta especificação, uma vez que a mesma foi desenvolvida para simular a interação dos clientes com a camada de negócios da aplicação sem a definição de uma interface para clientes. Como nosso teste não é um *benchmark* comparativo entre bancos de dados, concluímos não ser necessário seguir a especificação TPC-W à risca.

Ao invés de criarmos uma interface de usuário e utilizá-la para executar os testes de forma automatizada, optamos por criar dois clientes que acessam a camada de negócios diretamente. Esses dois clientes simulam a interação de dois personagens distintos com o sistema: clientes do site e administradores. Os clientes do site têm as seguintes opções: procurar livros por título, autor e editora, obter os dados detalhados de um livro escolhido, colocar livros em seu carrinho de compra e efetuar a compra. Os administradores são os responsáveis pela alimentação dos dados do site, dispondo das seguintes opções: alterar os dados de um livro, e cadastrar novos autores e novos livros no site.

Esses dois atores distinguem-se também pelos cenários gerados por suas interações com o sistema de compras: *read-intensive* ou *write-intensive*. Clientes acessam o sistema em cenários onde ocorre leitura intensiva de dados e pouca ou nenhuma escrita, uma vez que o único dado cadastrado pela interação do cliente com a aplicação é o registro de sua compra, se efetuada. Os administradores, por sua vez, acessam o site para causar alterações no conjunto de dados cadastrados no repositório, seja pela alteração de dados pré-existentes, seja pelo cadastramento de novos dados.

A utilização desses dois atores na aplicação de testes permitiu-nos avaliar o *framework* de cache em cenários com comportamento do tipo *read-intensive*, mas com alterações esporádicas nos dados, que obriguem o *framework* a atualizar as informações armazenadas em cache. Optamos por avaliar o *framework* de cache em um cenário deste tipo por julgarmos que é neste tipo de cenário que um sistema de cache mais pode contribuir para o aumento do desempenho geral da aplicação.

A execução conjunta desses dois atores possibilita testar de forma mais completa o cache de objetos persistentes e dos métodos de busca, pois as alterações e inclusões realizadas pela simulação de um administrador permitirão avaliar os custos da gerência dos dados e dos métodos de busca através da comparação com os resultados das outras execuções.

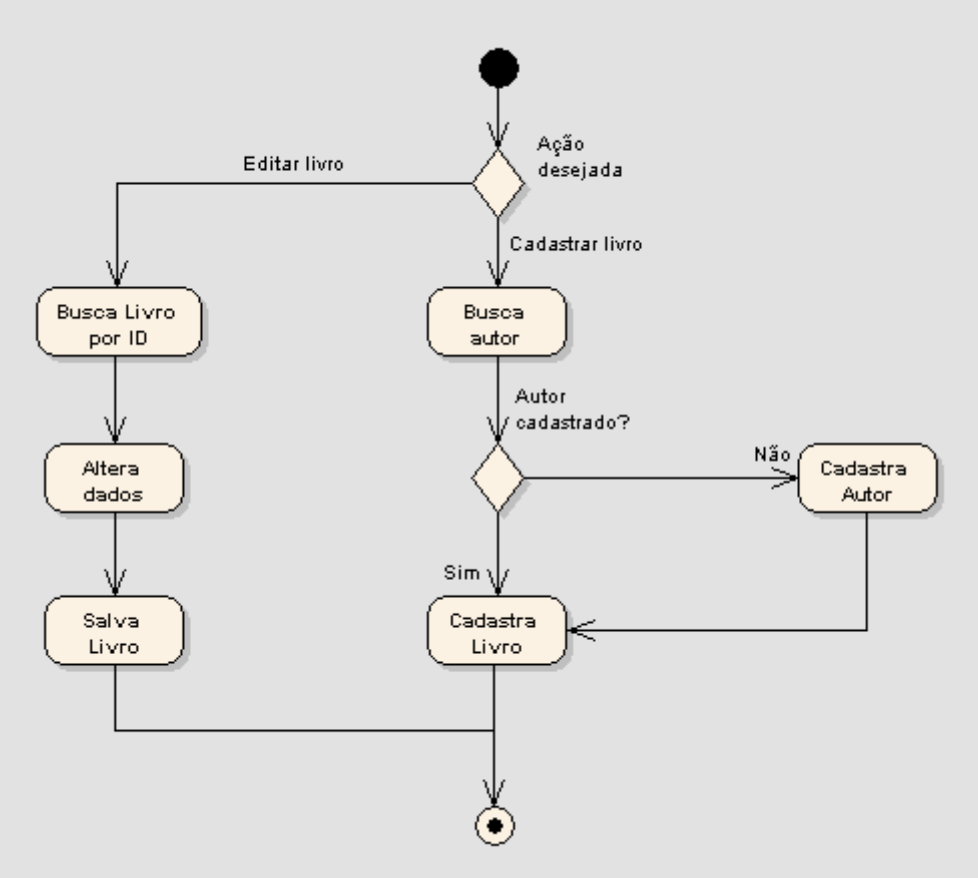

Figura 6.2 – Diagrama de atividades para o administrador do site

O programa que emula o comportamento de um administrador do sistema irá executar a cada dez segundos, quando o mesmo irá efetuar uma atualização, seguindo os passos delineados no diagrama de atividades exposto na figura 6.2. Essa atualização pode ocorrer na forma de uma alteração dos dados de um livro (20% de probabilidade) ou na forma de um

cadastramento de um novo livro e seu autor (80% de probabilidade), caso o último não esteja cadastrado no sistema.

O cadastramento de um novo livro será feito da seguinte forma:

- O administrador busca um autor pelo nome;
- Se o autor não estiver cadastrado no sistema, cria um novo autor;
- Cria um novo livro associado ao autor acima.

No caso da atualização dos dados de um livro, a execução do cenário se dá da seguinte forma:

- Busca um livro por seu identificador numérico (id);
- Altera alguns de seus dados – título, autor ou editora – randomicamente;
- Salva as alterações no repositório de dados.

O programa que emula o comportamento dos clientes do site no processo de seleção e compra de livros, por sua vez, irá executar uma vez por segundo. Nesse cenário serão executadas buscas de livros por título, autor e editora, com probabilidade de 1/3 para cada tipo. O comportamento desse ator é delineados no diagrama de atividades exposto na figura 6.3, explicado na seqüência abaixo:

- Se a lista de livros retornada pela busca executada anteriormente retornar algum livro, o cliente percorre esta lista;
- Para cada livro da lista, há uma chance de 20% de o cliente se interessar pelo mesmo e requisitar a obtenção de mais detalhes sobre o mesmo;
- Para cada livro detalhado, há uma probabilidade de 25% de o cliente colocá-lo em seu carrinho de compras;
- Independente de o livro detalhado ter sido colocado no carrinho de compras ou não, há uma chance de 15% de o cliente realizar novas consultas por livros, voltando ao cenário;

- Se nenhuma outra consulta for efetuada e houver livros no carrinho de compras, há uma chance de 75% de o cliente efetuar a compra, e 25% de descartá-la por completo.

- Caso o cliente opte por realizar a compra dos livros, os registros referentes a este pedido são registrados na base de dados.

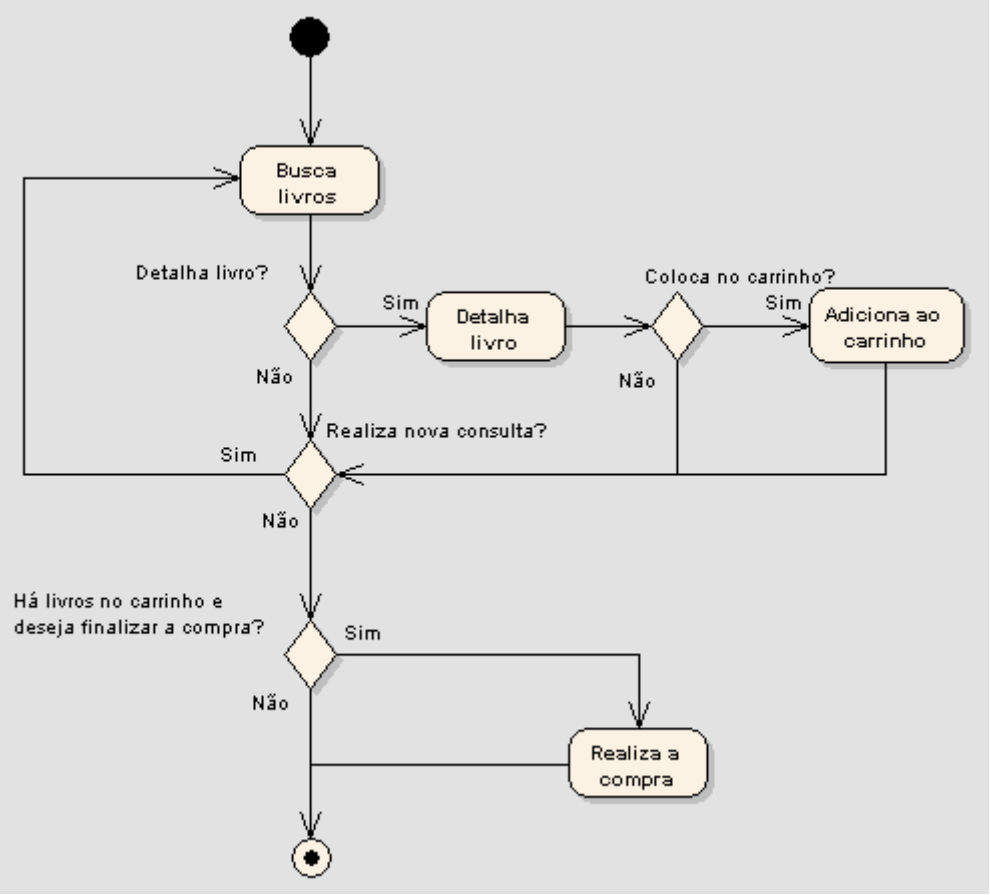

Figura 6.3 – Diagrama de atividades para o cliente do site

6.1.3 Arquitetura e implementação da aplicação de testes

A camada de persistência da aplicação foi criada utilizando o *framework* Hibernate (BAUER, KING, 2003) e objetos DAO para isolar a lógica de persistência do restante da aplicação. Esses objetos DAO são utilizados como fábricas de objetos, e seus métodos de criação, persistência, carga e remoção de dados foram definidos como métodos estáticos, assim como seus métodos de busca.

O Hibernate foi escolhido por simplificar o desenvolvimento da camada de persistência da aplicação e, ao mesmo tempo, possibilitar a execução dos outros cenários de teste (seção

6.1.1) sem a necessidade de se modificar o código-fonte da aplicação de testes. O Hibernate conta com um mecanismo de cache de objetos do repositório de dados, permitindo a comparação do desempenho do *framework* de cache com um mecanismo de cache estável e largamente utilizado. O Hibernate permite que o cache seja habilitado ou desabilitado de forma declarativa, sendo necessário apenas realizar pequenas alterações em seu arquivo de configuração.

A execução dos cenários que utilizam o *framework* de forma não-transparente foi configurada através de uma propriedade passada para a aplicação em tempo de execução, que define se o sistema de cache não-transparente deve ser ativado ou não. Apesar de esta verificação oferecer uma pequena perda de desempenho devido à necessidade da verificação da existência de tal propriedade, sua interferência nos resultados dos testes foi ignorada, uma vez que todos os cenários de teste utilizarão o mesmo código-fonte.

6.1.4 Configuração da aplicação e dos cenários de teste

Uma análise da aplicação de teste mostra que nem todas as classes de persistência se beneficiariam da utilização de um sistema de cache. Objetos do tipo Order e OrderItem são criados pelo cliente que simula um usuário, mas jamais são lidos novamente do repositório de dados em nossa aplicação. O uso de um sistema de cache nesta classe serviria apenas para adicionar um *overhead* desnecessário à aplicação, sem oferecer nenhum benefício adicional.

Objetos dos tipos Country, Customer e Address são bons candidatos para a utilização de um sistema de cache, visto que em nenhum momento instâncias dessas classes sofrem qualquer tipo de atualização. Objetos utilizados somente para leitura são perfeitos candidatos para a utilização de cache, pois conseguem obter todos os benefícios oferecidos por tais sistemas sem introduzir nenhum *overhead* adicional na manutenção dos dados em cache.

Finalmente, objetos dos tipos Author e Item têm comportamento do tipo *read-intensive* em nossa aplicação de testes, mas devido ao cliente administrador, sofrem algumas alterações. Itens são criados e modificados, e autores são apenas criados em nossa aplicação. É do nosso entendimento que objetos com comportamento deste tipo podem se beneficiar da aplicação de um sistema de cache, desde que os dados sejam gerenciados corretamente.

Na aplicação de testes foram utilizados quatro métodos de busca, sendo três para itens e um para autores. Para itens, configuramos os métodos de busca de livros por título, nome do

autor e nome da editora para utilizar o cache de métodos de busca e, para autores, o método de busca por último nome do autor.

Para o cenário que utiliza o sistema de cache do próprio Hibernate utilizamos o provedor Ehcache (2005), com as configurações para o cache de objetos descritas acima. Escolhemos este sistema por ser considerado um sistema de cache eficiente e por ser bastante utilizado junto ao Hibernate, permitindo uma comparação com o desempenho oferecido por nosso *framework* de cache. Optamos por não utilizar o TreeCache como provedor de cache no Hibernate devido ao fato de este sistema, no Hibernate, requerer o uso de um gerente de transação, o que complicaria desnecessariamente nossa aplicação – que passaria a depender de um servidor de aplicação – e adicionaria mais um *overhead* ao seu processamento.

É importante salientar que o Hibernate trabalha com dois sistemas de cache em sua execução, que são definidos como cache primário e secundário. O cache primário é utilizado para armazenar todos os dados recuperados do repositório de dados dentro de uma mesma sessão, e seus dados expiram quando esta sessão é fechada. Este sistema de cache não pode ser desativado e, portanto, estará ativo durante a execução de todos os cenários de teste.

O sistema de cache secundário é um sistema de cache auxiliar, que permite que os dados sejam armazenados e compartilhados por diversas sessões diferentes. O Hibernate dispõe de alguns provedores de cache já implementados em sua distribuição padrão, dentre os quais o provedor do Ehcache. Ao longo da execução dos cenários de teste, será este o sistema de cache que será ativado e desativado.

6.1.5 Resultados dos testes

Os resultados dos testes do *framework* de cache aplicado à camada de persistência da aplicação foram colhidos ao longo de sessenta minutos de execução, para cada cenário. De cada conjunto de resultados foram excluídos os dados obtidos nos primeiros e nos últimos cinco minutos. Os dados dos primeiros cinco minutos foram desconsiderados para colher apenas resultados obtidos através da real utilização dos sistemas de cache testados, quando os mesmos já se encontram carregados com as informações obtidas do repositório de dados. Os cinco últimos minutos, por sua vez, foram excluídos para garantir que nenhum dos testes fosse influenciado pela fase de término dos testes, quando os *threads* concorrentes recebem o sinal de finalização.

As tabelas 6.2 e 6.3 apresentam os resultados obtidos na execução dos cinco cenários de teste de objetos de persistência para buscas completamente randômicas, que também podem ser observadas nos gráficos exibidos nas figuras (6.4 e 6.5).

Tabela 6.2 – Quantidade de execuções dos cenários por minuto, buscas totalmente randômicas

Cenário	Média	Variância	Desvio padrão
Sem Cache	50,62	4,771	2,18
Cache Hibernate	49,88	2,720	1,65
***Framework* transparente**	47,24	2,145	1,46
***Framework* com métodos de busca**	19,78	10,134	3,18

Tabela 6.3 – Tempo (em segundos) de execução de um cenário com buscas totalmente randômicas

Cenário	Média	Variância	Desvio padrão
Sem Cache	0,25	0,003	0,05
Cache Hibernate	0,23	0,002	0,04
***Framework* transparente**	0,22	0,001	0,04
***Framework* com métodos de busca**	1,26	0,311	0,56

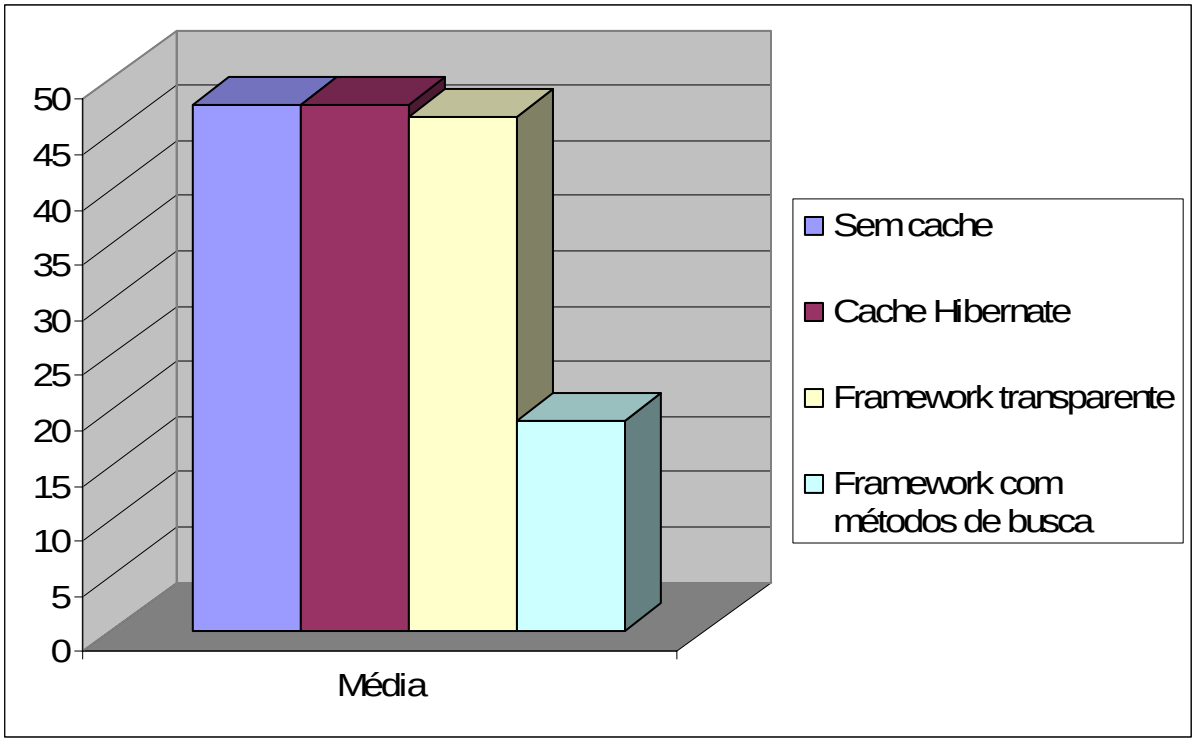

Figura 6.4 – Quantidade de execuções dos cenários por minuto, buscas totalmente randômicas

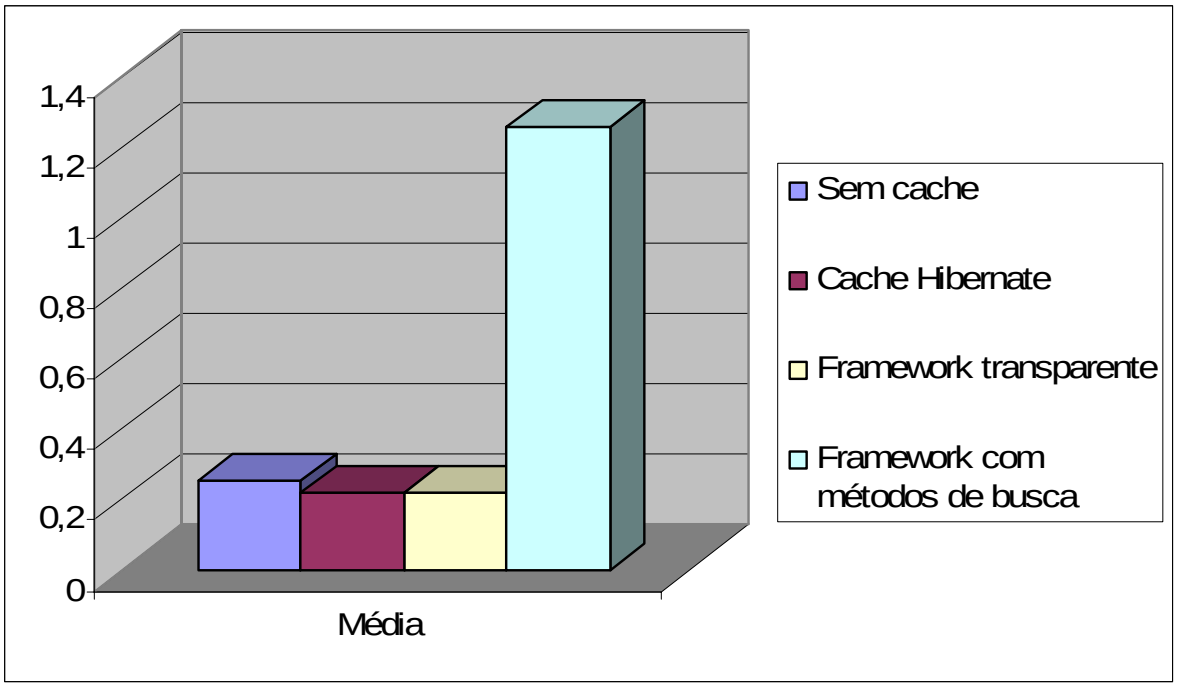

Figura 6.5 – Tempo (em segundos) de execução de um cenário com buscas totalmente randômicas

As tabelas 6.4 e 6.5 apresentam os resultados obtidos na execução dos cinco cenários de teste de objetos de persistência para buscas utilizando um conjunto limitado de valores para os seus parâmetros, que também podem ser observados nos gráficos exibidos nas figuras (6.6 e 6.7).

Tabela 6.4 – Quantidade de execuções dos cenários por minuto, buscas limitadas

Cenário	Média	Variância	Desvio padrão
Cache Hibernate	109,89	140,878	11,87
Sem Cache	102,58	301,035	17,35
***Framework* transparente**	119,00	241,556	15,54
***Framework* com métodos de busca**	19,95	79,942	8,94

Tabela 6.5 – Tempo (em segundos) de execução de um cenário com buscas limitadas

Cenário	Média	Variância	Desvio padrão
Sem Cache	2,76	0,098	0,313
Cache Hibernate	3,01	0,283	0,532
***Framework* transparente**	2,57	0,138	0,371
***Framework* com métodos de busca**	16,50	54,710	7,397

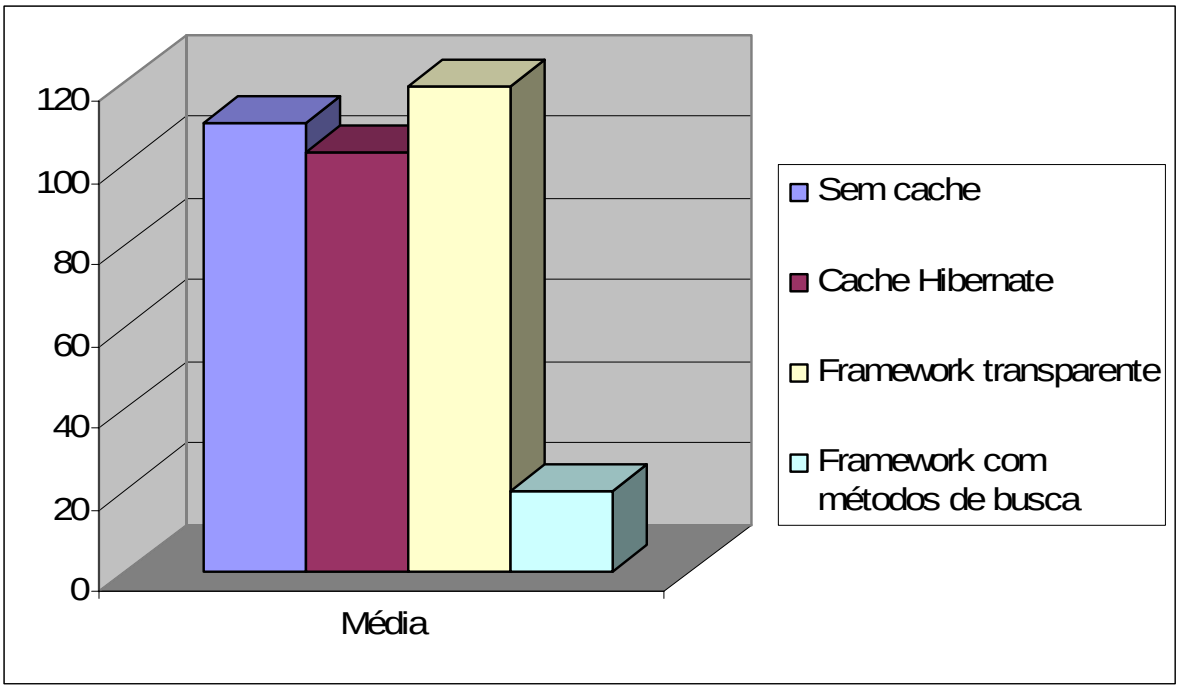

Figura 6.6 – Quantidade de execuções dos cenários por minuto, buscas limitadas

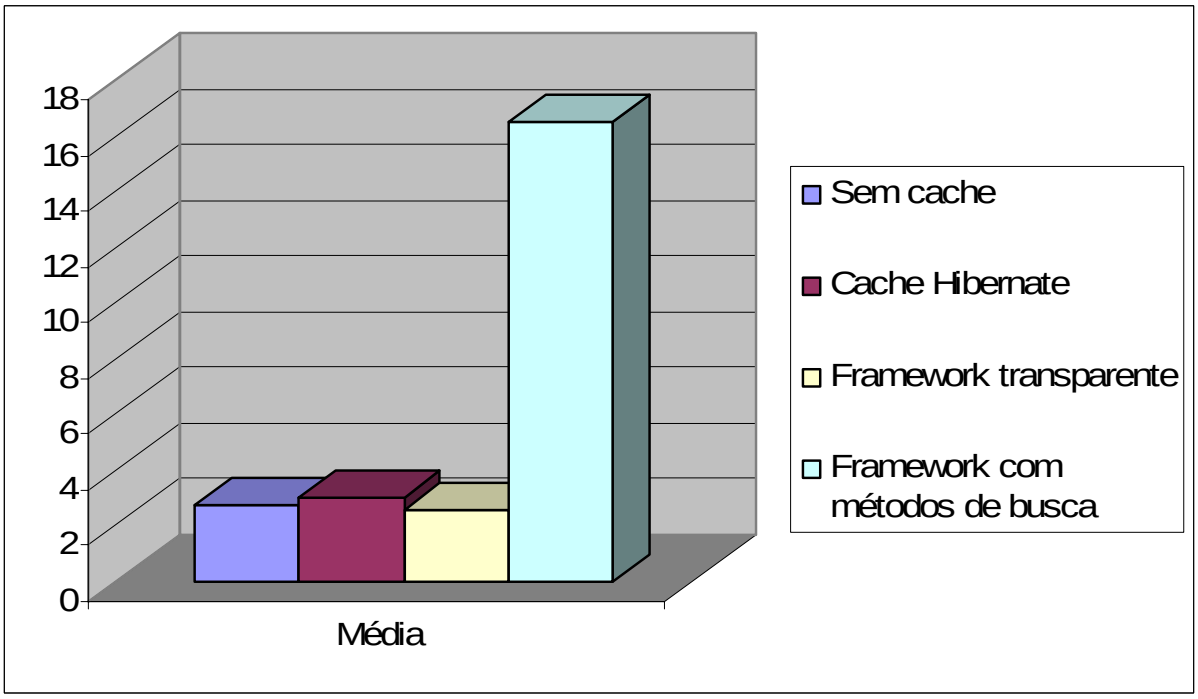

Figura 6.7 – Tempo (em segundos) de execução de um cenário com buscas limitadas

6.1.6 Interpretação dos resultados

Analisando as tabelas 6.2 e 6.3 pode-se observar que, exceto no caso da utilização de métodos de busca, os resultados não apresentam grande variação entre si nas buscas

totalmente randômicas. A análise desses dados demonstra que o Crystalline tem um bom desempenho, equiparando-se ao desempenho do provedor Ehcache do Hibernate no tempo de execução de cada interação.

Observando-se o desvio padrão do tempo de execução desses dois cenários (tabelas 6.2 e 6.3), pode-se notar que os resultados obtidos pela utilização do *framework* de cache encontram-se num intervalo menor do que o do Ehcache, o que pode indicar um tempo de resposta menor do sistema de cache, o que serve de indicador para validar a eficiência da estrutura utilizada para o armazenamento dos dados em cache. Os resultados dos testes com parâmetros de busca limitados seguem a mesma tendência, corroborando com esta interpretação.

A comparação dos resultados dos cenários que utilizam o Crystalline de forma transparente com os cenários que não utilizam qualquer sistema de cache demonstra que, nestes cenários, de acordo com as tabelas 6.3 e 6.5, o tempo de execução de cada interação com os clientes é reduzido em, respectivamente, 12% e 6,9%. Estes dados nos permitem afirmar que o *framework* de cache funciona corretamente, uma vez que ele reduz o tempo de resposta do sistema para seus clientes.

A análise dos resultados dos cenários que utilizaram o suporte aos métodos de busca mostrou-se decepcionante. Tanto no cenário com buscas totalmente randômicas como no cenário com parâmetros de busca limitados os tempos de resposta do sistema (tabelas 6.3 e 6.5) ficaram muito acima dos outros quatro cenários. Esses resultados eram esperados para o cenário de buscas totalmente randômicas, no qual a taxa de acerto do cache (*cache-hit*) deveria ser muito baixa devido à variedade dos parâmetros de busca fornecidos.

O fato de este tipo de cache ter afetado negativamente os tempos de resposta no cenário com parâmetros de busca limitados nos mostrou que o *overhead* introduzido na manutenção dos resultados dos métodos de busca foi demasiadamente grande, ofuscando os eventuais benefícios trazidos por esta técnica. Durante a execução dos dois cenários que utilizavam o suporte a métodos de busca, observou-se que o nível de uso da CPU foi significativamente maior do que na execução de todos os outros, o que indica que a carga de processamento na manutenção dos resultados destes métodos é significativa, degradando o desempenho da aplicação ao invés de melhorá-lo.

6.2 Módulo de cache de métodos de negócio

A última etapa dos testes de nosso *framework* envolveu o cache de métodos de negócio. Como a especificação do TPC-W (TPC, 2002) não fornece nenhum cenário que se beneficie da utilização de cache de métodos de negócio, essa parte do *framework* de cache não pôde ser testada junto aos cenários descritos anteriormente.

Um cenário no qual julgamos que o cache de métodos de negócio pode ser aplicado com um bom índice de sucesso é o de execução de métodos recursivos, que tendem a ser executados exaustivamente. Um bom exemplo de métodos desse tipo é o cálculo de números da série de Fibonacci, na qual o valor de um número na n-ésima posição é igual ao valor da soma dos seus dois antecessores na série. O cálculo dos elementos dessa série é recursivo por sua própria natureza, sendo um forte candidato a se beneficiar do uso de algum sistema de cache.

A aplicação de teste criada executa um método de negócio repetitivamente, e a mesma foi observada em dois cenários distintos: utilizando o *framework* de cache e não o utilizando. Não julgamos necessária a realização de testes utilizando o framework de cache de forma não-transparente, pois os dados coletados na execução dos testes para métodos de persistência já nos são suficientes para avaliar o impacto causado pela implementação da transparência.

```
public long fibonacci(int n){
    if(n == 1)
          return 0;
    if(n == 2)
          return 1;
    return fibonacci(n-1) + fibonacci(n-2);
}
```

Figura 6.8 – Implementação do método de negócio independente do estado interno do objeto

Para os testes, foi escolhido um método de cálculo do n-ésimo número da série de Fibonacci, com n variando uniformemente entre 1 e 50. Como métricas, foram utilizados o número de execuções por minuto e o tempo médio de cada execução deste método. O método de cálculo de Fibonacci foi implementado conforme exibido na figura 6.8. Estes testes foram executados em um servidor AMD Athlon 64 3200+, 1GB RAM, utilizando a máquina virtual Java da Sun, versão 1.5.0_05.

6.2.1 Resultados dos testes

Os resultados dos métodos de negócio, por sua vez, foram colhidos ao longo da execução dos testes por vinte minutos. Pelos mesmos motivos apresentados para os testes do cache aplicado à camada de persistência, os cinco primeiros e os cinco últimos minutos foram desconsiderados.

Tabela 6.6 – Execuções/minuto de métodos de negócio

Cenário	Média	Variância	Desvio padrão
Sem Cache	16,90	168,9888	12,9995
***Framework* transparente**	3,08E6	8,58E12	2,93E6

As tabelas 6.6 e 6.7 apresentam os resultados obtidos na execução dos testes de cache de métodos de negócio para o cálculo de elementos randômicos da seqüência de Fibonacci.

Tabela 6.7 – Tempo (em segundos) de execução de método de negócio

Cenário	Média	Variância	Desvio padrão
Sem Cache	6,6519	20,4534	4,5225
***Framework* transparente**	2,90E-5	5,15E-10	2,27E-5

6.2.2 Interpretação dos resultados

Os resultados da execução dos testes com o suporte a cache demonstram o benefício que o uso dessa técnica pode trazer às aplicações que a utilizarem, desde que os métodos sejam bem escolhidos. O método que calcula os números de Fibonacci foi implementado de forma recursiva propositalmente, para poder demonstrar o quanto pode ser ganho com a aplicação de um sistema de cache neste tipo de método.

A eliminação das execuções recursivas do método foi a grande responsável pelo aumento do desempenho da aplicação e a conseqüente redução do tempo de resposta do mesmo. Esse resultado mostra que o Crystalline aplica a técnica de memoização (MICHIE,1968 *apud.* MAYFIELD, HALL & FININ 1994) de forma dinâmica. Caso o método de cálculo dos números de Fibonacci tivesse sido implementado de forma não recursiva os ganhos obtidos pela utilização de um cache provavelmente não teriam sido tão acentuados assim.

7 Conclusões

Neste trabalho foi apresentada e especificação e a implementação de um serviço de cache totalmente transparente para aplicações sem tornar seu código-fonte dependente de uma plataforma de cache específica. Analisando a revisão bibliográfica, não encontramos outros trabalhos voltados para este tipo de desenvolvimento, mas encontramos algumas propostas de cache não convencionais para métodos de negócio das aplicações. A estas idéias somamos o funcionamento tradicional de um sistema de cache de componentes de persistência e o conceito de métodos de busca, resultando em uma camada de cache completa para ser utilizada em sistemas de informação.

A proposta inovadora deste trabalho não é o desenvolvimento de um sistema de cache em si, mas sim a forma como o mesmo pode ser acoplado a sistemas de informação. A utilização de programação orientada a aspectos para atingir este objetivo, apesar de intuitiva, não basta para adicionar o serviço de cache de forma transparente a uma gama variada de aplicações. Para isso, foi necessário definirmos as situações nas quais o Crystalline poderia ser utilizado, e as regras que o *framework* utiliza para obter os dados necessários para seu funcionamento correto nestas situações. Embora o conjunto de situações/regras definidas neste trabalho não cubra todas as aplicações existentes, ele é amplo o suficiente para permitir que um grande número de sistemas utilize o Crystalline sem maiores dificuldades.

Utilizamos uma aplicação de testes baseada no modelo definido pela especificação do TPC-W e outra para cálculo de números de Fibonacci para testar nossa proposta, e os resultados nos permitiram validar as estratégias adotadas para cache de métodos de persistência e para cache de métodos de negócio.

Através da análise destes resultados, mostramos que o serviço de cache aplicado à camada de persistência funciona corretamente e com um desempenho satisfatório, mesmo quando comparado com um sistema de cache largamente utilizado atualmente. O *overhead* adicionado pela camada AOP não teve impactos significativos, tornando possível a utilização transparente de um sistema de cache sem perdas de desempenho perceptíveis.

Os testes também confirmaram que o cache de métodos de negócio pode ser aplicado com sucesso em métodos com alta demanda de processamento repetitivo. A aplicação deste tipo de cache em métodos recursivos, que acessem componentes remotos ou que acessem

recursos computacionais mais lentos pode resultar em aumentos significativos no desempenho geral da aplicação.

Os resultados obtidos também demonstraram que utilizar o cache de métodos de busca com uma estratégia de manutenção de seus resultados é inviável devido ao intenso processamento exigido para manter estes resultados atualizados constantemente. É possível que, em consultas cujos resultados não sejam alterados freqüentemente, a utilização de uma estratégia de invalidação permita a utilização deste tipo de cache.

7.1 Contribuições

A maior contribuição deste trabalho é demonstrar ser possível criar serviços de infra-estrutura totalmente transparentes para as aplicações que os utilizem. Poucos sistemas de *middleware* permitem um nível de acoplamento tão baixo com as aplicações e ao mesmo tempo mantém um comportamento genérico, que possibilite sua utilização em uma gama variada de aplicações.

Outra contribuição que consideramos importante é a demonstração de que a manutenção dos resultados de um método de busca não é viável. Devido ao alto custo computacional necessário para verificar todos os resultados de métodos de busca armazenados em cache sempre que um objeto é criado, alterado ou removido do repositório, o desempenho da aplicação torna-se pior do que seria se não fosse utilizado nenhum sistema de cache.

Finalmente, nos alinhamos com Pohl, Schill, Göbel, Jakschitsch e Pfeifer (2003) na defesa do uso de cache para métodos de negócio para melhoria do desempenho geral das aplicações. Assim como esses pesquisadores, acreditamos e provamos que, quando bem aplicado, o cache de métodos de negócio pode trazer grandes benefícios para as aplicações que o utilizem.

7.2 Trabalhos futuros

Este trabalho provê diversos pontos de extensão, dentre os quais podemos destacar:

1. Implementação de uma estratégia de invalidação em substituição à estratégia de manutenção dos resultados dos métodos de busca. Acreditamos que algumas consultas específicas em cenários que não sofram atualizações freqüentes podem se beneficiar com a utilização do cache de métodos deste tipo. A estratégia de

invalidação é mais simples e menos custosa do que a estratégia de manutenção dos resultados e, em cenários *read-intensive*, pode ocasionar melhorias no desempenho das aplicações;

2. Especificação de novas regras para definições dos métodos de ciclo de vida dos componentes de persistência, permitindo que uma gama maior de aplicações possa utilizar este *framework* de cache de forma transparente. Atualmente, apenas aplicações que definam fábricas estáticas de objetos ou que definam objetos auto-persistidos podem utilizar o *framework* desenvolvido neste trabalho, limitando sua utilização.

3. Criação de novos serviços genéricos complementares que possam ser acoplados a este *framework*, tais como gerência de transações, coleta de métricas, controle de segurança e até mesmo estendê-lo para um sistema de persistência completo.

4. Adição de novos cenários de testes aos conjuntos apresentados nas seções 6.1 e 6.2, permitindo validar a arquitetura proposta neste trabalho em outras situações além das apresentadas nas seções supracitadas.

5. Re-execução dos testes utilizando um intervalo de tempo maior, de forma a aumentar a quantidade de *cache hits*, permitindo uma melhor avaliação do Crystalline.

Referências

ALMAER, D., **Give your DB a break**, TheServerSide.com, 2003. Disponível em: http://www.theserverside.com/resources/article.jsp?l=DB_Break, acessado em: 15/12/2003.

ALUR, D., CRUPI, J., MALKS, D., **Core J2EE Patterns: Best Practices and Design Strategies, Second Edition**, 2ª edição, Prentice Hall PTR, 2003.

ANTON, J. *et al.*, **Web Caching for Database Applications with Oracle Web Cache**, Special Interest Group on Management of Data, 2002. Disponível em: http://otn.oracle.com/products/ias/web_cache/pdf/oracle-webcache-SIGMOD2002.pdf, acessado em: 15/12/2003.

BAUER, C.; KING, G., **Hibernate in Action**, 1ª edição, Manning Publications, 2004.

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

BORSATO, A., **Enriquecendo a Transparência e Mecanismo de Busca em Sistemas Prevalentes**, 2004, Dissertação (Mestrado em Informática), Núcleo de Computação Eletrônica, Instituto de Matemática, Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2004.

BORTVEDT, J., **Functional Specification for Object Caching Service for Java (OCS4J) 2.0**, 2001. Disponível em: http://jcp.org/aboutJava/communityprocess/jsr/cacheFS.pdf, acessado em: 15/12/2003.

EHCACHE, 2005. Disponível em http://ehcache.sourceforge.net/, acessado em: 13/06/2006.

FRANKLIN, M., CAREY, M., LIVNY, M., **Transactional Client Server Cache Consistency: Alternatives and Performance**, ACM Transactions on Database Systems, vol. 22, número 3, p.p. 315-363, 1997.

GAMMA, E. *et al.*, **Design patterns – elements of reusable object-oriented software**, 1ª edição, Addison Wesley Professional, 1995.

GRAY, J., *et. al.*, **The dangers of replication and a solution**. In Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, p.p. 173-182, 1996.

JIMENEZ-PERIZ, R, *et. al.*, **How to select a replication protocol according to scalability, availability and communication overhead**. In Proceedings of IEEE Symposium on Reliable Distributed Systems, p.p. 24-33, 2001.

JBOSSCACHE, 2003. Disponível em: http://www.jboss.org/developers/projects/jboss/cache, acessado em: 17/02/2004.

JBOSSAOP, 2003. Disponível em: http://www.jboss.org/developers/projects/jboss/aop, acessado em: 17/02/2004.

KICZALES, G. *et al*., **Aspect Oriented Programming**. In Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finlândia, 1997.

KUROSE, J., ROSS, K, **Computer Networking – A top-down approach featuring the Internet**, 2ª edição, Addison Wesley, 2001.

LADDAD, R., **I want my AOP!, Part 1**, Portal JavaWorld, 2002. Disponível em: http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect.html, acessado em: 02/08/2005.

LADDAD, R., **AspectJ in Action: Practical Aspect-Oriented Programming**, 1ª edição, Manning Publications, 2003.

MAYFIELD, J., HALL, M., FININ, T., **Using Automatic Memoization as a Software Engineering tool in Real-World AI Systems**. In Proceedings of the 11th Conference on Artificial Intelligence for Applications, p.p. 87-93, 1995.

MCINNIS, K., **Component-based Development**, Castek Software Factory, 2000. Disponível em: http://www.cbd-hq.com/PDFs/cbdhq_000901km_cbd_con_tech_method.pdf, acessado em: 24/06/2004.

PFEIFER, D.; JAKSCHITSCH, H., **Method Based Caching in Multi-Tiered Server Applications**, Technical Report, 2003. Disponível em: http://www.ipd.uka.de/~pfeifer/publications/techreport_2003_11.pdf, acessado em: 17/12/2003.

POHL, C.; GÖBEL, S., **Integrating Orthogonal Middleware Functionality in Components Using Interceptors**, In Kommunikation in Verteilten Systemen (KiVS 2003), 2003.

POHL, C.; SCHILL, A., **Client-side Component Caching: A Flexible Mechanism for Optimized Component Attribute Caching**, In 4th International Conference on Distributed Applications and Interoperable Systems, 2003.

SUN Microsystems, **JNDI - Java Naming and Directory Interface** website, http://java.sun.com/products/jndi.

___, **Enterprise Java Beans Specification 2.1**, 2003a.

___, **Java TM Data Objects Specification**, 2003b. Disponível em: http://jcp.org/aboutJava/communityprocess/final/jsr012/index2.html, acessado em: 16/06/2005.

___, **Java Programming Language**, 2004. Disponível em: http://java.sun.com/j2se/1.5.0/docs/guide/language/, acessado em: 03/05/2006.

TANGOSOL, **Coherence**, 2001. Disponível em: http://www.tangosol.com/, acessado em: 15/12/2003.

TPC, **TPC Benchmark™ W (Web Commerce)**, 2002. Disponível em: http://www.tpc.org/tpcw/, acessado em: 12/01/2006.

XDOCLET, 2002. Disponível em: http://xdoclet.sourceforge.net/xdoclet/index.html, acessado em: 18/12/2005.