Padrões de Projeto para Estruturação de Aplicações Distribuídas Enterprise JavaBeans

Klissiomara Dias* and Paulo Borba†
Centro de Informática
Universidade Federal de Pernambuco

Resumo

Enterprise JavaBeans (EJB) auxilia o desenvolvimento de aplicações de negócio que lidam com aspectos como distribuição, persistência e transações. Aplicações dessa natureza, se desenvolvidas de forma ad hoc podem levar a sistemas cujo código mistura aspectos de negócio com aspectos não funcionais, podendo comprometer alguns fatores de qualidade, tais como reusabilidade e extensibilidade. Por este motivo, este artigo propõe dois padrões de projeto que auxiliam na estruturação de aplicações EJB, visando obter alguns benefıcios como reuso, modularidade, extensibilidade, independência de tecnologia (distribuição ou dados) e desempenho. Estes padrões auxiliam ainda na estruturação de aplicações EJB a partir de sistemas já existentes, sem EJB, minimizando o impacto das mudanças sobre as demais camadas da aplicação.

Abstract

Enterprise JavaBeans (EJB) technology provides support for the development of modern applications, taking into consideration aspects like distribution, persistence and transactions. Applications of this nature, if developed in ad hoc fashion can result in systems whose code mixes business and non-functional requirements (for example, distribution and persistence), being able to compromise some quality aspects, such as reusability and extensibility. This paper proposes two design patterns that aid in structuring EJB applications, in order to gain some benefits like reuse, modularity, extensibility, technology independence (i.e. distribution or data) and performance. In addition, they can assist in designing EJB applications from non-EJB already-existing systems, and thus softening the impact of changes on other application layers.

*Financiada pelo CNPQ. Email: kld2@cin.ufpe.br

†Parcialmente financiado pelo CNPq, vínculo 521994/96–9. Email: phmb@cin.ufpe.br

1 Introdução

O padrão arquitetural *Layer* (padrão em camadas) [5] é utilizado para a estruturação de aplicações complexas que lidam com diferentes requisitos, funcionais e não funcionais. Com a utilização desta arquitetura, é possível distribuir as classes que compõem o sistema em camadas bem definidas, de acordo com cada aspecto da aplicação (negócio, persistência, comunicação, etc.)

A divisão de um sistema em camadas permite obter aplicações modulares e reutilizáveis, uma vez que o código de diferentes aspectos (apresentação, comunicação, negócio e dados, por exemplo) não são misturados. Além disso, os elementos das diferentes camadas comunicam-se através de interfaces.

Em aplicações distribuídas, a comunicação entre objetos executando em diferentes máquinas é realizada através de mecanismos e protocolos de comunicação. Quando tais objetos manipulam aspectos de comunicação diretamente, a tendência é que suas funcionalidades sejam misturadas com as tarefas de comunicação. O mesmo acontece com aplicações que utilizam algum meio de armazenamento persistente. O desenvolvimento *ad hoc* de aplicações que utilizam alguma plataforma de persistência, para armazenamento e recuperação de seus objetos, leva a sistemas que misturam código de acesso a dados com o código de negócio da aplicação.

Enterprise JavaBeans (EJB) [14] é uma tecnologia que trata de aspectos como distribuição e persistência. Por este motivo, foi feita uma análise acerca da necessidade do uso de padrões existentes para a arquitetura EJB. Como resultado desta análise, surgiram os padrões apresentados neste artigo.

Desta forma, os padrões apresentados neste artigo são utilizados no contexto de aplicações EJB e estruturam classes e objetos que preenchem as camadas citadas acima. A apresentação dos padrões em um mesmo artigo visa facilitar sua compreensão, uma vez que estes estão relacionados:

- *Distributed Adapters Pattern with EJB* (DAP-EJB). O DAP-EJB corresponde à adaptação do padrão *Distributed Adapters Pattern* (DAP) [2], o qual foi primeiramente definido em *Progressive Development of Distributed Object-Oriented Applications* [1] e visa isolar o *middleware* da aplicação, tornando-a extensível para vários tipos de mecanismo de comunicação.

- *Persistent Data Collections with EJB* (PDC-EJB). O PDC-EJB corresponde à adaptação do padrão *Persistent Data Collections* (PDC) [11], o qual visa permitir reutilização da lógica de negócio para diferentes mecanismos de persistência.

2 DAP-EJB: Um Padrão para Distribuição com EJB

Objetivo

Fornecer uma estrutura para implementação de distribuição em um sistema com EJB, visando separação de conceitos e conseqüentemente fatores de qualidade como modularidade, extensibilidade e reusabilidade.

Contexto

O padrão DAP-EJB é utilizado no contexto de comunicação remota entre dois componentes, onde é desejável que tais componentes não estejam acoplados à tecnologia de distribuição.

Problema

Apesar de EJB prover interoperabilidade e transparência de localização para o acesso aos objetos remotos, os clientes de um *enterprise bean* ainda precisam fazer uso de interfaces e classes específicas da API de distribuição a fim de obter as referências remotas para os *beans*. Isto implica que a interface com o usuário acaba tendo código específico de EJB. O mesmo acontece com relação à camada de negócio. Como resultado, tanto a interface com o usuário quanto a camada de negócio ficam vulneráveis às modificações realizadas na camada de comunicação.

Forças

O DAP-EJB leva em consideração as seguintes forças:

- Um componente deve ser capaz de acessar serviços remotos fornecidos por outros componentes;
- Os componentes devem ser independentes do *middleware* da aplicação; isto permite, por exemplo, que o mesmo sistema possa utilizar diferentes *middleware* ao mesmo tempo ou, ainda, possa ser executado localmente.
- A modificação no código dos componentes para suportar comunicação deve ser minimizada; ou seja, a inserção do componente de distribuição em uma aplicação não distribuida deve causar pouco impacto no código já existente.
- Modificação da tecnologia de distribuição deve ser uma tarefa simples; é importante estruturar os aspectos de distribuição de forma modular, vislumbrando o fraco acoplamento destes em relação cliente e o negócio da aplicação.

Solução

Para resolver o problema apresentado, o DAP-EJB introduz o uso de um par de adaptadores [7] que são utilizados para encapsular o código relacionado à API de distribuição. O objetivo é permitir que a inserção, remoção, ou troca do *middleware* de distribuição de uma aplicação seja realizado de forma a minimizar as mudanças necessárias no código do sistema. O uso destes adaptadores isola a interface com o usuário e a camada de negócio da plataforma de distribuição do sistema, abstraindo desta forma a tecnologia utilizada para comunicação remota entre componentes.

Estrutura

O diagrama de classes da Figura 1 destaca a estrutura do padrão DAP-EJB. As classes em cinza denotam os adaptadores e seus colaboradores, os quais, basicamente, escondem

a API de distribuição da interface com o usuário e o código de negócio. As demais classes lidam com os aspectos de negócio da aplicação. Os elementos que fazem parte do padrão e seus colaboradores são explicados a seguir.

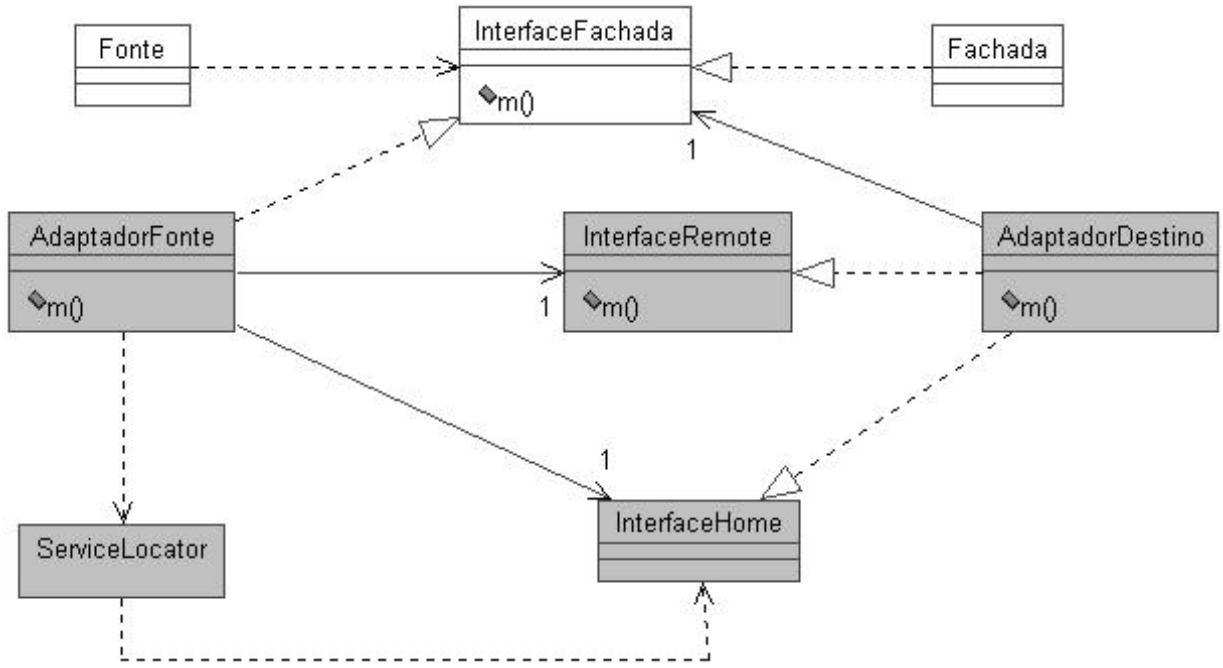

Figura 1: Estrutura do padrão DAP-EJB

- Fonte

  A classe `Fonte` representa qualquer objeto que faz o papel de cliente (GUI, por exemplo) da fachada e que está localizado em uma máquina remota em relação aos objetos do sistema.

- Fachada

  A classe `Fachada` é estruturada de acordo com o padrão *facade* [7], sendo responsável por encapsular todos os serviços oferecidos pelo sistema (no diagrama, o método `m` representa um dos possíveis serviços). Esta classe representa, na verdade, o objeto remoto a ser acessado pelo cliente.

- InterfaceFachada

  A `InterfaceFachada` é uma interface que abstrai o comportamento da fachada em um cenário distribuído. Esta classe, em conjunto com as classes `Fonte` e `Fachada` constituem uma camada independente da tecnologia de distribuição. Os demais elementos do diagrama ficam responsáveis por esse aspecto e constituem a camada de distribuição do sistema.

- AdaptadorFonte

O adaptador fonte é uma classe Java [8] "pura"e isola a classe `Fonte` do código de distribuição. Um objeto desta classe reside na mesma máquina que o objeto da classe fonte e trabalha como um *proxy* [7] para o adaptador destino. Repassa as chamadas feitas pelos clientes para o próprio adaptador destino, isolando todo o código relacionado com a plataforma de distribuição, mantendo o cliente isolado deste código (inclusive exceções de comunicação).

- AdaptadorDestino

  O adaptador destino é um *stateless session bean* [14] e, como todo componente EJB, possui duas interfaces remotas. Este componente é responsável por repassar as chamadas para o objeto fachada.

- ServiceLocator

  A classe `ServiceLocator` é uma classe auxiliar que visa abstrair a complexidade do processo de localização e criação dos *beans*, bem como melhorar o desempenho do sistema. É utilizada no padrão DAP-EJB para auxiliar o processo de criação das referências remotas ao adaptador destino.

Dinâmica

A Figura 2 apresenta o diagrama de seqüência para um cenário do DAP-EJB. Durante a inicialização, o `Fonte` cria um `AdaptadorFonte`, o qual executa o método `getHome` de `ServiceLocator` e este executa uma operação de `lookup` sobre o serviço de nomes JNDI, a fim de obter uma referência da interface *home* do `AdaptadorDestino`. Após obter a referência do *home* do adaptador destino, o `AdaptadorFonte` executa o método `create` sobre este a fim de obter uma referência à sua interface remota, no intuito de obter acesso aos serviços oferecidos pelo `AdaptadorDestino`. Ao executar uma operação `create` sobre o adaptador destino, uma única instância da `Fachada` é obtida através do método `getInstance`[1]. O `Fonte` então, invoca uma operação local `m` sobre o `AdaptadorFonte` e este invoca a operação remota `m` do adaptador destino, o qual delega esta chamada localmente para a `Fachada`.

Consequências

O DAP-EJB oferece as seguintes vantagens:

- Código modular

  O uso do padrão permite a estruturação dos aspectos de distribuição de forma modular, de modo a propiciar o fraco acoplamento destes em relação ao cliente e a camada de negócio da aplicação.

- Reutilização e extensibilidade

  Devido à estrutura modular obtida com o padrão, é possível utilizar as classes fonte e fachada mais facilmente em outras aplicações que utilizem diferentes tecnologias de distribuição. Além disso, mudanças na camada de comunicação são mais fáceis

[1] A `Fachada` é implementada como um *Singleton* [7]

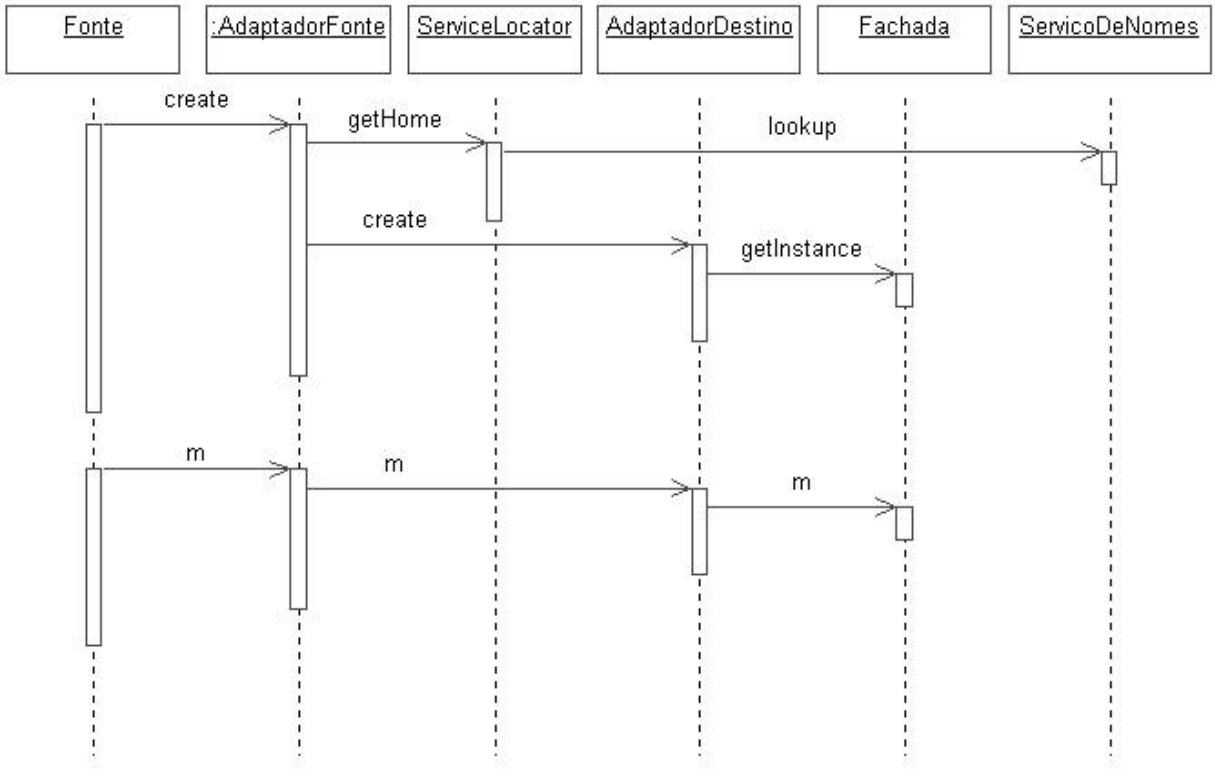

Figura 2: Diagrama de Sequência do DAP-EJB.

de realizar porque afetam somente os adaptadores fonte e destino. Um exemplo das situações citadas é um mesmo sistema ser acessado remotamente por um cliente utilizando EJB como API de distribuição ou ser acessado localmente, na máquina do cliente sem EJB. O mesmo sistema pode, ainda, ser acessado por um cliente que utiliza CORBA, por exemplo, como tecnologia de comunicação remota.

- Implementação progressiva

  O padrão suporta implementação progressiva [4]. Desenvolvedores constroem um protótipo funcionalmente completo, onde o cliente depende diretamente da fachada, e realizam os testes da funcionalidade do sistema a fim de validar seus requisitos funcionais. Depois, o componente de distribuição pode ser inserido causando pouco impacto no código já existente. Isto é possível porque o componente de distribuição implementa a mesma interface que a fachada. É importante destacar que o padrão além de permitir inserir o componente de distribuição em um sistema já existente, local, também permite adaptar um sistema distribuído em outra plataforma (por exemplo, RMI), sem que para isso seja necessário descartar as classes correspondentes ao cliente e fachada.

Por outro lado, o DAP-EJB possui as seguintes desvantagens:

- Aumento do número de classes

  Um par de adaptadores, duas interfaces remotas e um `ServiceLocator` são necessários. Todavia, essa estrutura é simples e seu código pode ser gerado de forma automática

com o auxílio de ferramentas, que são importantes para a utilização do padrão na prática. Estas ferramentas devem não somente auxiliar na geração dos adaptadores e elementos relacionados ao componente de distribuição, mas também na manutenção, quando da inserção de um método na fachada, por exemplo.

- Eficiência

  Com a introdução dos adaptadores, é necessário um maior número de invocações até que a chamada do método original seja executada no objeto remoto. Além do *overhead* causado com a introdução dos adaptadores, existe um outro fator que contribui para o *overhead* quando da invocação de métodos remotos que, neste caso, é uma característica intrínseca de EJB e que, portanto, não é uma desvantagem do padrão em si. Para se ter acesso a um objeto remoto EJB é necessário realizar duas chamadas de métodos sobre suas interfaces antes de invocar o método de negócio. Além disso, eficiência é um problema genérico da arquitetura em camadas, uma vez que um maior número de classes é introduzido no sistema, aumentando, com isso, a transferência de dados entre camadas e, por conseguinte, o número de chamadas de métodos [5].

Exemplo

O diagrama de classes UML [3] da Figura 3 ilustra a estrutura do padrão DAP-EJB através do exemplo de uma simples aplicação bancária. As classes em cinza correspondem aos adaptadores e elementos utilizados para esconder a API de distribuição do código de negócio e do cliente. As demais classes denotam os aspectos de negócio da aplicação.

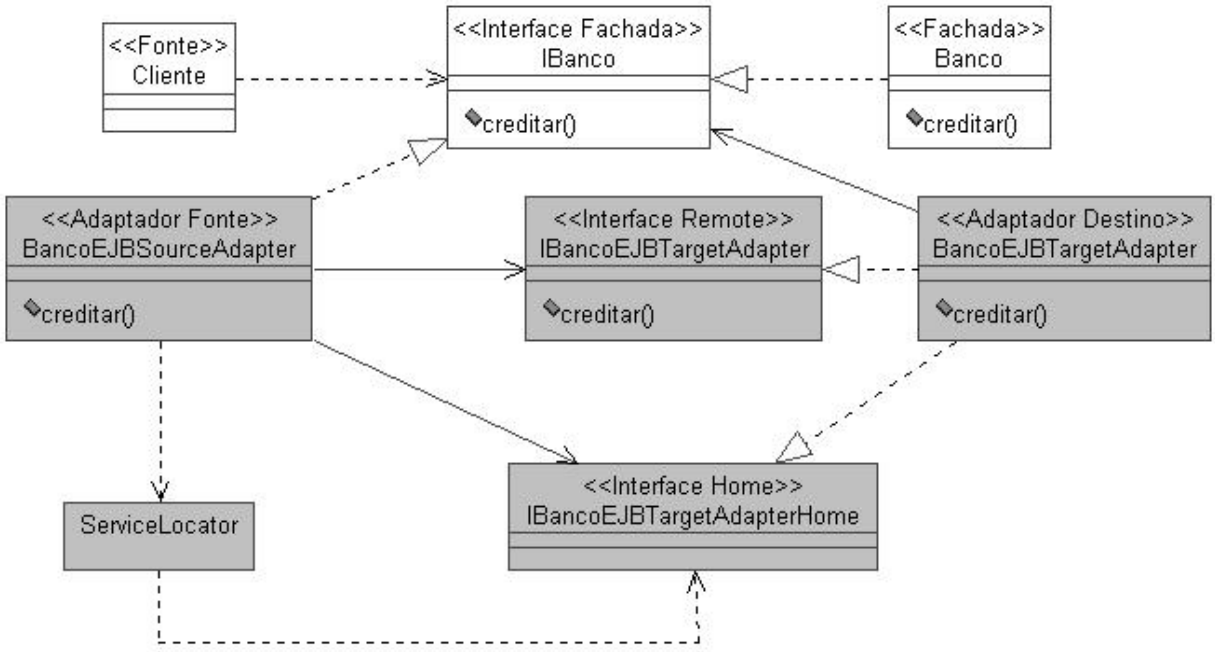

Figura 3: Estrutura de uma aplicação bancária de acordo com o padrão DAP-EJB.

Implementação

Adaptador destino implementado como *Stateless session bean*

O adaptador destino é implementado com um *stateless session bean* pelo fato de componentes dessa natureza representarem objetos cujas instâncias são equivalentes no *container*[2]. Uma mesma instância de um *stateless session bean* pode servir às requisições de diferentes clientes, minimizando os recursos necessários para suportar uma grande quantidade destes. Além disso, ao adaptador destino também podem ser atribuídos aspectos relacionados aos serviços de transações e segurança, por exemplo.

ServiceLocator para acesso às referências remotas dos *beans*

A tarefa para ter acesso a um componente EJB é comum para todos os clientes (externos ou internos) que precisam acessar seus serviços. Isto implica que muitos tipos de clientes repetidamente utilizam os serviços JNDI [10], uma API que fornece um conjunto de interfaces e classes para acessar uma vasta quantidade de recursos, entre os quais permite a localização de objetos remotos, o que resulta em código duplicado nos mesmos. Além disso, o processo necessário para localizar e obter referências remotas aos *homes* dos *beans* gasta recursos significativos do servidor de aplicação, o que pode causar impacto no desempenho do sistema. Neste contexto, a classe `ServiceLocator` [6], um padrão que visa abstrair a complexidade do processo de localização e criação dos *beans*, bem como melhorar o desempenho do sistema, é utilizada no padrão DAP-EJB para auxiliar o processo de criação das referências remotas.

Código

Nesta seção é apresentado o código para os elementos do exemplo do padrão. A interface com o usuário cria um `BancoEJBSourceAdapter` e delega as requisições do cliente para este. O adaptador fonte é uma classe Java "pura" e implementa a interface da fachada `IBanco` de modo a permitir que a interface com o usuário não tenha conhecimento da tecnologia de distribuição sendo utilizada.

O adaptador fonte declara como atributo as interfaces *home* e remota do adaptador destino, por questões de eficiência. Estes atributos são representados por `h` e `banco`, respectivamente.

```
public class BancoEJBSourceAdapter implements IBanco {
  private IBancoEJBTargetAdapter banco;
  private IBancoEJBTargetAdapterHome h;
```

Com o auxílio dos métodos da classe `ServiceLocator`, uma única instância da interface `IBancoEJBTargetAdapterHome` é obtida. O método auxiliar `conectar` realiza este processamento.

[2]Nome dado ao ambiente de execução dos compoenentes EJB.

```
    private void conectar() throws CommunicationException {
       Class home = IBancoEJBTargetAdapterHome.class;
       try{
          if (h == null){
              h = (IBancoEJBTargetAdapterHome)
                   ServiceLocator.getInstance().getHome("banco",home);
          }
          this.banco = h.create();
      } catch(ServiceLocatorException e){
                throw new CommunicationException (...);
      } catch(CreateException e){
                throw new CommunicationException (...);
      }
    }
```

A partir do método `getHome` da classe `ServiceLocator`, a referência à interface *home* do adaptador destino é obtida. Além disso, `conectar` também obtém a referência à interface remota do adaptador destino (`IBancoEJBTargetAdapter`), a partir do método `create` de sua interface *home*.

A exceção `ServiceLocatorException` é uma exceção de aplicação lançada pelo método `getHome` se alguma falha acontecer quando da obtenção do `home` do adaptador destino. A exceção `CreateException` é lançada pelo método `create` se a instância do adaptador destino não puder ser criada. As exceções de aplicação específicas de EJB são trocadas no adaptador fonte pela exceção genérica `CommunicationException`. Esta exceção não depende de qualquer tecnologia de distribuição particular e é definida de modo a permitir que o cliente seja isolado de exceções específicas da API de distribuição. No construtor do adaptador fonte o método `conectar` é chamado. Assim,

```
    public BancoEJBSourceAdapter() throws CommunicationException {
          conectar();
    }
```

quando uma instância do adaptador fonte é criada, o processo de localização e criação da instância do adaptador destino é também realizado.

Após obter a referência remota do *bean*, o adaptador fonte está pronto para executar chamadas aos métodos de negócio do adaptador destino. O método `creditar` do adaptador fonte delega as requisições da interface com o usuário para o adaptador destino.

```
  public void creditar(String numeroConta, double saldo)
           throws CommunicationException, ContaNaoExisteException {
     try{
      banco.creditar(numeroConta, saldo);
     } catch(RemoteException e){
         throw new CommunicationException(...);
     }
 }
...
}
```

Pelo fato do adaptador fonte invocar os métodos remotos do adaptador destino, a exceção `RemoteException` pode ser lançada se ocorrer alguma falha originada a partir da invocação remota. Neste caso, `RemoteException` deve ser trocada pela exceção genérica `CommunicationException`. Assim, o cliente não é exposto às exceções específicas da API de distribuição. A interface `IBancoEJBTargetHome` representa a interface *home* do adaptador destino e herda a interface `EJBHome`.

```
public interface IBancoEJBTargetAdapterHome extends EJBHome {
   public IBancoEJBTargetAdapter create()
      throws CreateException, RemoteException;
}
```

A exceção `RemoteException` deve ser declarada em todo método da interface *home* de EJB, e `CreateException` é uma exceção de aplicação específica de EJB lançada quando referências remotas do *bean* não podem ser criadas. O adaptador destino é um *stateless session bean* e por isso apresenta um único método `create`, sem argumentos, em sua interface *home*. Tal método é responsável por criar as suas referências remotas. A interface `IBancoEJBTargetAdapter` representa a interface remota do adaptador destino e declara os métodos invocados pelo adaptador fonte.

```
public interface IBancoEJBTargetAdapter extends EJBObject {
  public void creditar(String numero,double valor)
           throws ContaNaoExisteException, CommunicationException,
                  RemoteException;
 ...
}
```

Esta interface é o tipo da referência ao adaptador destino e seus métodos devem lançar `RemoteException`. A exceção `ContaNaoExisteException` é uma exceção de aplicação. Além do método `creditar`, outros métodos podem ser declarados nesta interface.

A classe `BancoEJBTargetAdapter` é a classe que representa o adaptador destino.

```
public class BancoEJBTargetAdapter implements SessionBean {
  private IBanco banco;
  private SessionContext context;
```

Esta possui uma referência à interface da fachada `IBanco` no intuito de delegar os serviços para esta executar. Um atributo da interface `SessionContext` também deve ser declarado no adaptador destino, uma vez que é um *session bean*. Esta interface é utilizada pelo *container* durante o ciclo de vida do *session bean*. Na realidade, o *container* tem acesso às informações de um *session bean* através desta interface.

O método `ejbCreate` de `BancoEJBTargetAdapter` obtém uma instância da classe fachada através do método `getInstance`.

```
  public void ejbCreate()
           throws CreateException, RepositorioException,
                  CommunicationException{
       banco = Banco.getInstance();
 }
...
```

Após obter a instância da fachada, o adaptador destino delega a invocação de seus métodos para os métodos desta. Por exemplo, quando o método `creditar` da classe `BancoEJBTargetAdapter` é executado,

```
public void creditar(String numero,double valor)
         throws ContaNaoExisteException, CommunicationException{
         banco.creditar(numero,valor);
}
  ...
```

o método `creditar` da fachada é invocado. Os métodos da interface da fachada `IBanco` declaram `CommunicationException`. Esta exceção genérica é declarada em `IBanco` prevendo que a aplicação se tornará distribuída. Por isso, os métodos do adaptador destino e sua interface remota devem também declarar esta exceção.

Por se tratar de um *session bean*, `BancoEJBTargetAdapter` deve implementar a interface *SessionBean*.

```
    public void ejbRemove() { }
    public void ejbActivate() { }

    public void ejbPassivate() { }
    public void setSessionContext(SessionContext sc){
      this.context = sc;
    }
}
```

O *container* utiliza os métodos dessa interface para notificar as instâncias dos *session beans* sobre os eventos do seu ciclo de vida.

Variações

Variações do DAP-EJB são possíveis. Por exemplo, a fachada do sistema poderia fazer o papel do adaptador fonte e o adaptador destino, um *session bean*, seria introduzido entre esta é a coleção de negócio da aplicação.

Esta é uma abordagem simplificada para o uso dos adaptadores, uma vez que um menor número de classes deve ser gerado, além disso, os benefícios como o uso de *session beans* (distribuição e gerenciamento de transações, por exemplo) são mantidos. No entanto, perde-se um pouco em extensibilidade, uma vez que código específico de EJB é inserido na camada de negócio do sistema (entre a fachada e as demais classes que compõem a camada de negócio).

Usos Conhecidos

O DAP-EJB tem sido utilizado em um sistema de informação para o serviço público de saúde. Este sistema foi desenvolvido para ser executado via *Web*. Neste caso, servlets [9] agem como clientes do adaptador fonte. O adaptador fonte interage com o adaptador destino, e este com a fachada, localizada na mesma máquina.

Um outro uso do DAP-EJB é em um sistema que fornece serviços para o gerenciamento de contabilidade, controle de acesso e serviços financeiros. Na realidade, este sistema utiliza a variação do DAP-EJB, citada na seção anterior. Neste sistema, a fachada da aplicação é utilizada como o adaptador fonte, fazendo acesso ao adaptador destino da aplicação.

O DAP-EJB poderia também ser utilizado em outros tipos de sistema, tais como:

- Um sistema para gerenciar clientes de uma empresa de telecomunicação. O sistema é capaz de registrar telefones móveis, gerenciar informações de clientes e a configuração dos serviços de telefonia. Este sistema pode ser utilizado via *Web*.

- Um sistema para provas interativas. Este sistema tem sido utilizado para fornecer diferentes tipos de provas, tais como simulados baseados em exames de seleção para a universidade, ajudando os alunos a avaliar seus conhecimentos antes de realizarem exames reais.

- Um sistema de supermercado complexo. Este sistema será usado em vários supermercados e já está sendo utilizado em outras empresas do mesmo ramo.

Padrões Relacionados

- DAP (Distributed Adapters Pattern). No trabalho *A Design Pattern for Object-Oriented Distributed Applications* [2], foi descrito o padrão DAP, o qual é utilizado no contexto de comunicação remota entre dois componentes, e visa isolar o código relacionado à API de comunicação destes. Este serviu como base para a adaptação do padrão para *Enterprise JavaBeans* (DAP-EJB) proposto aqui.

  O padrão apresentado neste artigo também utiliza adaptadores como o DAP, no entanto, estes não isolam somente os aspectos de distribuição, mas também são responsáveis pelo gerenciamento de transações e segurança, por exemplo. É importante destacar, no entanto, que o gerenciamento de transações e segurança obtidos com o DAP-EJB só é possível devido a uma característica intrínseca da tecnologia EJB, a qual permite o gerenciamento automático desses aspectos por intermédio de seus componentes. Portanto, esta não é uma característica do padrão, mas sim um benefício obtido com a tecnologia EJB.

  Na realidade, o padrão DAP-EJB, permitiu mostrar que o DAP pode ser também implementado com EJB, com pequenas adaptações, por exemplo, com relação aos aspectos de gerenciamento dos serviços citados acima. A estrutura do padrão permanece a mesma, as mudanças dizem respeito muito mais às características do adaptador destino.

- *Bussiness Delegate* [6] e *Session Facade* [6]. Estes padrões são similares aos adaptadores fonte e destino, respectivamente. No entanto, eles não visam estruturar uma aplicação independente de tecnologia, apesar de poderem fazê-lo. A forma como estão estruturados não segue esta filosofia. O *Bussiness Delegate* é utilizado para isolar o cliente das especificidades da tecnologia EJB, como exceções e *lookup*, por exemplo. No entanto, o *Session Facade* não tem o mesmo propósito do adaptador destino, uma vez que não é utilizado para isolar a tecnologia de EJB do resto da

aplicação. Ele serve como um único ponto de acesso para os demais componentes do sistema (entity beans [14] e afins).

- *Wrapper-Facade* [12]. Este padrão encapsula funções de baixo nível (tais como *sockets* e *threads*) da aplicação. O DAP-EJB encapsula a API de distribuição EJB, da aplicação.

- *Adapter*, *Facade*. DAP-EJB é implementado utilizando os padrões de projeto [7] *Adapter* e *Facade*.

- *Singleton* [7]. Um objeto da classe `Fachada` é implementado como um *Singleton*.

- *Abstract Factory*. Em conjunto com o DAP-EJB, classes auxiliares são utilizadas para o propósito de configuração. Tais classes são estruturadas de acordo com o padrão *Abstract Factory* [7]. Assim, dependendo das informações contidas, por exemplo, em um arquivo de configuração, o sistema poder ser executado localmente(resultando em uma referência para a fachada) ou remotamente (resultando em uma referência para o adaptador fonte).

  Desta forma, a utilização de fábricas permite que o código do cliente seja isolado das mudanças ligadas ao código de distribuição, aumentando, assim, a modularidade do sistema.

3 PDC-EJB: Um padrão para Persistência com EJB

Objetivo

Fornecer uma forma de estruturar aplicações complexas implementadas com EJB de modo a separar o código de acesso a dados do código de negócio e de interface com o usuário. Classes específicas são utilizadas para separar estes conceitos, e interfaces garantem a independência entre a camada de negócio e a camada de dados de um sistema.

Contexto

O padrão PDC-EJB está inserido no contexto de aplicações que utilizam algum tipo de armazenamento e acesso a dados de forma persistente.

Problema

O desenvolvimento *ad hoc* de aplicações que utilizam alguma plataforma de persistência, para armazenamento e recuperação de seus objetos, leva a sistemas que misturam código de acesso a dados com o código de negócio da aplicação. Em particular, aplicações construídas desta forma não podem dispor de objetos de negócio reutilizáveis por outras aplicações, que utilizem diferentes tecnologias para a persitência dos dados. Da mesma forma, se a plataforma de persistência for substituída (JDBC [15] por EJB, por exemplo), o impacto das mudanças no código do sistema não é localizado, ou seja, as classes relacionadas ao domínio do negócio da aplicação também devem ser modificadas.

Desta forma, caso seja necessário adaptar um sistema para utilizar outro mecanismo de persistência, tem-se, na verdade, que desenvolver um novo sistema. Ou seja, reutilização e extensibilidade são seriamente comprometidas em sistemas desenvolvidos sem estruturação alguma, pois não há uma distinção clara entre o código de negócio, que contém regras e objetos de negócio, e o código de dados.

Forças

- Problemas relacionados aos requisitos de negócio do sistema devem ser manipulados independente das operações de acesso a dados;
- A modificação no código do sistema para suportar persistência deve ser minimizada;
- O tipo de mecanismo de armazenamento[3] pode ser substituído durante a vida útil de um sistema;
- Classes de negócio podem ser reutilizadas em sistemas diferentes.

Solução

O padrão PDC-EJB utiliza um conjunto de classes para estruturar o código relacionado ao domínio de objetos do negócio e o código de acesso a dados, a fim de evitar a mistura de código relacionado a tais aspectos, obtendo, com isso, extensibilidade e reutilização das classes. Para tal, o padrão utiliza a separação das classes do sistema em dois tipos:

- classes para descrever os objetos de negócio, resultantes dos requisitos funcionais; e
- classes para manipulação e armazenamento de dados.

A comunicação entre esses dois tipos de classes é realizada através de interfaces que garantem uma maior independência do código de negócio em relação à forma como efetivamente são implementadas as operações de persistência (o acesso ao banco de dados ou outro mecanismo, como arquivos, por exemplo).

Estrutura

O diagrama de classes da Figura 4 destaca a estrutura do padrão PDC-EJB. As classes que denotam os elementos que fazem parte do padrão são explicadas a seguir.

- Fachada

  Esta classe representa todos os serviços do sistema e define uma interface que abstrai os objetos de negócio da aplicação [4]. Ela mantém uma referência para os vários objetos da classe `ColecaoDeNegocio` da aplicação e delega as chamadas para estes.

[3] Termo utilizado aqui, para descrever o meio no qual os objetos de negócio do sistema são armazenados, por exemplo, um banco de dados relacional.

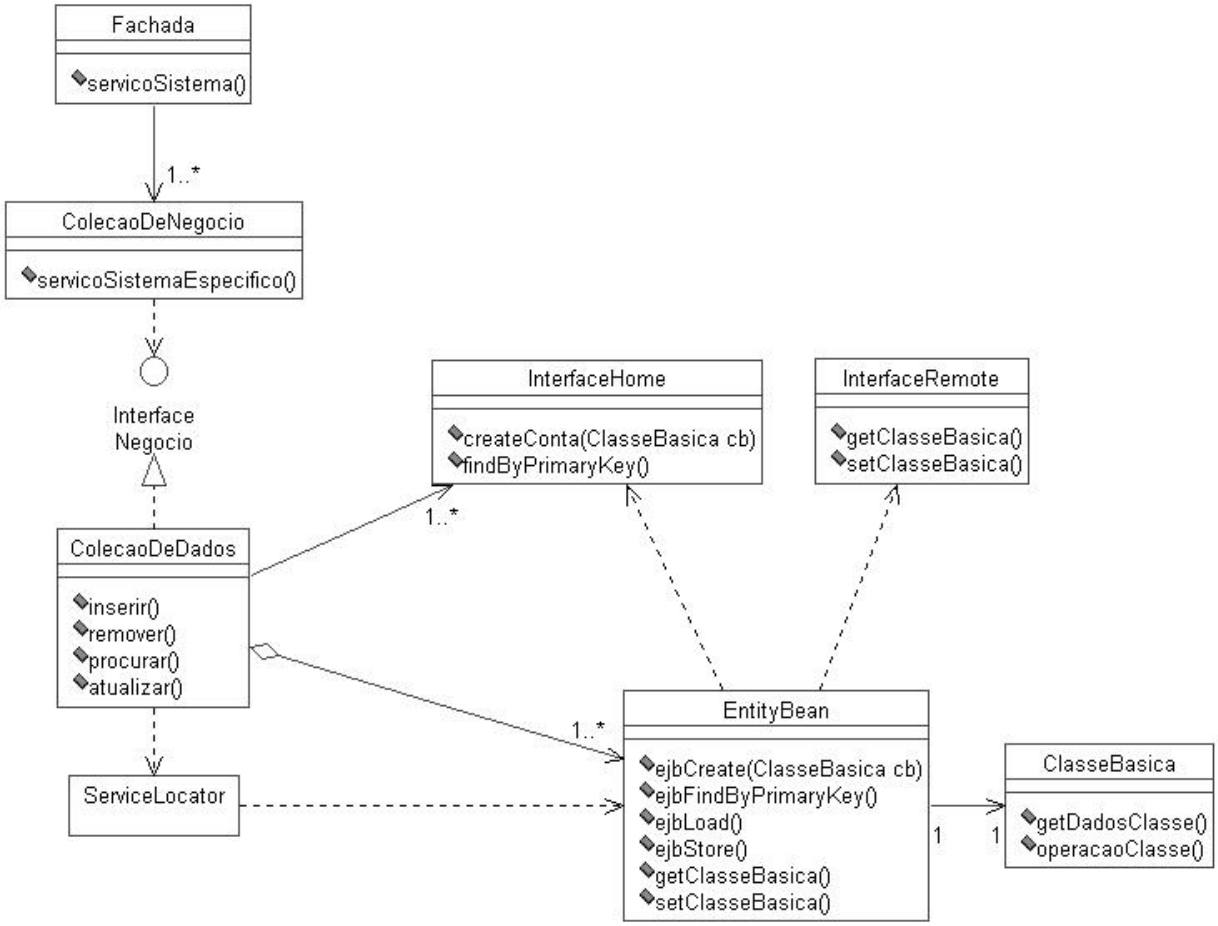

Figura 4: Estrutura do PDC-EJB.

- ClasseBasica

  A classe básica representa o objeto básico de negócio (por exemplo, conta, cliente) refletindo claramente o domínio do problema. Os métodos desta classe contêm somente operações relacionadas aos requisitos funcionais do sistema e métodos `get` e `set` para obter informações sobre os atributos desta classe.

- EntityBean

  Para cada classe básica da aplicação, deve existir uma classe *entity bean* correspondente. Esta classe possui operações para persistência dos objetos das classes básicas no sistema. A classe `EntityBean` possui uma dependência com sua classe básica correspondente e seus métodos manipulam os objetos desta. Porém, a classe básica não possui dependência com o *entity bean*, podendo ser reutilizada em outras aplicações.

- ColecaoDeNegocio

  Esta classe representa o agrupamento dos objetos básicos de negócio, tendo como operações a inserção, busca e exclusão de elementos do repositório, verificações ou testes de pré-condições relativos à estas manipulações e mais as operações que invocam as operações típicas de objetos de negócio.

- ColecaoDeDados

  A coleção de dados contém código de manipulação da estrutura de armazenamento persistente, correspondente a cada classe básica de negócio. O código dos métodos da mesma depende da API específica da plataforma utilizada para armazenamento (no caso, EJB é utilizado para persistência dos dados). Assim, mudanças na API de persistência não causam impacto na camada de negócio da aplicação, o impacto é centralizado nesta classe (uma vez que a interface negócio-dados isola essas mudanças). Uma classe coleção de dados implementa sua interface negocio-dados correspondente. Esta última é apresentada a seguir.

- InterfaceNegocioDados

  Esta interface possui assinaturas dos métodos de acesso aos dados, como inserção, atualização, consultas e exclusão. Esta interface estabelece uma comunicação entre os objetos das classes coleção de negócio e os objetos das classes coleção de dados, proporcionando extensibilidade. Uma classe `ColecaoDeNegocio` possui uma referência a esta interface. Desta forma, a classe `ColecaoDeNegocio` não precisa ser modificada quando a classe `ColecaoDeDados` mudar, desde que esta sempre implemente esta interface.

Os elementos que correspondem à fachada, coleção de negócio, coleção de dados e interface negócio-dados são implementados como classes Java "puras". Apesar de estar trabalhando com EJB, existem justificativas para não tornar tais classes *session beans*. O uso de *sessions beans* proporciona muitos benefícios. Para um *bean*, o *container* gerencia vários serviços de forma automática. Estes serviços envolvem gerenciamento de transações, controle do acesso concorrente, gerenciamento das instâncias no servidor e, por conseguinte, gerenciamento da memória, entre outros. A fim de fornecer todos esses serviços de forma transparente, o *container* executa uma vasta quantidade de processamento, incluindo geração de classes e mecanismos que visam auxiliá-lo no emprego correto e adequado dos serviços que gerencia.

Desta forma, toda vez que um método de um *bean* é requisitado, seja por um cliente externo ou mesmo por outro *bean*, o *container* intercepta cada chamada antes de reenviá-la para o objeto apropriado. Essa interceptação é necessária para que o *container* tenha conhecimento de todas as características do *bean* em questão: qual o tipo de mecanismo de transação, qual o atributo de transação para o método chamado, que componentes o *bean* referencia, que outros recursos utiliza, e assim por diante. Após ter acesso a todas essas informações, o *container* executa os processamentos devidos e finalmente repassa a chamada para a instância apropriada.

De fato, a execução de um método implica em vários processamentos que devem ser executados pelo *container* antes de passar a requisição da chamada para o *bean*. Em um sistema que utiliza muitas classes *session beans*, o desempenho do sistema tende a degradar. Por isso, as classes fachada, coleção de negócios, e interface negócio dados não são implementadas como *beans*.

Além das razões supracitadas relacionadas à eficiência, a implementação das classes de negócio como classes Java "puras", permite que estas possam reutilizadas em outras aplicações que utilizem uma plataforma de distribuição diferente, uma vez seu código não está atrelado a uma tecnologia de distribuição específica.

A coleção de dados também é implementada como uma classe Java, no entanto ela possui código relativo a API de EJB, uma vez que utiliza os serviços de persistência de *entity beans*. Esta classe, bem como a interface negócio-dados são importantes porque evitam que a coleção de negócio tenha aspectos específicos da API de EJB. Isto permite o desacoplamento entre a tecnologia utilizada para pesistência dos dados e camada de negócio do sistema.

Outro aspecto a ser considerado na estrutura do padrão PDC-EJB é o relacionamento entre as classes básicas de negócio e *entity beans*. *Entity beans* fornecem operações para persistir os dados e para realizar parte da lógica do negócio numa única classe. Isto resulta na mistura de papéis, ou seja, código de acesso a dados persistentes e código para manipulação da lógica de negócio são especificados na mesma classe. Os métodos de negócio de um *entity bean* são declarados em sua interface remota. Isto implica que toda invocação de um método de negócio de um *entity bean* é feita remotamente. Além disso, sempre que é feita uma chamada de método sobre a interface *home* de um *entity bean* (para criar ou localizar uma instância da entidade), o cliente recebe uma referência remota do objeto e não sua cópia. A abordagem com *entity beans* traz algumas consequências:

- O acesso concorrente ao *entity bean* é gerenciado pelo *container*

  Como cada cliente tem acesso a uma referência remota do *entity bean*, fica a cargo do *container* organizar o acesso e controlar a sincronização entre os clientes de modo a manter o estado do *bean* sempre coerente. Este é um dos grandes benefícios de *entity beans*, visto que o programador não precisa se preocupar em fornecer código necessário ao gerenciamento do acesso concorrente.

- Os métodos de negócio são declarados na interface remota do *bean*

  Isto implica que quando o cliente deseja executar um método de negócio, ele o faz de forma remota. Isto pode causar impacto no desempenho do sistema. Em particular, no caso da entidade possuir uma grande quantidade de informações que são acessadas por métodos `get` e `set`, pode-se ter um gargalo no tráfego dessas informações pela rede. Há um custo associado à invocação de métodos e à troca de dados remota.

- Mistura de conceitos

  Entity beans misturam código relacionado à forma como os dados são persistidos (fornecidos pelos métodos da interface *home*) com os métodos de negócio, fornecidos através da interface remota.

Com o padrão PDC-EJB é possível minimizar ou evitar o impacto causado pelos últimos itens. As classes básicas de negócio ficam responsáveis por processar parte da lógica de negócio e os *entity beans* assumem o papel de persistir os dados resultantes deste processamento. Com isso, é possível tornar clara a separação dos papéis. Para cada classe básica existe um *entity bean* associado. Para facilitar o entendimento, a parte da estrutura do padrão correspondente a este aspecto é ilustrada na Figura 5.

Nada é mudado com relação à classe básica; acrescenta-se um *entity bean* para facilitar a implementação da coleção de dados. A coleção de dados utiliza os serviços de persitência de *entity beans*. Os métodos `getClasseBasica` e `setClasseBasica` são utilizados pela

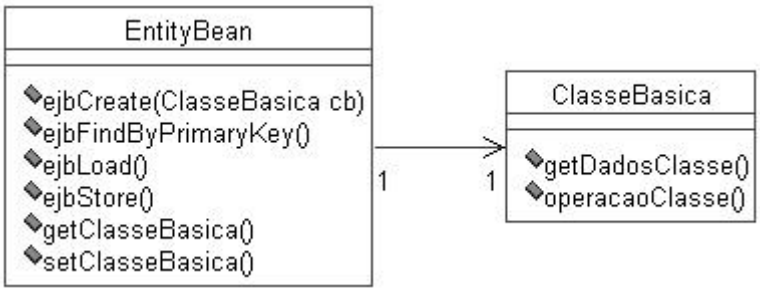

Figura 5: Estrutura para o uso de classes básicas e *entity beans*.

coleção de dados para retornar um clone do objeto `ClasseBasica` e atualizar o objeto `ClasseBasica`, respectivamente.

Para realizar esta abordagem, cada instância do *entity bean* deverá estar sincronizada com a instância da classe básica de negócio a fim de executar as alterações de forma coerente e mantendo a integridade dos dados a serem persistidos.

Com esta abordagem os clientes têm acesso às cópias dos objetos básicos de negócio em vez de referências remotas dos *entity beans*. Executam o processamento localmente, a partir destas cópias e o resultado deste processamento é persistido no banco com o auxílio de *entity beans*.

Pelo fato dos clientes manipularem cópias dos objetos básicos de negócio em vez de referências remotas de entity beans, o mecanismo de controle de concorrência automático fornecido por entity beans é perdido. Entretanto, os benefícios alcançados com a estrutura do padrão compensam essa perda uma vez que o controle de concorrência pode ser facilmente solucionado com o auxílio de mecanismos, tais como *timestamp* [13].

Dinâmica

A Figura 6 apresenta o diagrama de seqüência para um dos possíveis usos do PDC-EJB. Neste cenário, quando um método da `Fachada` é invocado, é realizada a delegação para um método da `ColecaoDeNegocio` (no exemplo, uma operação de consulta que recupera um objeto do banco de dados). O objeto da `ColecaoDeNegocio` executa possíveis validações e testes relativos aos dados informados para a consulta, e invoca a operação `procurar` sobre a `ColecaoDeDados`. Esta utiliza as operações de `EntityBean` para o acesso ao banco de dados. Através da operação `findByPrimaryKey` obtém-se a referência remota do objeto armazenado no banco. A partir desta referência, o objeto da classe básica é obtido através da operação `getClasseBasica` e retornado para o cliente, como resultado da consulta.

Conseqüências

A utilização do PDC-EJB traz as seguintes vantagens:

- Reutilização e extensibilidade

  Devido à estrutura modular do padrão, mudanças na camada de dados não causam impacto nas demais camadas. Interfaces entre a camada de negócio e a camada de dados provêem essa extensibilidade. Isto permite que a camada de negócio não tenha

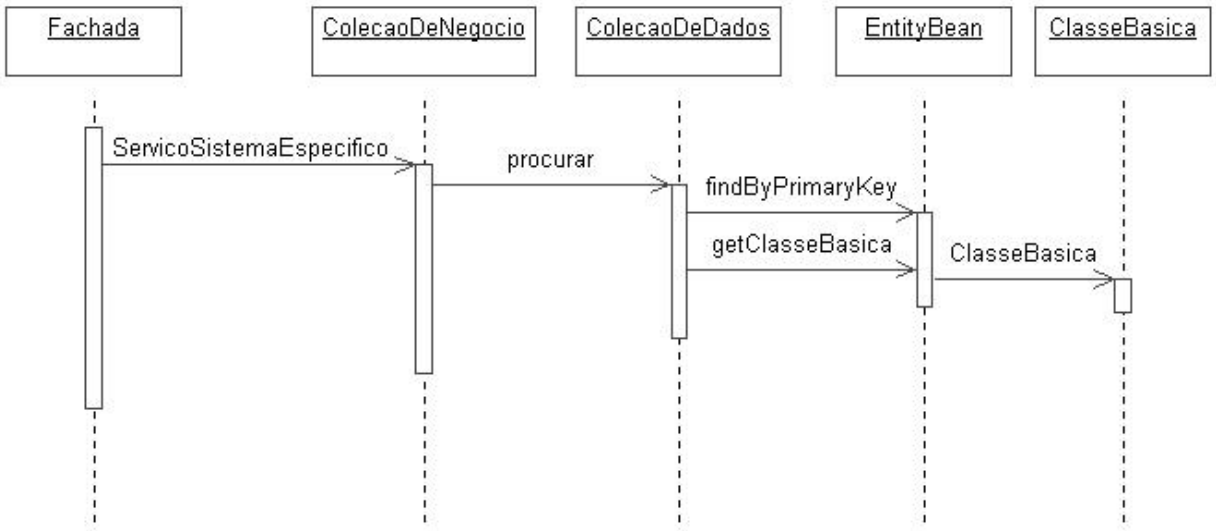

Figura 6: Diagrama de Sequência do PDC-EJB.

conhecimento se a tecnologia utilizada para persistir os dados é JDBC, EJB, etc. Além disso, a estrutura que o padrão apresenta possibilita que as classes básicas de negócio possam ser facilmente reutilizadas em outras aplicações que utilizem outras tecnologias de banco de dados.

- Redução do tráfego na rede

  Em vez de várias chamadas a métodos `get` para obter os atributos de um *bean*, o padrão fornece uma única chamada para obter todos os valores encapsulados em um objeto básico. Isso faz com que uma quantidade de dados seja transferida pela rede em uma única chamada remota. Isto sem dúvida diminui a carga imposta pelos acessos remotos e, consequentemente, melhora o desempenho do sistema.

- Facilidades para teste

  A classe básica possui apenas métodos de negócio e métodos "acessores", sem código de acesso a dados. Desta forma, fica mais fácil testar somente a funcionalidade do sistema usando uma versão volátil do mesmo, sem o uso de *entity beans* apenas com coleções de dados voláteis.

- Simplificação de *entity beans* e interfaces remotas

  Além dos métodos `get` e `set`, os *entity beans* possuem somente os métodos padrão para persistir os dados; os métodos de negócio são executados pela classe básica. O cliente do *entity bean*, no caso a coleção de dados, tem acesso ao objetos básicos de negócio a partir dos métodos `getClasseBasica` e `setClasseBasica`, os únicos métodos declarados na interface remota do *bean*. Esta estruturação também fornece um maior potencial para geração automática de código.

Por outro lado, o PDC-EJB apresenta as seguintes desvantagens:

- Não utilização do controle de concorrência de EJB

  De acordo com a estrutura do padrão, o cliente do sistema tem acesso aos clones dos objetos básicos e não às referências remotas dos *entity beans*. Os clientes executam modificações sobre cópias locais do objeto da classe básica. Uma vez que as modificações foram realizadas, o cliente invoca o método `setClasseBasica` (na verdade, quem invoca este método é a coleção de dados), passando o objeto modificado para o *entity bean*, e este se encarrega de atualizar os seus atributos e persistí-los no banco de dados. O problema acontece quando outros clientes requisitam o mesmo objeto.

  Apesar do *entity bean* atualizar os valores, este não está ciente dos vários clientes que obtiveram cópias do mesmo objeto e por isso não pode propagar a atualização do objeto para os vários clientes. Estes clientes acabam tendo instâncias de objetos que não refletem seu estado real no banco. No entanto o uso de mecanismos como *timestamp* [13], por exemplo, podem ser utilizados para resolver essa deficiência.

- Duplicação

  Os atributos da classe básica e do *entity bean* correspondente são duplicados. Isto se deve à limitação da especificação 1.1 de EJB, que exige que *entity beans* CMP declarem seus atributos públicos. De outra forma, o *bean* poderia herdar a classe básica. Os atributos da classe básica são declarados *private* por questões de encapsulamento. Isto implica que mudanças nos atributos da classe básica devem ser refletidas nos atributos do *entity bean*. No entanto, a mudança é localizada e poderia ter um apoio preconizado para manter a consistência.

- Produtividade

  Duplica-se o número de objetos que representam entidades persistentes: é necessário um *entity bean* e uma classe básica para cada entidade forte. Torna-se então necessário o uso de ferramentas para gerar o código do *entity bean* a partir da classe básica, por exemplo, bem como manter a consistência após alterações. Além disso, um gerador de código poderia também automatizar a criação dos adaptadores e classes auxiliares utilizadas na aplicação.

Exemplo

O diagrama de classes UML da Figura 7 ilustra os elementos que compõem o padrão através de uma simples aplicação bancária.

As classes `Banco`, `CadastroDeContas` e `Conta` correspondem aos objetos do domínio do problema. As classes que lidam com os aspectos de persistência dos dados e, portanto, fazem parte da camada de dados do sistema, são representadas pelas classes `IRepositorioConta`, `RepositorioDeContasEJB` e `ContaEJB`. A comunicação entre os dois tipos de classes é realizada através de interfaces. Este aspecto é importante porque permite estruturar os aspectos de forma modular reduzindo o impacto causado por possíveis modificações do sistema tanto para requisitos funcionais (como a introdução de novos serviços) quanto não funcionais (como adaptar o sistema para suportar outro mecanismo de persistência ou melhorar a *performance* das consultas).

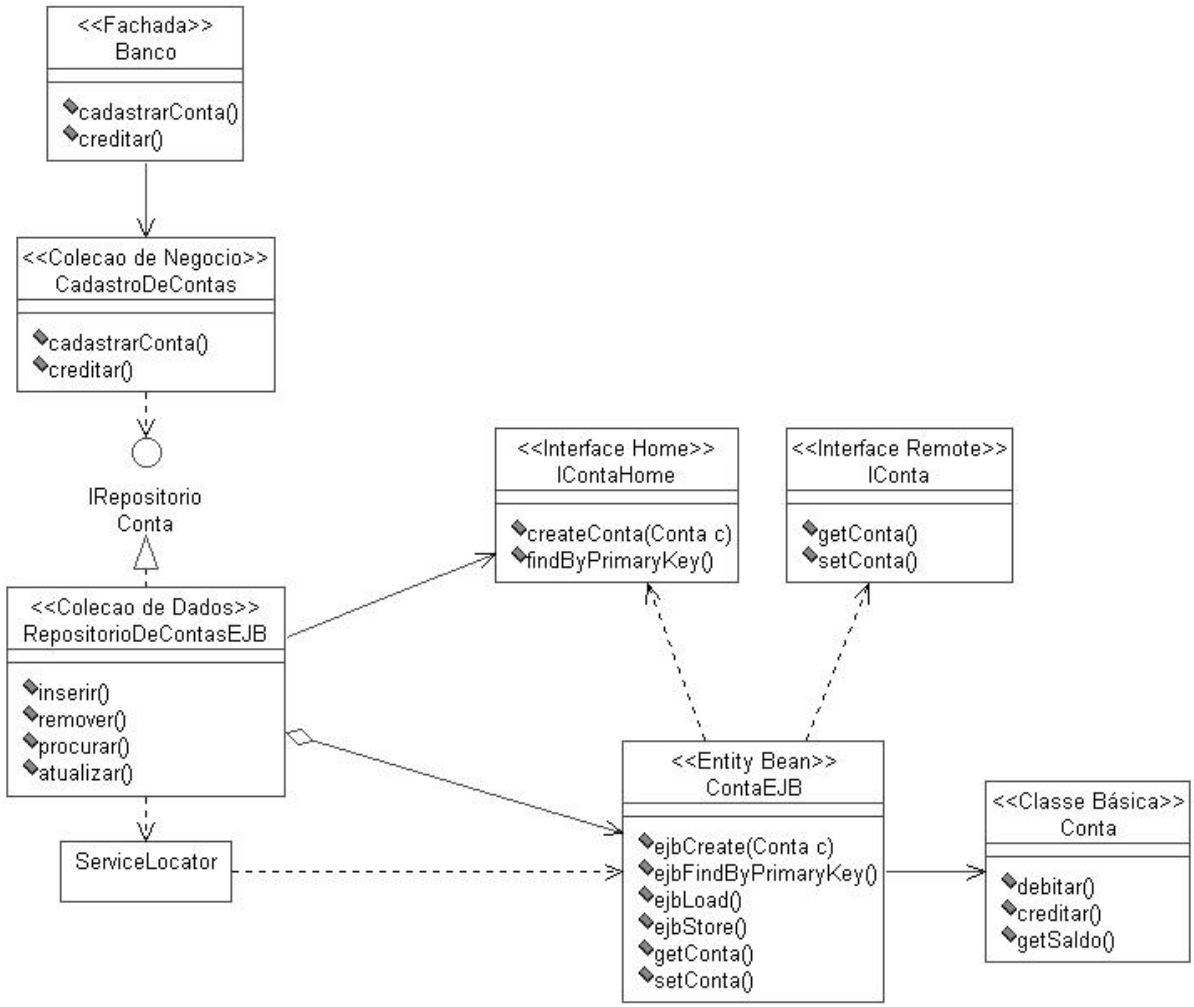

Figura 7: Exemplo de uma aplicação bancária estruturada de acordo com o padrão PDC-EJB.

Implementação

ServiceLocator para localização de *entity beans*

A tarefa para ter acesso a um componente EJB é comum para todos os clientes (externos ou internos) que precisam acessar seus serviços. Isto implica que muitos tipos de clientes repetidamente utilizam os serviços JNDI [10], uma API que fornece um conjunto de interfaces e classes para acessar uma vasta quantidade de recursos, entre os quais permite a localização de objetos remotos, o que resulta em código duplicado nos mesmos. Além disso, o processo necessário para localizar e obter referências remotas aos *homes* dos *beans* gasta recursos significativos do servidor de aplicação, o que pode causar impacto no desempenho do sistema. Neste contexto, a classe `ServiceLocator` [6], um padrão que visa abstrair a complexidade do processo de localização e criação dos *beans*, bem como melhorar o desempenho do sistema, é utilizada no padrão PDC-EJB para auxiliar o processo de criação das referências remotas de *entity beans*.

Transações

O padrão PDC-EJB é utilizado em conjunto com o padrão DAP-EJB, descrito da Seção 2. Neste caso, o adaptador destino, um *session bean*, é responsável por gerenciar as transações associadas aos seus métodos, os quais simplesmente delegam todas as invocações para os métodos da fachada.

A partir de atributos de transação especificados para os métodos do adaptador destino, o *container* gerencia o contexto transacional dos mesmos. A especificação garante que o contexto transacional é propagado a todos os objetos que participam na realização de uma operação, incluindo a fachada. Portanto, no padrão PDC-EJB, uma classe fachada não precisa especificar código para o tratamento de transações de seus métodos, uma vez que esta tarefa é delegada ao *container*.

Da mesma forma, aspectos relativos ao gerenciamento de segurança também ficam à cargo do padrão DAP-EJB. O gerenciamento de transações, bem como o gerenciamento de segurança são realizados pelo adaptador destino.

Código

Esta seção apresenta a implementação dos principais elementos do PDC-EJB. O exemplo utilizado é a aplicação bancária, introduzida na Figura 7. A explicação do PDC-EJB é realizada de forma *bottom up* a fim de facilitar o entendimento. Desta forma, o primeiro elemento a ser apresentado é a classe básica de negócio `Conta`, a qual reflete o domínio do problema.

A classe `Conta` é uma classe Java que implementa a interface `Serializable` [8]. Não existem métodos nesta interface e ela simplesmente indica para o sistema que um objeto pode ser transformado em um *stream* de bytes para poder ser transmitido pela rede. Os atributos

```
public class Conta implements java.io.Serializable {
  private String numero;
  private double saldo;
```

`numero` e `saldo` são declarados com visibilidade *private* por questões de encapsulamento e são inicializados no construtor da classe. Além disso, a classe básica também declara um construtor vazio.

Para cada atributo declarado na classe básica, são declarados métodos `get` e `set` correspondentes para a obtenção de informações sobre os atributos da classe. Os métodos

```
  public String getNumero() { return numero; }

  public double getSaldo() { return saldo; }
```

`getNumero` e `getSaldo` representam os métodos "acessores"para os atributos `numero` e `saldo`, respectivamente. Além destes, os métodos `set` para cada atributo também são declarados.

Além de métodos "acessores", a classe básica também possui métodos de negócio correspondentes ao domínio da aplicação. O método `creditar`, por exemplo,

```
    public void creditar(double valor) {
     saldo += valor;
    }
  }
```

corresponde a uma operação de negócio da classe `Conta`. Outros métodos de negócio, além de `creditar` também devem declarados nesta classe.

O *entity bean* `ContaEJB`[4] e suas interfaces *home* e remota, são apresentados. A interface *home* contém os métodos de acesso ao banco de dados `create` e `findByPrimaryKey`. Estes são métodos padrão de EJB utilizados para criar e localizar uma entidade banco, respectivamente.

```
public interface IContaHome extends EJBHome {
   public IConta create(Conta conta)
       throws RemoteException, CreateException;

   public IConta findByPrimaryKey(String numero)
       throws FinderException, RemoteException;
```

O tipo de retorno de tais métodos é a referência remota do *entity bean*. A exceção `RemoteException` é declarada na assinatura dos métodos da interface `IContaHome` pelo fato desta tratar-se de uma interface remota. `CreateException` e `FinderException` são exceções especificas de EJB e também devem ser declaradas nas assinaturas dos métodos `create` e `findByPrimaryKey`, respectivamante.

Além dos métodos específicos de EJB, outros métodos `findXX` podem ser especificados, como por exemplo o método `findByAll`

```
   public Collection  findByAll() throws FinderException, RemoteException;
}
```

que retorna uma coleção de referências a todas as entidades armazenadas no banco de dados.

A interface remota `IConta` declara somente métodos `get` e `set` para a classe básica `Conta`.

```
public interface IConta extends EJBObject {
  public Conta getConta() throws RemoteException;
  public void setConta(Conta conta) throws RemoteException;
}
```

Os métodos de negócio são executados localmente, a partir dos clones das classes básicas obtidos através do método `getConta` da interface remota. Após o processamento local dos métodos de negócio, o resultado pode ser atualizado através do método `setConta`.

A classe `ContaEJB` implementa a interface `EntityBean`, específica de EJB, a qual fornece métodos para a manipulação das entidades no banco, pelo *container*.

Os atributos do *entity bean* que devem ser mapeados para tabelas no banco de dados, são declarados públicos. Esta é uma restrição da especificação 1.1 de EJB. Neste caso, os dois atributos apresentados, `numero` e `saldo` são declarados com visibilidade `public`.

[4] O código do *entity bean* no exemplo apresentado utiliza persistência gerenciada pelo *container* (CMP).

```
public class ContaEJB implements EntityBean {
  public String numero;
  public double saldo;
  private Conta conta;
  private EntityContext context;
```

Um atributo do tipo da interface `EntityContext` também deve ser declarado para que o *container* possa ter acesso às informações dos *entity beans*. O *entity bean* também declara `conta` como atributo por questões de desempenho, para evitar que sempre seja necessário criar um objeto da classe `Conta` antes de enviá-lo para o cliente, através do método `getConta`.

Os atributos de `ContaEJB` são inicializados no método `ejbCreate`, o qual corresponde ao método `create` da interface `IContaHome` e é responsável por criar uma entidade no banco. O método recebe como parâmetro um objeto da classe `Conta`. Os atributos do *bean* são inicializados a partir do estado do atributo `c`. Desta forma, quando da criação de uma entidade no banco de dados, o estado da instância do *entity bean* (`ContaEJB`) é sincronizado com o estado da instância de sua classe básica (`Conta`) correspondente.

```
public String ejbCreate(Conta c) throws CreateException {
    numero = c.getNumero();
    saldo = c.getSaldo();
  return null;
}
```

Um objeto da classe básica é passado como parâmetro para o método `ejbCreate` de seu *entity bean* correspondente para evitar o tráfego de parâmetros pela rede, em vez de enviar vários atributos pela rede, o objeto completo e enviado, diminuindo o *overhead* associado. Após os atributos do *bean* serem inicializados, estes são inseridos no banco de dados. O tipo de retorno do método é um objeto do tipo da chave primária armazenada na tabela. No código apresentado, o retorno é `null` porque em persistência gerenciada pelo *container* (CMP), o *container* é responsável por gerar a chave primária da tabela é retorná-la como resultado da inserção. O código para inserção no banco é gerado pelo *container*.

Os métodos oriundos da interface `EntityBean`, são declarados e implementados pelas classes geradas pelo *container*, tendo seus métodos com o corpo vazio. Eles são responsáveis pela persistência da entidade no banco de dados,

```
void ejbStore() { }
void ejbLoad() { }
void ejbRemove() { }
```

sendo invocados pelo *container* durante a execução dos métodos pelo cliente.

Os métodos `getConta` e `setConta` são responsáveis por manter o estado das instâncias dos objetos de negócio (Conta, no exemplo) sincronizado com o estado das instâncias dos *entity beans* (ContaEJB, no exemplo).

```
public void setConta(Conta c) {
     numero = c.getNumero();
```

```
        saldo = c.getSaldo();
        conta = c;
  }

  public Conta getConta() {
      conta.setNumero(this.numero);
      conta.setSaldo(this.saldo);
      return conta;
  }

}
```

A classe `RepositorioDeContasEJB` representa a coleção de dados do padrão PDC-EJB. É uma classe Java "pura"e utiliza os serviços de persistência de *entity beans*. Isto acontece devido ao fato de *entity beans* representarem entidades persistentes e, portanto, em uma aplicação EJB são responsáveis por persistir tais entidades no banco de dados.

Esta classe implementa a interface negócio-dados `IRepositorioConta`, a qual fornece métodos para manipular os dados armazenados no banco de dados. Esta interface é apresentada mais adiante. Um atributo da interface *home* do *entity bean* é declarado na coleção de dados por questões de eficiência.

O acesso às referências dos *homes* dos *entity beans* é realizado com o auxílio do método auxiliar `getHome`. Este método faz acesso ao método de mesmo nome (getHome) da classe `ServiceLocator`. Através deste método, é possível localizar um *entity bean* mantendo uma única instância da referência remota à sua interface *home* durante a execução dos clientes.

```
  class RepositorioDeContasEJB implements IRepositorioConta {
      private IContaHome home;
        private IContaHome getHome()
           throws ServiceLocatorException {
           if (home ==null) {
              home= (IContaHome)
                     ServiceLocator.getInstance().getHome("conta",
                                                    IContaHome.class);
          }
        return home;
      }
}
```

Todos os métodos desta classe que acessam as operações de persistência de *entity beans*, utilizam o método auxiliar `getHome` para localizar o *entity bean*.

O método `inserir` é utilizado para incluir uma entidade no banco de dados. Recebe como parâmetro um objeto da classe básica (`Conta`) e chama o método `create` do *entity bean* passando `conta` como parâmetro. As exceções específicas de EJB relacionadas ao mecanismo de armazenamento de dados, são substituídas na coleção de dados pela exceção genérica `RepositorioException`. Por isso, esta é declarada na assinatura do método. A troca de exceções permite isolar a coleção de negócio da API de EJB.

```
  void inserir(Conta conta)  throws RepositorioException{
    try {
      IContaRemote contaBean = getHome().create(conta);
    } catch (Exception e) {
              throw new RepositorioException(e);
      }
  }
```

O método `procurar` localiza uma entidade no banco a partir de `codigo` e retorna um objeto da classe básica `Conta`. Para tal, utiliza o método de EJB `findByPrimaryKey`, o qual retorna uma referência remota da entidade armazenada no banco. Após obter a referência remota, o método `getConta` é invocado.

```
      public Conta procurar(String codigo) throws  RepositorioException  {
         Conta c = null;
           try {
            IContaRemote contaBean = getHome().findByPrimaryKey(codigo);
             c = contaBean.getConta();
          } catch (Exception e) {
                throw new RepositorioException(e);
          }
           return c;
       }
}
```

Este permite obter um clone do objeto `Conta`, o qual é retornando para o cliente. Isto permite que os clientes da aplicação manipulem cópias dos objetos remotos, em vez de manipulá-los remotamente, melhorando com isso, o desempenho, visto que invocação de métodos e manipulação destes remotamente são tarefas que degradam o desempenho do sistema.

Assim, a coleção de dados isola dos clientes das camadas acima, o acesso às referências remotas dos *entity beans*, uma vez que repassa para as camadas superiores os clones das classes básicas em vez de referências remotas. Isto pode trazer problemas de inconsistência dos dados acessados por clientes concorrentes. No entanto este aspecto pode ser facilmente solucionado com a introdução de mecanismos como *timestamp*.

A interface `IRepositorioConta` é a interface negócio-dados. Esta interface é implementada pela classe RepositorioDeContasEJB.

```
public interface IRepositorioConta {
    public Conta procurar(String numero)
        throws RepositorioException;
    public void atualizar(Conta conta)
        throws RepositorioException;
    public void inserir(Conta conta)
        throws RepositorioException;
    public Boolean existe (String numero)
        throws RepositorioException;
}
```

A classe coleção de negócio corresponde, no exemplo, à classe `CadastroDeContas`. Esta classe representa uma coleção de objetos da aplicação e fornece serviços para manipular um cadastro de contas. Esta classe utiliza os serviços da coleção de dados através da interface `IRepositorioConta` e seu código é apresentado a seguir.

```
public class CadastroDeContas {
  private IRepositorioConta repConta;

    public CadastroDeContas(IRepositorioConta rep) {
      repConta = rep;
    }
```

O construtor de `CadastroDeContas` recebe como argumento um objeto que implementa a interface negócio-dados. A partir do atributo `repConta`, a coleção de negócio invoca os métodos da coleção de dados.

Duas das operações para esta classe são apresentadas. O método `cadastrarConta` é utilizado para inserir um objeto `Conta` no sistema. Para tal, primeiramente é verificado se um objeto de mesmo número já existe, lançando a exceção `ContaJaExisteException` em caso positivo. Caso o objeto a ser cadastrado ainda não exista no sistema, o método da coleção de negócio invoca o método `inserir` da coleção de dados, através da interface negócio-dados.

```
    public void cadastrarConta(Conta conta)
             throws ContaJaExisteException, RepositorioException{
        if (repConta.existe(conta.getNumero()))
              throw new ContaJaExisteException();
        else
          repConta.inserir(conta);

    }
```

O método `creditar` consulta o sistema para uma determinada conta e, se a consulta for bem sucedida, um valor é adicionado ao saldo da conta e as informações são atualizadas na coleção de dados. Todavia, se a conta não existe, uma exceção é lançada.

```
   public void creditar(String numero,double valor)
            throws ContaNaoExisteException, RepositorioException {
      if (repConta.existe(numero)){
          Conta c = repConta.procurar(numero);
          c.creditar(valor);
          repConta.atualiza(c);
       }
          else throw new ContaNaoExisteException();
       }
  ...
}
```

A implementação dos demais métodos da coleção de negócio é feita de forma similar, e são omitidos aqui por questões de brevidade.

A classe `Banco` (fachada do sistema) contém todos os serviços oferecidos pela aplicação. Esta classe possui uma referência à classe `CadastroDeContas`. O construtor cria um objeto do tipo coleção de negócio e o atribui ao atributo `cadConta`. A partir deste ponto, a fachada pode invocar os métodos da coleção de negócio.

```
class Banco {
  private CadastroDeContas cadConta;
      Banco(){
        cadConta =
        new CadastroDeContas(new RepositorioDeContasEJB());
     }
```

Dois dos métodos da fachada são apresentados. O método `cadastrarConta` invoca o método `cadastrarConta` da coleção de negócio.

```
   void cadastrarConta(Conta conta)
     throws RepositorioException, ContaJaExisteException {
       cadConta.cadastrarConta(conta);
 }
```

O mesmo acontece com o método `creditar`

```
  void creditar(String numero, double saldo)
    throws RepositorioException, ContaNaoExisteException {

      cadConta.creditar(numero, saldo);
}
  ...
}
```

que também utiliza o atributo `cadConta` para invocar métodos da coleção de negócio, no caso, o método de mesmo nome `creditar`. As exceções `ContaJaExisteException` e `ContaNaoExisteException` são exceções específicas da aplicação.

Usos Conhecidos

Como parte do padrão DAP-EJB, o PDC-EJB é também utilizado nos mesmos sistemas que este.

Para o sistema de informação do serviço público de saúde, o PDC-EJB é utilizado tal como descrito nesta seção. Esta aplicação tem como funcionalidade, receber e controlar as denúncias, notificações, além de fornecer informações importantes sobre o sistema público de saúde, que sejam do interesse da população.

O PDC-EJB é também utilizado no sistema que fornece serviços para o gerenciamento de contabilidade, controle de acesso e serviços financeiros. Neste sistema, a coleção de negócio da aplicação é opcional em algumas partes da aplicação. Nos casos onde a coleção de negócio é opcional, o adaptador destino, acessa a coleção de dados, através da interface negócio-dados do sistema.

Outros possíveis usos do PDC-EJB:

- Um sistema para gerenciar clientes de uma empresa de telecomunicação. O sistema é capaz de registrar telefones móveis, gerenciar informações de clientes e a configuração dos serviços de telefonia. Este sistema pode ser utilizado via *Web*.

- Um sistema para provas interativas. Este sistema tem sido utilizado para fornecer diferentes tipos de provas, tais como simulados baseados em exames de seleção para a universidade, ajudando os alunos a avaliar seus conhecimentos antes de realizarem exames reais.

- Um sistema de supermercado complexo. Este sistema será usado em vários supermercados e já está sendo utilizado em outras empresas do mesmo ramo.

Padrões Relacionados

- *Persistent Data Collections* (PDC) [11]. Este padrão também foi adaptado para o padrão apresentado nesta seção. O PDC promove código modular fornecendo um conjunto de classes e interfaces que separam o código de acesso a dados do código de negócio e interface com o usuário [11]. O padrão para EJB também observa tais aspectos com a diferença que não faz uso de uma classe auxiliar (MecanismoDePersistencia) para gerenciamento das transações e conexão com banco, como faz o PDC. Além disso, a classe básica é associada a um *entity bean*. Neste padrão, a primeira possui métodos `get` e `set` e métodos de negócio, enquanto a segunda possui somente métodos de acesso ao banco e métodos que permitem obter clones dos objetos das classes básicas associadas.

- Value Object [6]. É similar à classe básica apresentada no PDC-EJB. Este padrão tem como função encapsular os dados do negócio, ou seja, em vez de um cliente fazer várias requisições remotas a métodos `get` e `set` para obter os dados da entidade, ele o faz através de um único método que é utilizado para encapsular todos os dados necessários à requisição do cliente.

  Apesar de fornecer um mecanismo que melhora consideravelmente o desempenho do sistema, uma vez que evita o fluxo de chamadas remotas à métodos `get` e `set`, os métodos de negócio continuam sendo executados remotamente, uma vez que permanecem sendo declarados na interface remota do *bean*. Além disso, o método `ejbCreate` recebe os atributos do bean como parâmetro e não o objeto correspondente à sua classe básica, como sugere o padrão apresentado aqui.

- *Facade* [7]. A classe `Fachada` do PDC-EJB é a implementação direta do padrão *Facade*.

- *Singleton* [7]. Por questões de eficiência, os objetos da classe `Fachada` são implementados como *Singleton*. Desta forma, geralmente somente um objeto fachada é requerido na aplicação.

- *Bridge* [7]. A interface negócio-dados do padrão PDC-EJB é implementada como *Bridge*. Desta forma, ela é utilizada para permitir a comunicação entre as camadas de negócio e dados mantendo a primeira isolada da API de persistência da aplicação.

Agradecimentos

Nossos agradecinmentos especiais a Márcio Barros, nosso *shepherd*, pelos comentários e sugestões importantes que proporcionaram melhorias no nosso padrão.

Referências

[1] Vander Alves. Desenvolvimento Progressivo de Programas Distribuídos Orientados a Objetos. Master's thesis, Centro de Informática – Universidade Federal de Pernambuco, Fevereiro 2001.

[2] Vander Alves and Paulo Borba. Distributed Adapters Pattern: A Design Pattern for Object–Oriented Distributed Applications. In *First Latin American Conference on Pattern Languages Programming, Sugarloaf PLoP*, Rio de Janeiro, Brazil, 3th–5th October 2001.

[3] Grady Booch et al. *The Unified Modeling Language User Guide*. Object Technology. Addison-Wesley, first edition, 1999.

[4] Paulo Borba, Saulo Araújo, Hednilson Bezerra, Marconi Lima, and Sérgio Soares. Progressive implementation of distributed Java applications. In *Engineering Distributed Objects Workshop, ACM International Conference on Software Engineering*, pages 40–47, Los Angeles, USA, 17th–18th May 1999.

[5] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern–Oriented Software Architecture:A System of Patterns, volume 1*. John Wiley & Sons, 1996.

[6] John Crupi DeepPAK Alur and Dan Malks. *core J2EE Patterns –Best Practices and Design Strategies*. Prentice Hall, first edition, 2001.

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object–Oriented Software*. Addison–Wesley, 1994.

[8] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison–Wesley, second edition, June 2000.

[9] Marty Hall. *Core Servlets and JavaServer Pages*. Prentice-Hall, second edition, 2000.

[10] Rosanna Lee and Scott Seligman. *JNDI API Tutorial and Reference: Building Directory-Enabled Java(TM) Applications*. Addison-Wesley, 2000.

[11] Tiago Massoni, Vander Alves, Sérgio Soares, and Paulo Borba. PDC: Persistent Data Collections pattern. In *First Latin American Conference on Pattern Languages Programming, Sugarloaf PLoP*, Rio de Janeiro, Brazil, 3th–5th October 2001.

[12] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern–Oriented Software Architecture: Patterns for Concurrent and Networked Objects, volume 2*. John Wiley & Sons, 2000.

[13] Sérgio Soares. Desenvolvimento Progressivo de Programas Concorrentes Orientados a Objetos. Master's thesis, Centro de Informática – Universidade Federal de Pernambuco, Fevereiro 2001.

[14] Sun Microsystems. The Enterprise JavaBeans 1.1 Specification. Disponível em http://-java.sun.com/products/ejb/docs.html, Outubro 2000.

[15] Seth White, Maydene Fisher, Rick Cattell, Graham Hamilton, and Mark Hapner. *JDBC(tm) API Tutorial and Reference: Universal Data Access for the Java(tm) 2 Platform*. Addison–Wesley, second edition, June 1999.