UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO – BACHARELADO

FRAMEWORK DE PERSISTÊNCIA DE DADOS COM CRIPTOGRAFIA DE CHAVE ASSIMÉTRICA UTILIZANDO HIBERNATE

JEAN CARLOS PEREIRA

BLUMENAU
2007

2007/2-20

JEAN CARLOS PEREIRA

FRAMEWORK DE PERSISTÊNCIA DE DADOS COM CRIPTOGRAFIA DE CHAVE ASSIMÉTRICA UTILIZANDO HIBERNATE

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciências da Computação — Bacharelado.

Prof. Alexander Roberto Valdameri, Mestre - Orientador

BLUMENAU

2007

2007/2-20

FRAMEWORK DE PERSISTÊNCIA DE DADOS COM CRIPTOGRAFIA DE CHAVE ASSIMÉTRICA UTILIZANDO HIBERNATE

Por

JEAN CARLOS PEREIRA

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

Presidente:	Prof. Alexander Roberto Valdameri, Mestre – Orientador, FURB
Membro:	Prof. Paulo Fernando da Silva, Mestre – FURB
Membro:	Prof. Francisco Adell Pericas, Mestre – FURB

Blumenau, 22 de novembro de 2007

Dedico este trabalho a todas as pessoas que me apoiaram durante a realização do mesmo, especialmente, minha noiva Daiana.

AGRADECIMENTOS

A minha família, que sempre me apoiou.

À minha noiva Daiana Ferreira, que soube me apoiar e compreender todos os momentos difíceis.

Aos meus amigos, por todo o apoio.

Ao meu orientador, Alexander Roberto Valdameri, por ter acreditado na conclusão deste trabalho.

Sofremos demasiado pelo pouco que nos falta
e alegramo-nos pelo muito que temos.

Wilian Shakespeare

RESUMO

Este trabalho apresenta o desenvolvimento de um *framework* de persistência de dados com suporte a criptografia de chave assimétrica. Através do recurso de *annotations* é possível adicionar características como criptografia ou assinatura digital aos atributos utilizando qualquer algoritmo fornecido pela aplicação. Para tanto, é utilizado o *framework* de persistência objeto-relacional Hibernate e a *Application Programming Interface* (API) *Bouncy Castle*. Por fim é mostrado um estudo de caso utilizando o *framework* desenvolvido.

Palavras-chave: Criptografia. Persistência de dados. Chave assimétrica. Hibernate.

ABSTRACT

This work presents the development of a data persistence framework with support to assimetric key cryptography. Through the annotations resource it is possible to add characteristics like cryptography or digital signature to attributes using any algorithm delivered by the application. For this, is used the object-relational persistence framework Hibernate e a Application Programming Interface (API) Bouncy Castle. Finally, is showed a study case using the developed framework.

Key-words: Criptography. Data persistence. Assimetric key. Hibernate.

LISTA DE ILUSTRAÇÕES

Quadro 1 – Bancos de dados suportados pelo Hibernate 24
Figura 1 – Arquitetura Hibernate 25
Figura 2 – Diagrama de casos de uso 31
Quadro 2 – Detalhamento do caso de uso `Criptografia de atributos` 32
Quadro 3 – Detalhamento do caso de uso `Decriptografia de atributos` 32
Quadro 4 – Detalhamento do caso de uso `Assinatura digital de atributos` 33
Quadro 5 – Detalhamento do caso de uso `Verificação da assinatura digital de atributos` 33
Figura 3 – Diagrama de atividades da persistência de objetos 34
Figura 4 – Diagrama de atividades da materialização de objetos 35
Figura 5 – Diagrama de classes do framework 36
Figura 6 – Diagrama de pacotes. 37
Quadro 6 – Classes do pacote `interceptor` 38
Quadro 7 – Classes do pacote `provider` 38
Figura 7 – Diagrama de seqüência da persistência dos objetos 39
Figura 8 – Diagrama de seqüência da materialização dos objetos 40
Quadro 8 – Anotação `SignedField`. 41
Figura 9 – Diagrama de classes da aplicação 43
Figura 10 – Modelo entidade-relacionamento da aplicação 43
Quadro 9 – arquivo hibernate.properties. 44
Quadro 10 –Implementação da classe MedicalEncryptionProvider. 44
Quadro 11 –Implementação da classe MedicalSignatureProvider. 45
Quadro 12 – Declaração de um atributo assinado digitalmente 46
Quadro 13 – Alteração no configurador do Hibernate 46
Figura 11 – Tela de login 46
Figura 12 – Tela utilizada pelo médico para consultar dados dos pacientes 47
Figura 13 – Tela de registro de históricos médicos 47
Quadro 14 – Classe `br.com.medical.model.History` 48
Figura 14 – Consulta a tabela HISTORY 48
Figura 15 – Consulta do histórico do paciente 48
Quadro 15 – Mapeamento da classe History 54

Quadro 16 – Mapeamento da classe Person 55
Quadro 17 – Mapeamento da classe User 56

LISTA DE SIGLAS

AC – Autoridade Certificadora

AES – *Advanced Encryption Standard*

ANSI – *American National Standards Institute*

API – *Application Programming Interface*

BCC – Curso de Ciências da Computação – Bacharelado

DCL – *Data Control Language*

DDL – *Data Definition Language*

DES – *Data Encryption Standard*

DML – *Data Manipulation Language*

DQL – *Data Query Language*

DSC – Departamento de Sistemas e Computação

EA – *Enterprise Architect*

HQL – *Hibernate Query Language*

ICP – Infraestrutura de Chaves Públicas

IETF – *Internet Engineering Task Force*

ITU – *International Telecommunications Union*

JDBC – *Java DataBase Conectivity*

JEE – *Java Enterprise Edition*

JSE – *Java Standard Edition*

JMS – *Java Message System*

JTA – *Java Transaction Api*

OO – Orientação a Objetos

OQL – *Object Query language*

ORM – *Object Ralational Mapping*

PBE – *Password Based Encryption*

PGP – Pretty Good Privacy

RF – Requisito Funcional

RNF – Requisito Não Funcional

SBC – Sociedade Brasileira de Computação

SEQUEL – *Structured English Query Language*

SGBD – Sistema Gerenciador de Banco de Dados

SQL – *Structured Query Language*

UML – *Unified Modeling Language*

XML – *eXtensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO 15
1.1 OBJETIVOS DO TRABALHO 16
1.2 ESTRUTURA DO TRABALHO 16
2 FUNDAMENTAÇÃO TEÓRICA 17
2.1 BANCO DE DADOS 17
2.1.1 Modelo relacional 18
2.2 SQL 19
2.2.1 Subconjuntos da linguagem SQL 20
2.2.1.1 *Data Manipulation Language* (DML) 20
2.2.1.2 *Data Definition Language* (DDL) 20
2.2.1.3 *Data Control Language* (DCL) 21
2.2.1.4 *Data Query Language* (DQL) 21
2.3 JDBC 22
2.4 HIBERNATE 23
2.5 SEGURANÇA EM BANCO DE DADOS 26
2.6 CRIPTOGRAFIA 26
2.7 CERTIFICADO DIGITAL 27
2.8 SERVIÇOS DE SEGURANÇA DE DADOS 28
2.9 TRABALHOS CORRELATOS 29
3 DESENVOLVIMENTO DO FRAMEWORK 30
3.1 REQUISITOS DO FRAMEWORK 30
3.2 ESPECIFICAÇÃO DO FRAMEWORK 30
3.2.1 Diagrama de casos de uso 31
3.2.2 Diagrama de atividades 33
3.2.3 Diagrama de pacotes 37
3.3 IMPLEMENTAÇÃO DO FRAMEWORK 41
3.3.1 Técnicas e ferramentas utilizadas 41
3.3.2 Operacionalidade da implementação 42
3.4 RESULTADOS E DISCUSSÃO 49
4 CONCLUSÕES 50
4.1 EXTENSÕES 50

REFERÊNCIAS BIBLIOGRÁFICAS 51
APÊNDICE A – Mapeamento das classes para o modelo relacional 54

1 INTRODUÇÃO

Atualmente o custo do desenvolvimento de um sistema é composto não somente pela implementação do sistema em si, mas também por todos os componentes de software de infra-estrutura que compõem o produto final (SILVA JR., 2005).

Parte do tempo despendido no desenvolvimento destes componentes é empregado na busca pela segurança das informações manipuladas pelo sistema de informação. É necessário garantir a integridade das informações não somente de acessos adversos ao sistema, mas também de falhas de implementação ou do modelo de negócio que permitam o acesso não autorizado às informações restritas por usuários não autorizados.

Este objetivo pode ser alcançado nos vários níveis de interação do sistema. Um destes é o nível de persistência das informações, aonde as informações manipuladas pelo sistema são mantidas em repouso. As vantagens de assegurar a segurança neste nível são: sigilo, integridade, autenticação e não repúdio das informações (SCHWEBEL, 2005, p. 11).

Neste nível o *framework* de persistência Hibernate apresenta-se como uma solução para o mapeamento de estruturas em memória volátil para memória secundária, abstraindo da aplicação detalhes relacionados ao Sistema Gerenciador de Banco de Dados (SGBD), sem, no entanto, preocupar-se com questões relacionadas à segurança da informação.

O objetivo de apresentar uma solução que agregue as características necessárias para assegurar a segurança em nível de persistência de informações, com a flexibilidade apresentada pelo Hibernate leva ao desenvolvimento de um dialeto[1] de dados do Hibernate, que forneça transparência de segurança quanto à criptografia das informações representadas pelos objetos. Desta forma as aplicações que já utilizam o Hibernate como mecanismo de persistência, podem adicionar criptografia sem impactos na sua infra-estrutura, bastando para isto a alteração do dialeto utilizado nos arquivos de configuração.

---

[1] Um dialeto define qual SGBD está sendo utilizado pelo Hibernate e como os comandos *Object Query Language* (OQL) devem ser convertidos em *Structured Query Language* (SQL). Dialetos pré-configurados são dialetos desenvolvidos pelo fabricante do SGBD ou pelo próprio Hibernate e fornecidos com a distribuição padrão do Hibernate (HIBERNATE, 2006).

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é desenvolver um *framework* de persistência de dados que permita que as informações sejam mantidas criptografadas no sistema de armazenamento.

Os objetivos específicos do trabalho são:

a) criar um *framework* composto por um dialeto permitindo a utilização de criptografia sem grandes alterações no código de aplicações que já utilizam o Hibernate;
b) realizar as adaptações necessárias nos algoritmos de chave assimétrica para compatibilizar com as especificidades existentes no SGBD;
c) prover implementações de pelo menos dois dos principais algoritmos de criptografia mais utilizados comercialmente;
d) validar o dialeto através da implementação de uma aplicação de exemplo que utilize SGBD através do Hibernate.

1.2 ESTRUTURA DO TRABALHO

Este trabalho está dividido em quatro capítulos. O segundo deles apresenta os assuntos estudados durante o desenvolvimento deste trabalho. Nele, de forma geral, são discutidos paradigmas e conceitos que envolveram a concepção do trabalho.

No terceiro capítulo é apresentado o desenvolvimento do trabalho, iniciando pelos requisitos que o *framework* deve atender e seguido pela especificação do mesmo. Então é feita uma apresentação da implementação, incluindo um estudo de caso, e logo em seguida são discutidos os resultados do desenvolvimento.

Por fim, o quarto capítulo apresenta as conclusões e sugestões para extensões em trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

A implementação de um *framework* de persistência requer o estudo de vários conceitos, técnicas e algoritmos. Assim, neste capítulo são apresentados conceitos, técnicas e ferramentas utilizados na concepção deste trabalho, dentre eles: SGBD, Criptografia, Hibernate. Na última sessão são descritos alguns trabalhos correlatos.

2.1 BANCO DE DADOS

Um sistema de banco de dados é essencialmente um sistema computadorizado de arquivamento de registros. Muitos dos arquivos que tradicionalmente são guardados sob forma de papel podem ser guardados de forma conveniente em um banco de dados (DATE, 1985, p. 21). Um SGBD é definido por Leocácio (2005) como "o software responsável pelo gerenciamento (armazenamento e recuperação) dos dados no Banco de Dados".

Segurança, em SGBDs, refere-se à proteção do banco de dados contra acessos acidentais ou intencionais utilizando controles baseados em computador ou não. Segurança pode ser traduzida em hardware, software e um conjunto de procedimentos técnicos e particulares que, juntos, ajudam a garantir que pessoas não autorizadas tenham acesso a informações que não podem ver.

São três objetivos principais que devem ser considerados ao projetar uma aplicação de banco de dados, no que se refere à segurança (LOURENÇO, 2007, p. 1):

a) confidencialidade: a informação não deve ser disponibilizada a usuários não autorizados;
b) integridade: apenas usuários autorizados podem modificar os dados;
c) disponibilidade: os usuários autorizados a acessar os dados não devem ter seu acesso negado.

A maneira mais prática de classificar bancos de dados é de acordo com a forma que seus dados são vistos pelo usuário, ou seja, seu modelo de dados (BAPTISTA, [2006?], p. 2).

Atualmente, a classificação mais comum citaria 4 modelos básicos:

a) modelos navegacionais, divididos em:
   -modelo hierárquico,

-modelo em redes;

b) modelo relacional;
c) modelo orientado a objetos;
d) modelo de entidades e relacionamentos;
e) modelo de lista invertida;
f) modelo relacional estendido;
g) modelo semi-estruturado.

2.1.1 Modelo relacional

O modelo relacional é uma teoria matemática desenvolvida por Edgar Frank Codd para descrever como as bases de dados devem funcionar. Embora esta teoria seja a base para o software de bases de dados relacionais, muito poucos sistemas de gestão de bases de dados seguem o modelo de forma restrita, e todos têm funcionalidades que violam a teoria, desta forma variando a complexidade e o poder (COOD, 1990, p. 70).

De acordo com a arquitetura ANSI em três níveis, os Bancos de Dados relacionais consistem de três componentes:

a) uma coleção de estruturas de dados, formalmente chamadas de relações, ou informalmente tabelas, compondo o nível conceitual;
b) uma coleção dos operadores, a álgebra e o cálculo relacionais, que constituem a base da linguagem *Structured Query Language* (SQL);
c) uma coleção de restrições da integridade, definindo o conjunto consistente de estados de base de dados e de alterações de estados. As restrições de integridade podem ser de quatro tipos:
   - domínio (ou tipo de dados),
   - atributo,
   - variável de relação,
   - restrições de base de dados.

Diferentemente dos modelos navegacionais, não existem quaisquer ponteiros, de acordo com o Princípio de Informação: toda informação tem de ser representada como dados; qualquer tipo de atributo representa relações entre conjuntos de dados.

Distintamente dos bancos de dados em rede, nos bancos de dados relacionais os relacionamentos entre as tabelas não são codificados explicitamente na sua definição. Em vez

disso, se fazem implicitamente pela presença de atributos chave. As bases de dados relacionais permitem aos utilizadores escreverem consultas, reorganizando e utilizando os dados de forma flexível e não necessariamente antecipada pelos projetistas originais. Esta flexibilidade é especialmente importante em bases de dados que podem ser utilizadas durante décadas, tornando as bases de dados relacionais muito populares no meio comercial.

Um dos pontos fortes do modelo relacional de banco de dados é a possibilidade de definição de um conjunto de restrições de integridade. Estas definem os conjuntos de estados e mudanças de estado consistentes do banco de dados, determinando os valores que podem e os que não podem ser armazenados (DATE, 1985, p. 54).

2.2 SQL

Segundo Date (1985, p. 71) "a SQL é uma linguagem padrão para se lidar com bancos de dados relacionais, e é aceita por quase todos os produtos existentes no mercado".

A SQL foi desenvolvida originalmente no início dos anos 70 nos laboratórios da IBM em San Jose, dentro do projeto System R, que tinha por objetivo demonstrar a viabilidade da implementação do modelo relacional proposto por Edgar. F. Codd. O nome original da linguagem era *Structured English Query Language* (SEQUEL) (CHAMBERLIN, 1977 p. 71).

A linguagem SQL é um grande padrão de banco de dados. Isto decorre da sua simplicidade e facilidade de uso. Ela se diferencia de outras linguagens de consulta a banco de dados no sentido em que uma consulta SQL especifica a forma do resultado e não o caminho para chegar a ele. Ela é um linguagem declarativa em oposição a outras linguagens procedurais. Isto reduz o ciclo de aprendizado daqueles que se iniciam na linguagem.

Embora a SQL tenha sido originalmente criada pela IBM, rapidamente surgiram vários dialetos desenvolvidos por outros produtores. Essa expansão levou à necessidade de ser criado e adaptado um padrão para a linguagem. Esta tarefa foi realizada pela *American National Standards Institute* (ANSI) em 1986 e ISO em 1987 (DATE, 1985 p. 71).

A SQL foi revista em 1992 e a esta versão foi dado o nome de SQL-92. Foi revisto novamente em 1999 e 2003 para se tornar SQL:1999 (SQL3) e SQL:2003, respectivamente. O SQL:1999 usa expressões regulares de emparelhamento, queries recursivas e gatilhos. Também foi feita uma adição controversa de tipos não-escalados e algumas características de orientação a objetos (OO). O SQL:2003 introduz características relacionadas ao XML,

seqüências padronizadas e colunas com valores de auto-generalização (inclusive colunas-identidade) (DATE, 1985 p. 242).

Tal como dito anteriormente, a SQL, embora padronizada pela ANSI e ISO, possui muitas variações e extensões produzidas pelos diferentes fabricantes de sistemas gerenciadores de bases de dados. Tipicamente a linguagem pode ser migrada de plataforma para plataforma sem mudanças estruturais principais.

2.2.1 Subconjuntos da linguagem SQL

A linguagem SQL se divide em cinco principais grupos, descritos á seguir.

2.2.1.1 *Data Manipulation Language* (DML)

A DML é um subconjunto da linguagem SQL usada para selecionar, inserir, atualizar e apagar dados (DATE, 1985 p. 32). Os principais comandos deste subconjunto são:

a) `SELECT` é o comumente mais usado, comanda e permite ao usuário especificar uma query como uma descrição do resultado desejado;
b) `INSERT` é usada para somar uma fila (formalmente uma tupla) a uma tabela existente;
c) `UPDATE` para mudar os valores de dados em uma fila de tabela existente;
d) `DELETE` permite remover filas existentes de uma tabela.

2.2.1.2 *Data Definition Language* (DDL)

O subconjunto DDL permite ao usuário definir tabelas novas e elementos associados. A maioria dos bancos de dados de SQL comerciais tem extensões proprietárias no DDL (DATE, 1985 p. 32).

Os comandos básicos da DDL são:

a) `CREATE` cria um objeto (uma Tabela, por exemplo) dentro da base de dados;
b) `DROP` apaga um objeto do banco de dados;

c) `ALTER` permite ao usuário alterar um objeto, por exemplo, adicionando uma coluna a uma tabela existente.

2.2.1.3 *Data Control Language* (DCL)

A DCL controla os aspectos de autorização de dados e licenças de usuários para controlar quem tem acesso para ver ou manipular dados dentro do banco de dados (HIETEL, 2005).

As principais palavras-chaves da DCL:

a) `CREATE` cria um objeto (uma Tabela, por exemplo) dentro da base de dados;
b) `GRANT` - autoriza ao usuário executar ou definir operações;
c) `REVOKE` - remove ou restringe a capacidade de um usuário de executar operações;
d) `BEGIN WORK` (ou `START TRANSACTION`, dependendo do dialeto SQL) pode ser usado para marcar o começo de uma transação de banco de dados que pode ser completada ou não;
e) `COMMIT` torna todos os dados das mudanças permanentemente;
f) `ROLLBACK` faz com que as mudanças nos dados existentes desde que o último `COMMIT` ou `ROLLBACK` sejam descartadas.
g) `COMMIT` e `ROLLBACK` interagem com áreas de controle como transação e locação. Ambos terminam qualquer transação aberta e liberam qualquer cadeado ligado a dados. Na ausência de um `BEGIN WORK` ou uma declaração semelhante, a semântica de SQL é dependente da implementação.

Outros comandos DCL:

a) `ALTER PASSWORD`;
b) `CREATE SYNONYM`.

2.2.1.4 *Data Query Language* (DQL)

Embora tenha apenas um comando, a DQL é a parte da SQL mais utilizada. O comando SELECT é composto de várias cláusulas e opções, possibilitando elaborar consultas das mais simples as mais elaboradas.

2.3 JDBC

Java DataBase Conectivity (JDBC) é um padrão industrial para a conectividade independente de banco de dados entre a linguagem de programação Java e um grande número de banco de dados – bancos de dados SQL e outras fontes de dados tabulares, como tabelas ou arquivos planos. JDBC fornece uma API ao nível de chamada para acesso ao banco de dados baseado em SQL.

A tecnologia JDBC permite que o uso da linguagem de programação Java para explorar as potencialidades "Escreva uma vez, Execute em qualquer lugar" em aplicações que requeiram acesso aos dados corporativos. Com um *driver* com a tecnologia JDBC, é possível conectar todos os dados corporativos mesmo em um ambiente heterogêneo (SUN MICROSYSTEMS, 2006). Os drivers JDBC podem ser classificados como:

a) ponte JDBC-ODBC: é o tipo mais simples, mas restrito à plataforma Windows. Utiliza ODBC para conectar-se com o banco de dados, convertendo métodos JDBC em chamadas às funções do ODBC. Esta ponte é normalmente usada quando não há um driver puro-Java (tipo 4) para determinado banco de dados, pois seu uso é desencorajado devido à dependência de plataforma;
b) driver API-Nativo: traduz as chamadas JDBC para as chamadas da API cliente do banco de dados usado. Como a Ponte JDBC-ODBC, pode precisar de software extra instalado na máquina cliente;
c) driver de Protocolo de Rede: traduz a chamada JDBC para um protocolo de rede independente do banco de dados utilizado, que é traduzido para o protocolo do banco de dados por um servidor. Por utilizar um protocolo independente, pode conectar as aplicações clientes Java a vários bancos de dados diferentes. É o modelo mais flexível;
d) driver nativo: converte as chamadas JDBC diretamente no protocolo do banco de dados. Implementado em Java, normalmente é independente de plataforma e escrito pelos próprios desenvolvedores. É o tipo mais recomendado para ser usado.

2.4 HIBERNATE

O Hibernate é um *framework* de acesso a banco de dados escrito em Java. O objetivo do Hibernate é facilitar a construção de aplicações Java dependentes de bases de dados relacionais e particularmente, facilitar o desenvolvimento das consultas e atualizações dos dados. O uso de ferramentas de mapeamento objeto-relacional, como o Hibernate, diminuem a complexidade resultante da convivência de modelos diferentes: o modelo orientado a objetos (da linguagem Java) e o relacional (da maioria dos SGBDs) (HIBERNATE, 2006).

O Hibernate é um poderoso serviço de consultas e persistência objeto-relacional. Ele permite desenvolver classes persistentes seguindo o idioma orientado a objetos, incluindo associação, herança, polimorfismo, composição e coleções. Hibernate permite expressar consultas em sua própria extensão SQL, nomeada como *Hibernate Query Language* (HQL), bem como em SQL nativo, ou com critérios orientados a objetos (HIBERNATE, 2006).

A principal função do Hibernate é realizar *Object Relation Mapping* (ORM), sendo o nome dado às soluções em que é automatizado o mapeamento dos objetos manipulados pela aplicação em estruturas de bancos de dados relacionais. ORM realiza o mapeamento de objetos de um banco relacional para classes de linguagem Java. Segundo Bauer e King (2005, p. 58), o Hibernate define a associação entre a classe e a tabela do SGBD através de arquivos *eXtensible Markup Language* (XML) ou através de anotações específicas que somente são compatíveis com a versão 5.0 da *Java Standard Edition* (JSE).

É através desta associação que é possível converter expressões escritas em OQL para comandos JDBC que são compatíveis com o banco de dados escolhido, tornando uma mesma expressão OQL compatível com diferentes bancos de dados. Devido às especificidades de cada banco de dados, é através dos dialetos que é feita a tradução de uma instrução OQL para uma instrução SQL. O quadro 1 apresenta os bancos de dados atualmente suportados.

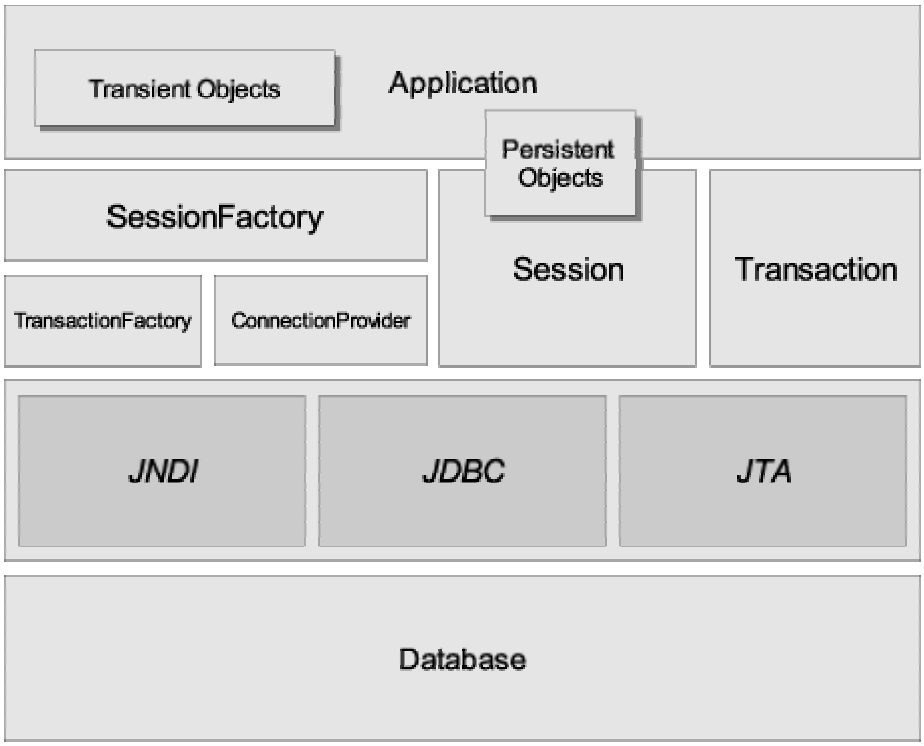
Banco de dados	Dialeto
DB2	`org.hibernate.dialect.DB2Dialect`
DB2 AS/400	`org.hibernate.dialect.DB2400Dialect`
DB2 OS390	`org.hibernate.dialect.DB2390Dialect`
PostgreSQL	`org.hibernate.dialect.PostgreSQLDialect`
MySQL	`org.hibernate.dialect.MySQLDialect`
MySQL com InnoDB	`org.hibernate.dialect.MySQLInnoDBDialect`
MySQL com MyISAM	`org.hibernate.dialect.MySQLMyISAMDialect`
Oracle	`org.hibernate.dialect.OracleDialect`
Oracle 9i/10g	`org.hibernate.dialect.Oracle9Dialect`
Sybase	`org.hibernate.dialect.SybaseDialect`
Sybase Anywhere	`org.hibernate.dialect.SybaseAnywhereDialect`
Microsoft SQL Server	`org.hibernate.dialect.SQLServerDialect`
SAP DB	`org.hibernate.dialect.SAPDBDialect`
Informix	`org.hibernate.dialect.InformixDialect`
HypersonicSQL	`org.hibernate.dialect.HSQLDialect`
Ingres	`org.hibernate.dialect.IngresDialect`
Progress	`org.hibernate.dialect.ProgressDialect`
Mckoi SQL	`org.hibernate.dialect.MckoiDialect`
Interbase	`org.hibernate.dialect.InterbaseDialect`
Pointbase	`org.hibernate.dialect.PointbaseDialect`
FrontBase	`org.hibernate.dialect.FrontbaseDialect`
Firebird	`org.hibernate.dialect.FirebirdDialect`

Quadro 1 – Bancos de dados suportados pelo Hibernate

O Hibernate também provê gerenciamento de estado dos objetos, permitindo que o desenvolvedor pense no estado dos objetos, sem necessariamente se preocupar com a execução de comandos SQL. Os seguintes estados são definidos e suportados:

a) transiente: um objeto é considerado transiente quando foi instanciado através do operador `new` e não foi associado a nenhuma sessão Hibernate, não há uma representação para este objeto na base de dados e não há um identificador definido para o mesmo;
b) persistente: é considerado persistente o objeto que possui uma representação na base de dados e possui um valor de identificador, ele está por definição no escopo da sessão Hibernate. O Hibernate irá detector quaisquer modificações feitas e irá sincronizar o estado com o banco de dados quando a unidade de trabalho terminar.
c) desassociado: Um objeto neste estado esteve em um estado persistente, mas a sessão em que estava associado foi fechada. A referência para o objeto continua válida, e pode ser modificada normalmente. O objeto pode ser associado a uma nova sessão em um momento posterior, tornando-o persistente novamente. Este estado permite um modelo de programação com transações longas, que requerem intervenção humana.

A arquitetura do Hibernate é descrita na figura 1.



Fonte: Hibernate (2007).

Figura 1 – Arquitetura Hibernate

Os objetos na figura podem ser definidos como:

a) `SessionFactory` (`org.hibernate.SessionFactory`): um *cache* imutável de mapeamentos compilados para uma base de dados única. É uma fábrica para `Session` e um cliente de `ConnectionProvider`. Pode manter um *cachê* de segundo nível (opcional) dos objetos que são reusáveis entre as transações;

b) `Session` (`org.hibernate.Session`): um objeto de vida curta representando a conversação entre a aplicação e o armazenamento persistente. Encapsula uma conexão JDBC e é uma fábrica para Transaction. Mantém um *cache* de primeiro nível para os objetos persistentes que é usado quando navegando pelo grafo de objetos ou ao procurar objetos pelo identificador;

c) `Transaction` (`org.hibernate.Transaction`): é um componente opcional, de vida curta, usado pela aplicação para garantir a atomicidade da unidade de trabalho. Abstrai a aplicação de detalhes de transações JDBC, JTA ou CORBA. Uma transação pode aninhar outras transações em muitos casos;

d) `ConnectionProvider` (`org.hibernate.connection.ConnectionProvider`): é uma fábrica e um *pool* de conexões JDBC. Abstrai da aplicação detalhes como o *datasource* ou o driver utilizado na conexão. Não é exposto para a aplicação mas pode ser estendido ou implementado pelo desenvolvedor;

e) `TransactionFactory` (`org.hibernate.TransactionFactory`): é uma fábrica para instâncias de `Transaction`. Não é exposto para a aplicação mas pode ser estendido ou implementado pelo desenvolvedor.

2.5 SEGURANÇA EM BANCO DE DADOS

De acordo com Tonahan (2006, p. 211), a segurança é a condição de ser protegido de encontro ao perigo ou à perda. No sentido geral, a segurança é um conceito similar à proteção. A nuance entre os dois é uma ênfase adicionada em ser protegido dos perigos que originam da parte externa. Os indivíduos ou as ações que transgridam em cima da condição da proteção são responsáveis pela ruptura da segurança.

Para o Banco Bradesco S.A. (2006), "A segurança da informação é um conjunto de medidas que se constituem basicamente de controles e política de segurança, tendo como objetivo a proteção das informações dos clientes e da empresa, controlando o risco de revelação ou alteração por pessoas não autorizadas."

A relação entre bancos de dados e segurança da informação pode ser definida como:

> A segurança de um banco de dados (BD) é baseada em modelos e métodos que procuram garantir a proteção de dados contra a divulgação, alteração ou destruição não autorizada. Em síntese é garantir que um usuário tenha, de fato, autorização para executar a operação que estiver tentando executar. Podemos portanto questionar os fatores que deverão ser analisados para obter esta segurança. (SILVA, 2006).

2.6 CRIPTOGRAFIA

A criptografia tem como objetivo tornar a informação sigilosa a um adversário que possa vir a interceptá-la. A palavra criptografia é formada por duas palavras gregas: *kryptós*, (oculto, secreto) e *gráphein* (escrita, escrever) (CARVALHO, 2001, p. 51).

Cifragem é o processo pelo qual a criptografia torna dados incompreensíveis, isto é, codificados ou secretos, normalmente chamados de texto cifrado. Já o processo inverso é a decifragem, em que dados são decodificados para possibilitar sua leitura e normalmente chamados de texto claro (CARVALHO, 2001, p. 52).

> Alguns processos de criptografia geram textos cifrados em que não há um processo

> inverso. Estes processos, denominados de mão única são conhecidos como **função para resumo de dados ou função hash**. Em outros processos, para cifrar e decifrar os dados se utiliza uma chave, da qual a segurança depende em grande parte. O número de bits de uma chave é um parâmetro fundamental no sistema de segurança, pois quanto maior for o número de bits da chave, maior é o número de possibilidades de chaves que podem ser utilizadas, assim dificultando que um ataque de força bruta descubra a chave. (SCHWEBEL, 2005, p. 41, grifo nosso).

Existem duas maneiras diferentes de cifrar a informação desejada. A primeira maneira é através da utilização de chaves simétricas, também conhecidas como chaves secretas, "[...] onde o emissor e o receptor fazem uso da mesma chave, isto é, uma única chave é usada na codificação e na decodificação da informação." (ALECRIM, 2005). Isto exige que o emissor e receptor conheçam a chave usada.

A segunda maneira é através da utilização de chaves assimétricas, ou chaves públicas, onde:

> "[...] a chave assimétrica trabalha com duas chaves: uma denominada privada e outra denominada pública. Nesse método, uma pessoa deve criar uma chave de codificação e enviá-la a quem for mandar informações a ela. Essa é a chave pública. Uma outra chave deve ser criada para a decodificação. Esta - a chave privada - é secreta." (ALECRIM, 2005).

Este trabalho não apresenta o estudo de algoritmos de criptografia e mecanismos de certificação digital.

2.7 CERTIFICADO DIGITAL

Um certificado digital é um arquivo de computador que contém um conjunto de informações referentes a entidade para o qual o certificado foi emitido (seja uma empresa, pessoa física ou computador) mais a chave pública referente a chave privada que acredita-se ser de posse unicamente da entidade especificada no certificado.

Normalmente é usado para ligar uma entidade a uma chave pública. Para garantir a integridade das informações contidas neste arquivo ele é assinado digitalmente, no caso de uma Infraestrutura de Chaves Públicas (ICP), o certificado é assinado pela Autoridade Certificadora (AC) que o emitiu e no caso de um modelo de teia de confiança como o *Pretty Good Privacy* (PGP) o certificado é assinado pela própria entidade e assinado por outros que dizem confiar naquela entidade. Em ambos os casos as assinaturas contidas em um certificado são atestamentos feitos por uma entidade que diz confiar nos dados contidos naquele certificado.

O padrão mais comum para certificados digitais no âmbito de uma ICP é o ITU-T X.509. O X.509 foi adaptado para a Internet pelo grupo da *Internet Engineering Task Force* (IETF) PKIX (STALLINGS 2003, p. 420)

2.8 SERVIÇOS DE SEGURANÇA DE DADOS

Serviços de segurança são usados para garantir objetivos da segurança de dados (MAIWALD, 2001, p. 278). Um serviço de segurança é caracterizado por um conjunto de mecanismos, procedimentos e outros controles, atendendo a um determinado objetivo de segurança. Meneses, Oorschot e Vanstone (1997) e Burnett e Paine (2002, p. 48) enfatizam os seguintes serviços de segurança:

a) sigilo – utilizado para manter o conteúdo dos dados conhecidos somente por aqueles que são autorizados. É sinônimo de confidencialidade e privacidade;
b) integridade – é um mecanismo que identifica a ocorrência de uma manipulação de dados não autorizada. A manipulação de dados inclui formas como inclusão, exclusão e alteração;
c) autenticação – está relacionada com a identificação e muitas vezes dividida em categorias distintas: autenticação de entidade, ou seja, se duas entidades estão se comunicando, cada uma pode ser identificada; e a autenticação da origem dos dados, que proporciona implicitamente a integridade dos dados, por exemplo, através de um resumo de dados;
d) não-repúdio – permite que uma determinada entidade não negue um compromisso ou ação já efetuada por ela. Em outras palavras, é uma imposição legal que orienta e impele as pessoas a honrar suas palavras.

Além dos serviços acima citados, a recomendação X.800 da *International Telecommunications Union* (ITU) acrescenta o serviço de controle de acesso (INTERNATIONAL TELECOMMUNICATIONS UNION, 1991). Este serviço define a prevenção do uso desautorizado de algum recurso, controlando quem pode acessar o recurso, sob que condições o acesso pode ocorrer e o que aqueles que acessam o recurso têm permissão para fazer (STALLINGS, 2003, p. 174).

2.9 TRABALHOS CORRELATOS

Alguns SGBDs, como o Empress, fornecem tipos de dados específicos para criptografia, porém restringem a portabilidade da aplicação (HOLOVAC, 2006).

Schwebel (2005) desenvolveu um *framework* orientado a objetos para a segurança de dados em repouso chamado Frasedare. O Frasedare é composto de componentes para ambiente integrado de desenvolvimento Delphi 5 da Borland que aplicam criptografia e gerenciam o controle de chaves. O Frasedare foi validado em uma aplicação médica, assegurando que as informações do paciente só podem ser lidas pelo mesmo, ou pelo profissional da área de saúde que o atendeu, no entanto, diversas modificações foram necessárias para a utilização do Frasedare neste sistema.

Gianisini Jr. (2006) propôs uma solução de replicação de dados utilizando o Hibernate e filas *Java Messaging System* (JMS) para gerenciar SGBDs Oracle e Microsoft SQL Server. Esta aplicação utiliza-se de uma infra-estrutura de internet, sem a utilização de certificados digitais, para replicar dados em lote entre filiais e uma matriz.

3 DESENVOLVIMENTO DO FRAMEWORK

O desenvolvimento do *framework* foi fundamentado nos estudos realizados no capítulo anterior. Este capítulo apresenta os requisitos, a especificação e a implementação do mesmo.

3.1 REQUISITOS DO FRAMEWORK

O *framework* desenvolvido deve atender os seguintes requisitos:

a) permitir a utilização do *framework* Hibernate.
b) suportar a utilização de algoritmos de criptografia fornecidos pela aplicação que utiliza o *framework* através de interfaces (RF);
c) prover uma implementação padrão do algoritmo RSA (RF);
d) garantir que um objeto transiente seja persistido com os atributos criptografados (RF);
e) garantir que um objeto persistido seja materializado com os atributos decriptografados (RF);
f) prover o recurso de *annotations* da linguagem Java 5 como meio de definição para atributos (RNF);
g) permitir que um atributo seja assinado digitalmente (RF);
h) garantir a autenticidade da assinatura digital em um atributo que foi assinado (RF);
i) prover uma implementação padrão de assinatura digital abstraindo o par de chaves (RF);
j) permitir que seja definido um atributo diferente para a persistência do valor cifrado ou da assinatura digital (RF);
k) ser implementado em Java utilizando o ambiente de programação Eclipse (RNF).

3.2 ESPECIFICAÇÃO DO FRAMEWORK

Para facilitar o entendimento da especificação, foram utilizados modelos da *Unified*

Modeling Language (UML) para a especificação dos diagramas de casos de uso, atividades, classes, pacotes e seqüência. A ferramenta utilizada para especificar os diagramas foi o Enterprise Architect (EA). Visando melhorar a leitura dos diagramas de classe, sem comprometer o conteúdo, algumas informações dos diagramas foram suprimidas, tais como métodos e atributos. Ainda, os diagramas foram divididos por atividades ou assunto, de modo a fornecer diagramas menores e mais legíveis.

3.2.1 Diagrama de casos de uso

O *framework* consiste em quatro casos de uso onde o desenvolvedor pode interagir. Os casos de uso contemplam todos os requisitos especificados. A figura 1 apresenta o diagrama de casos de uso, juntamente com os requisitos contemplados.

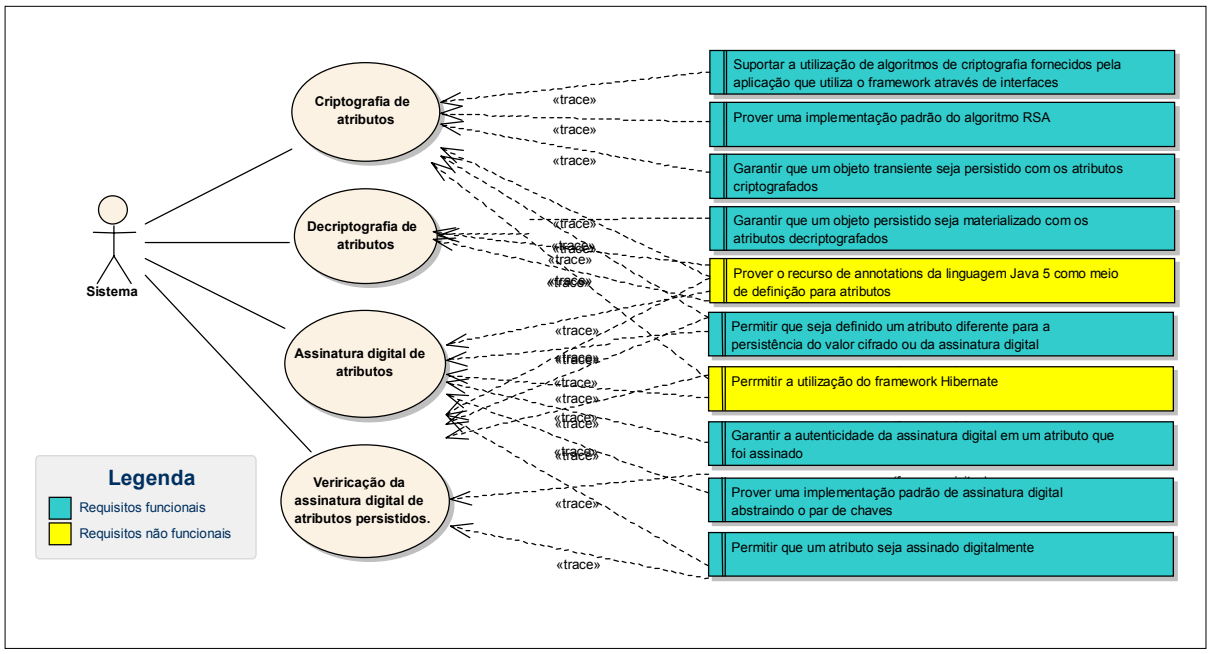


Figura 2 – Diagrama de casos de uso

Os casos de uso apresentados na figura 2 são detalhados nos quadros 2 a 5.

Criptografia de atributos	
Descrição	Realiza a criptografia nos atributos de um objeto que será persistido.
Pré-condições	Uma anotação deve ser definida no atributo indicando que o mesmo deve ser criptografado.
Fluxo principal	1. O sistema cria um objeto. 2. O sistema define os valores dos atributos no objeto. 3. O sistema salva o objeto através de uma sessão Hibernate. 4. O framework intercepta a operação de salvamento do objeto. 5. O framework verifica através do meta dados da classe quais atributos do objeto possuem uma anotação indicando que o campo deve ser criptogrado. 6. O framework solicita ao sistema um chave secreta ou uma chave pública e o algoritmo desejado. 7. O framework cifra cada um dos atributos que possuem a anotação e redefine o valor do atributo com a informação cifrada. 8. O hibernate persiste o objeto com os atributos cifrados.
Fluxo alternativo	No passo 7, se for informado na anotação um atributo diferente para persistir a informação, este atributo deve ser utilizado.
Fluxo de exceção	No passo 7, caso ocorra algum erro no processo de criptografia o erro será repassado para a aplicação.
Pós-condições	Os campos na base de dados relacionados aos atributos que possuem a anotação irão conter uma representação cifrada da informação original.

Quadro 2 – Detalhamento do caso de uso `Criptografia de atributos`

Decriptografia de atributos	
Descrição	Realiza a decriptografia nos atributos de um objeto que foi persistido.
Pré-condições	Uma anotação deve ser definida no atributo indicando que o mesmo deve ser criptografado.
Fluxo principal	1. O sistema solicita a materialização de um objeto através da sessão Hibernate. 2. O framework intercepta a operação de materialização do objeto. 3. O framework verifica através do meta dados da classe quais atributos do objeto possuem uma anotação indicando que o campo deve ser criptogrado. 4. O framework solicita ao sistema um chave secreta ou uma chave privada e o algoritmo desejado. 5. O framework decripta cada um dos atributos que possuem a anotação e redefine o valor do atributo com a informação original. 6. O hibernate retorna o objeto para o sistema.
Fluxo alternativo	No passo 7, se for informado na anotação um atributo diferente para persistir a informação, a informação cifrada será obtida através deste atributo.
Fluxo de exceção	No passo 7, caso ocorra algum erro no processo de decriptografia o erro será repassado para a aplicação.
Pós-condições	O objeto materializado terá o valor dos atributos que contém a anotação em sua representação original, decriptografado.

Quadro 3 – Detalhamento do caso de uso `Decriptografia de atributos`

Assinatura digital de atributos	
Descrição	Assina digitalmente atributos de um objeto que será persistido.
Pré-condições	Uma anotação deve ser definida no atributo indicando que o mesmo deve ser assinado digitalmente.
Fluxo principal	1. O sistema cria um objeto. 2. O sistema define os valores dos atributos no objeto. 3. O sistema salva o objeto através de uma sessão Hibernate. 4. O framework intercepta a operação de salvamento do objeto. 5. O framework verifica através do meta dados quais atributos da classe do objeto possuem uma anotação indicando que o campo deve ser assinado digitalmente. 6. O framework solicita ao sistema um chave privada e o algoritmo desejado. 7. O framework assina digitalmente cada um dos atributos que possuem a anotação e redefine o valor do atributo com o valor assinado. 8. O hibernate persiste o objeto com os atributos assinados.
Fluxo alternativo	No passo 7, se for informado na anotação um atributo diferente para persistir a assinatura digital, este atributo deve ser utilizado para persistir a assinatura. O valor permanece inalterado.
Fluxo de exceção	No passo 7, caso ocorra algum erro no processo de assinatura o erro será repassado para a aplicação.
Pós-condições	Os campos na base de dados relacionados aos atributos que possuem a anotação irão conter uma assinatura digital e uma informação original.

Quadro 4 – Detalhamento do caso de uso `Assinatura digital de atributos`

Verificação da assinatura digital de atributos	
Descrição	Verifica a validade da assinatura digital nos atributos de um objeto que foi persistido.
Pré-condições	Uma anotação deve ser definida no atributo indicando que o mesmo deve ser assinado digitalmente.
Fluxo principal	1. O sistema solicita a materialização de um objeto através da sessão Hibernate. 2. O framework intercepta a operação de materialização do objeto. 3. O framework verifica através do meta dados da classe quais atributos do objeto possuem uma anotação indicando que o campo deve ser assinado digitalmente. 4. O framework solicita ao sistema uma chave pública e o algoritmo desejado. 5. O framework verifica o texto assinado digitalmente e redefine o valor do atributo com a informação original. 6. O hibernate retorna o objeto para o sistema.
Fluxo alternativo	No passo 7, se for informado na anotação um atributo diferente para persistir a informação, este atributo deve ser utilizado.
Fluxo de exceção	No passo 7, caso ocorra algum erro no processo de criptografia o erro será repassado para a aplicação.
Pós-condições	Os campos na base de dados relacionados aos atributos que possuem a anotação irão conter uma representação cifrada da informação original.

Quadro 5 – Detalhamento do caso de uso `Verificação da assinatura digital de atributos`

3.2.2 Diagrama de atividades

As atividades executadas pelos casos de uso Criptografia de atributos e Assinatura digital de atributos são detalhadas no diagrama de atividades apresentado na figura 3. Cada

atividade representa uma etapa do processo.

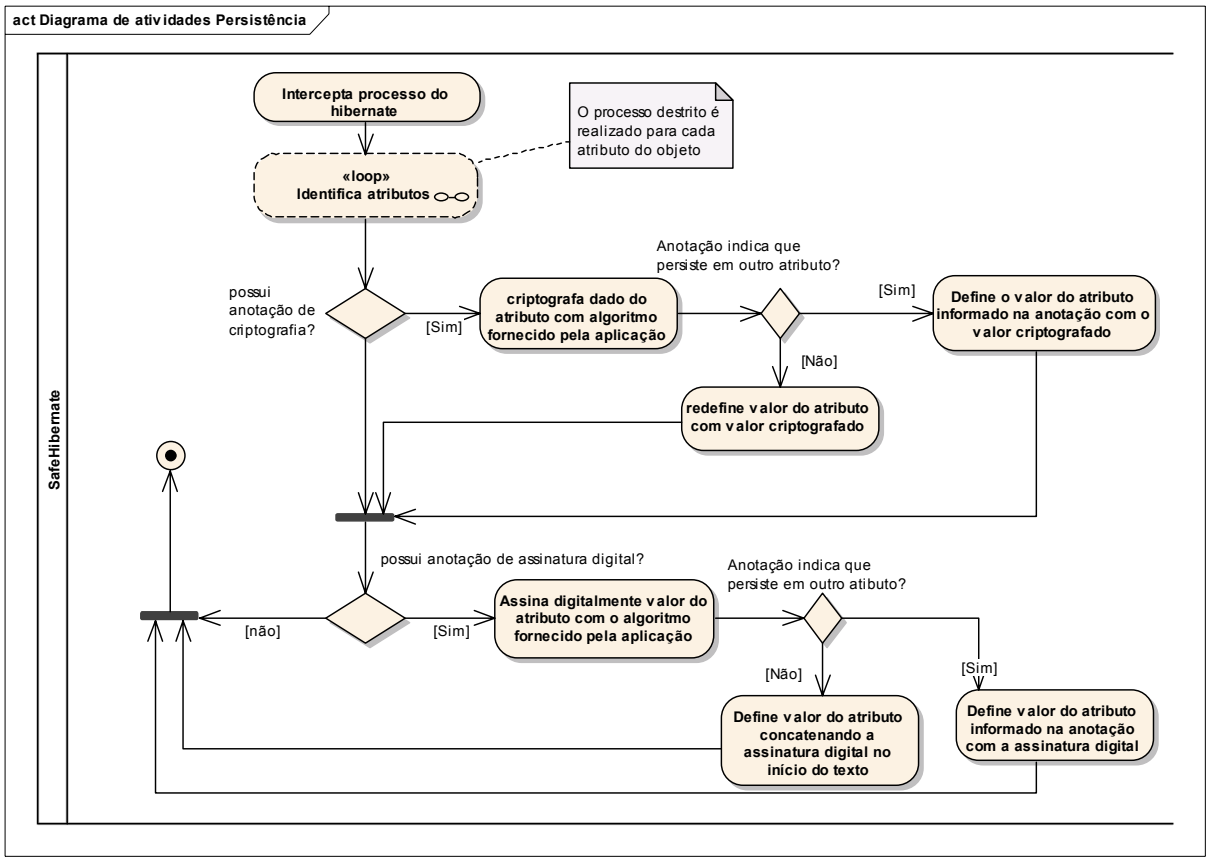


Figura 3 – Diagrama de atividades da persistência de objetos

As atividades executadas pelos casos de uso `Decriptografia de atributos` e `Verificação de assinatura digital de atributos` são detalhados no diagrama de atividades apresentado na figura 4. Cada atividade representa uma etapa do processo.

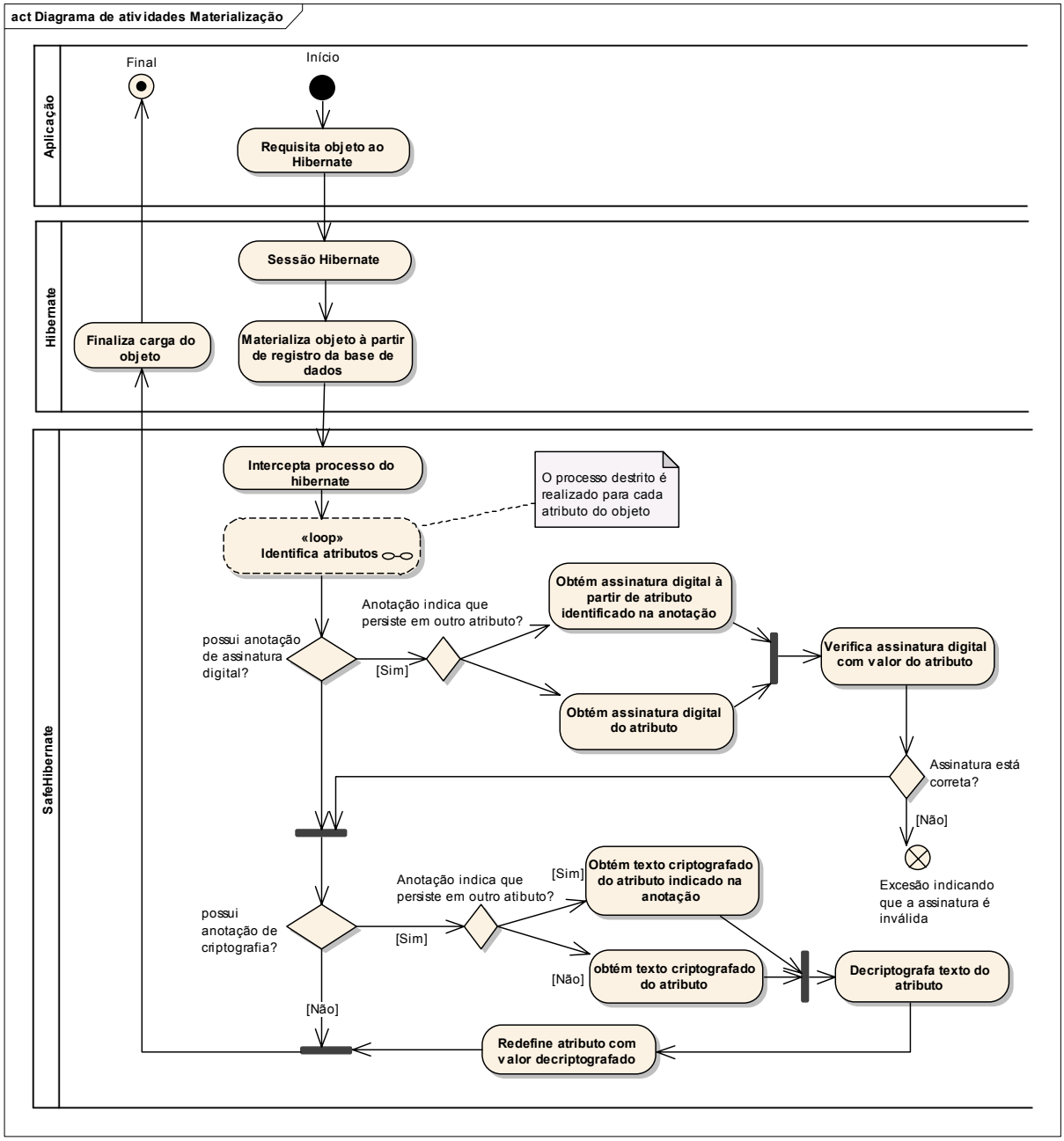


Figura 4 – Diagrama de atividades da materialização de objetos

O diagrama de classes do framework é apresentado na figura 5.

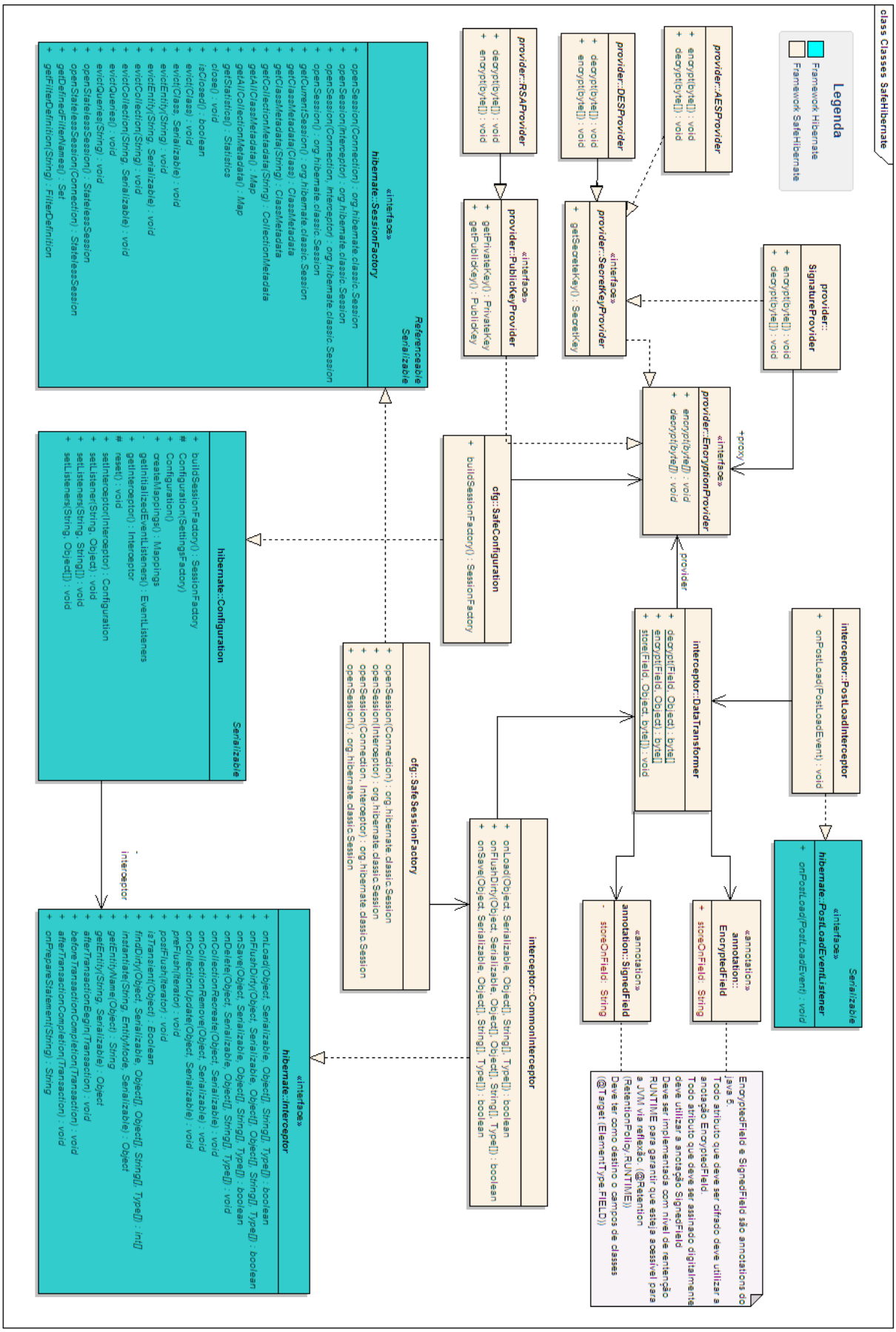


Figura 5 – Diagrama de classes do framework

As classes exibidas na figura 5 serão detalhadas na próxima sessão.

3.2.3 Diagrama de pacotes

A distribuição de pacotes é descrita na figura 6.

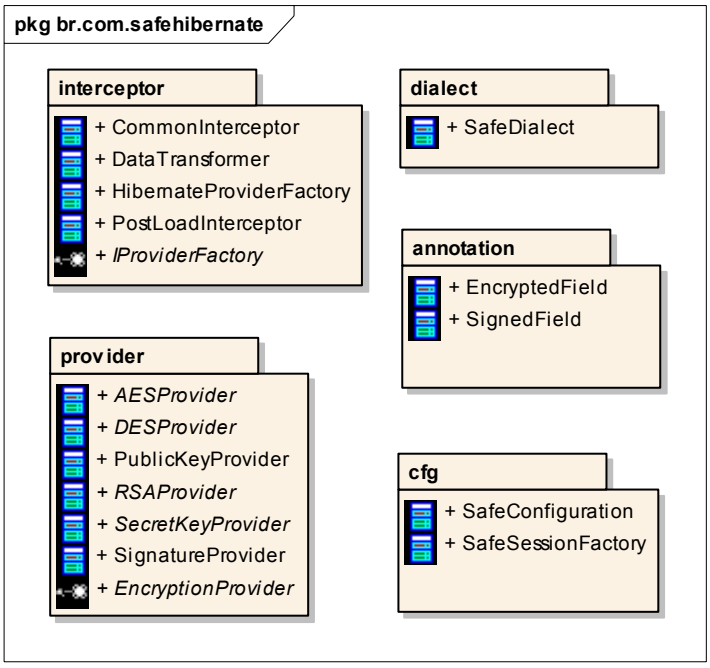


Figura 6 – Diagrama de pacotes

Os pacotes estão distribuídos da seguinte forma:

a) `interceptor` contém as classes responsáveis por interceptar as operações do Hibernate e realizar as alterações necessárias;

b) `provider` contém as classes responsáveis por fornecer as implementações padrão de algoritmos de criptografia e de assinatura digital;

c) `dialect` contém a implementação do dialeto que realiza a ponte entre o *framework* e o Hibernate;

d) `cfg` contém as classes responsáveis por facilitar a integração entre o *framework* e o Hibernate.

As classes do pacote interceptor são descritas no quadro 6.

Classe	Descrição
`CommonInterceptor`	Implementação da interface `org.hibernate.Interceptor`. Realiza inspeção nas operações de persistência ou materialização do Hibernate, permitindo que os valores sejam modificados.
`DataTransformer`	Responsável por realizar as tranformações necessárias em um objeto ao persistir ou materializar o mesmo.
`IProviderFactory`	Interface que especifica factory para obtenção dos provedores de criptografia e de assinatura digital.
`HibernateProviderFactory`	Implementação da interface IProviderFactory que obtém os providers através de propriedades especificadas no arquivo hibernate.properties.
`PostLoadInterceptor`	Interceptor responsável pelo evento "post-load" do Hibernate através do qual intercepta a operação após a materialização do objeto e realizada a decriptografia e a verificação da assinatura digital dos dados persistidos.

Quadro 6 – Classes do pacote `interceptor`

As classes do pacote provider são descritas no quadro 7.

Classe	Descrição
`AESProvider`	Provider para a o algoritmo de criptografia de chave secreta Advanced Encryption Standard.
`DESProvider`	Provider para o algoritmo de criptografia de chave secreta Data Encryption Standard.
`PublicKeyProvider`	Implementação abstrata de `EncryptionProvider` para algoritmos de chave pública.
`RSAProvider`	Implementação do algoritmo de chave privada RSA.
`SecretKeyProvider`	Implementação abstrata de `EncryptionProvider` para algoritmos de chave secreta.
`SignatureProvider`	Provider para algoritmos de assinatura digital que pode ser configurado para atuar em conjunto com algoritmos de criptografia.
`EncryptionProvider`	Interface base para as implementações de criptografia.

Quadro 7 – Classes do pacote `provider`

Os pacotes `dialect` e `cfg` possuem classes utilitárias para a configuração do dialeto do *framework* e para a configuração do Hibernate.

A lógica utilizada ao realizar a materialização dos objetos é apresentada na figura 7.

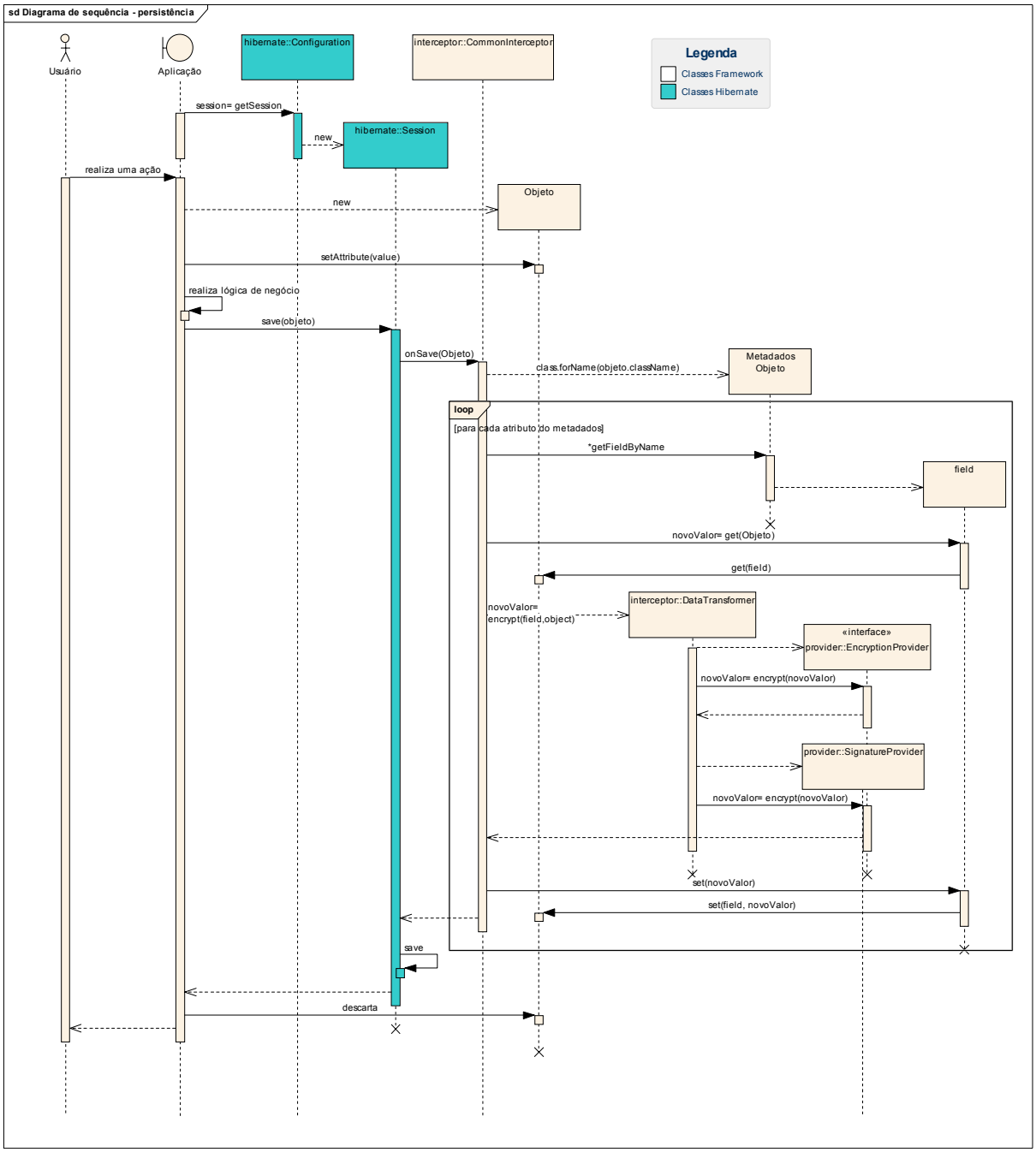


Figura 7 – Diagrama de seqüência da persistência dos objetos

Quando o usuário salva um objeto através de uma sessão Hibernate todos os *listeners* registrados durante a configuração do Hibernate que implementam a interface `org.hibernate.Interceptor` são notificados da operação; A classe `CommonInterceptor` é notificada pois é uma implementação desta interface e está devidamente registrada no objeto `configuration`.

Através do parâmetro objeto do método `onSave` é determinado via reflexão quais são os atributos do objeto a ser salvo e o objeto `DataTransformer` é notificado da operação com

o objeto e o campo como parâmetros.

Caso o atributo possua a anotação `EncryptedField` o objeto `DataTransformer` encripta e atualiza o valor do atributo através de uma instância de `EncryptionProvider` obtida através de `IProviderFactory`.

Se o atributo também possuir a anotação `SignatureField` o objeto `DataTransformer` assina e atualiza o valor do atributo através de uma instância de `SignatureProvider` também obtida através de `IProviderFactory`.

Após a inspeção de todos os atributos do objeto o Hibernate completa a persistência e retorna o controle para a aplicação.

O processo de materialização do objeto detalhado na figura 8 se assemelha muito ao processo de persistência. A principal diferença consiste na verificação da assinatura antes da decriptografia dos dados.

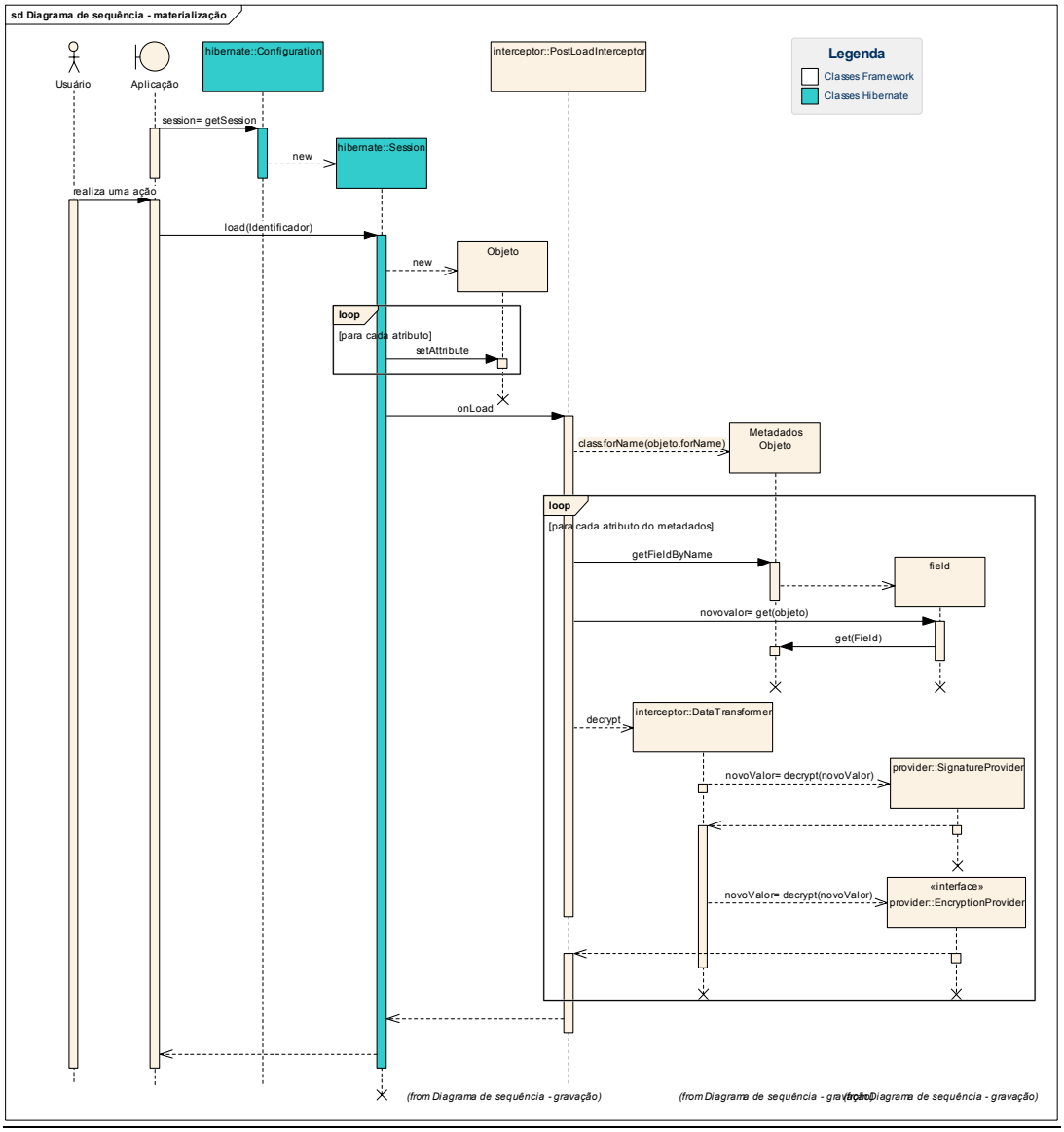


Figura 8 – Diagrama de seqüência da materialização dos objetos

3.3 IMPLEMENTAÇÃO DO FRAMEWORK.

Devido à necessidade de analisar os metadados dos objetos fornecidos pela aplicação utilizou-se do recurso de *annotations* especificado no documento *Java Specification Requirements* (JSR) 175 (JAVA COMMUNIT PROCESS, 2006). A anotação SignedField é apresentada no quadro 8.

```
package br.com.safehibernate.annotation;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

//Define a política de rentenção desta anotação.
//Definindo como Runtime esta anotação será mantida pela JVM
//durante a execução do programa, permitindo que seja obtida
//via reflexão.
@Retention (RetentionPolicy.RUNTIME)

//Define que esta anotação só pode ser utilizada em campos de classes
@Target (ElementType.FIELD)

//Define que pode ser usada pelo JavaDoc nos atributos que a utilizam
@Documented

/**
 * Anotação que identifica que o atributo deve ser assinado digitalmente
 */
public @interface SignedField {

       /**
        * Permite que a assinatura seja armazenada em outro campo
        */
       String storeOnField() default "this";
}
```

Quadro 8 – Anotação `SignedField`

A seguir são mostradas as técnicas e ferramentas utilizadas e a operacionalidade do *framework*.

3.3.1 Técnicas e ferramentas utilizadas

O *framework* foi escrito em linguagem Java utilizando o ambiente de programação Eclipse 3.3.1 com a JVM 1.5. A API *Bouncy Castle* foi utilizada para prover as implementações dos algoritmos de criptografia disponibilizados pelo *framework*. Para a assinatura digital foi utilizado o *design pattern* Proxy.

Design patterns, em linhas gerais, podem ser considerados como formas convenientes

de reutilizar código orientado a objetos e entre desenvolvedores (COOPER, 1998, p. 10). Cada *design pattern* possui um nome e se resume em descrever um problema, uma solução, e por fim, as conseqüências de sua utilização (GAMMA et al, 1998m p.17).

O *framework* empregou os seguintes *design patterns*:

a) Singleton: o *design pattern* singleton é utilizado para restringir a instanciação de uma classe para um único objeto, seu emprego se fez nececessário na classe `br.com.safeHibernate.interceptor.DataTransformer` por uma questão performática, uma vez que o custo para instanciar um objeto de EncryptionProvider ao cifrar ou decifrar uma informação é muito alto.
b) Decorator: o *design pattern* decorator adiciona características a uma classe sem alterar sua estrutura. A classe `br.com.safehibernate.dialect.SafeDialect` utiliza-se deste *design pattern* para realizar a interface com o dialeto do banco de dados.

3.3.2 Operacionalidade da implementação

A operacionalidade da aplicação se dá através da utilização do dialeto SafeHibernate em substituição ao dialeto Hibernate.

Como estudo de caso, foi implementada uma aplicação de exemplo que controla registros médicos de pacientes. A aplicação possui os seguintes requisitos:

a) permitir a utilização de diferentes bancos de dados sem modificar a estrutura interna do programa;
b) garantir a integridade das informações através da utilização de assinatura digital;
c) garantir o sigilo dos registros médicos garantindo que somente o médico e o paciente possam ter acesso às mesmas;
d) garantir a autenticação dos usuários;
e) garantir o não-repúdio quanto ao registro realizado pelo médico.

Como servidor de aplicação foi utitilizado o *container* JEE JBoss 4.2.2.

A figura 9 mostra o diagrama de classes do modelo de negócio da aplicação:

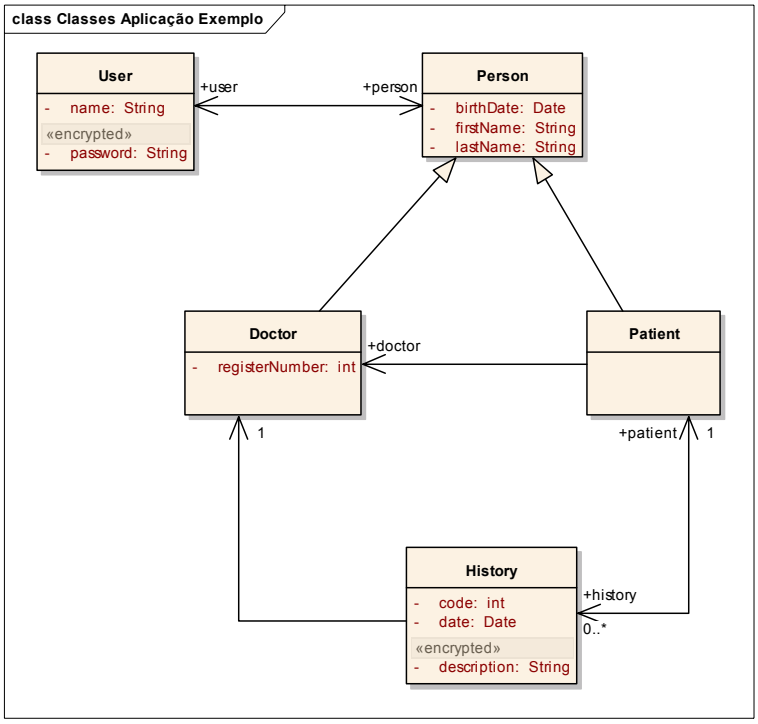


Figura 9 – Diagrama de classes da aplicação

O mapeamento das classes para o modelo relacional utilizando o banco de dados MySQL pode ser verificado no apêndice A.

O modelo entidade relacionamento da aplicação é apresentado na figura 10.

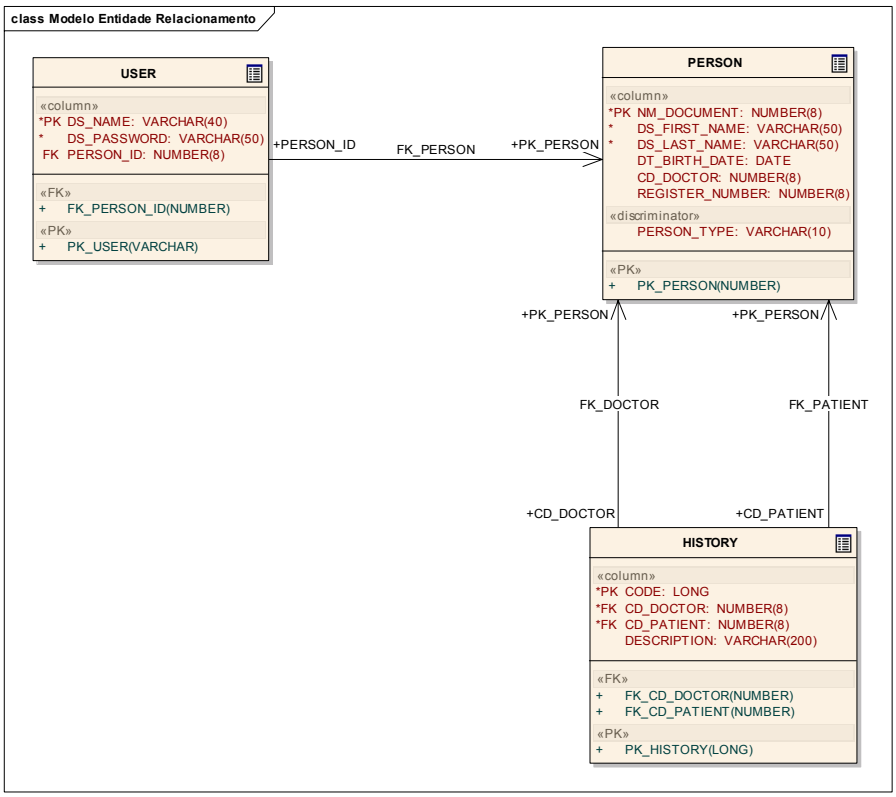


Figura 10 – Modelo entidade-relacionamento da aplicação

A aplicação foi construída utilizando inicialmente o *framework* Hibernate e

posteriormente foram feitas as alterações necessárias para a utilização do *framework* SafeHibernate. As alterações necessárias foram as seguintes:

a) adicionar ao arquivo de configuração do Hibernate a entrada do dialeto `SafeDialect` realizando interface para o dialeto original e a implementação da interface `CertificationProvider` e da interface `SignatureProvider`. O arquivo `hibernate.properties` após a modificação é apresentado no quadro 9.

```
#Configuração para acesso ao bando de dados
connection.driver_class=com.mysql.jdbc.Driver
connection.url=jdbc:mysql://localhost/medicalapp
connection.username=hibernate
connection.password=hibernate

#Configura o safeHibernate para utilizar MySql como dialeto primário
safeHibernate.wrappedDialect=org.hibernate.dialect.MySQLDialect

#Define qual é a classe que provê acesso ao certificado do usuário logado.
safeHibernate.encryptionProvider=br.com.medical.util.MedicalEncryptionProvider
safeHibernate.signatureProvider=br.com.medical.util.MedicalSignatureProvider

#Debug
show_sql=true
hibernate.format_sql=true
hibernate.hbm2ddl.auto=update
```

Quadro 9 – arquivo hibernate.properties

b) implementar uma classe que realiza a interface `EncryptionProvider` para permitir acesso as chaves do usuário ativo durante a criptografia. A classe é apresentada no quadro 10;

```
/**
 * Provê uma chave secreta para realizar a criptografia dos
campos
 * É utilizado como algoritmo o DES no modo ECB
 */
public class MedicalEncryptionProvider extends DESProvider {

	//Esta threadLocal mantém uma instância de UserTO por
usuário.
	public static ThreadLocal<UserTO> threadLocal = new
ThreadLocal<UserTO>();

	/**
	 * Retorna a chave secreta do usuário através da
ThreadLocal
	 */
	@Override
	public SecretKey getSecretKey() {
		return threadLocal.get().secretKey;
	}
}
```

Quadro 10 –Implementação da classe MedicalEncryptionProvider.

c) Implementar uma classe que realiza a interface `SignatureProvider` para permitir acesso as chaves do usuário ativo durante a assinatura digital.

A classe é apresentada no quadro 11;

```
/**
 * Provê um par de chaves para a assinatura digital dos campos
 */
public class MedicalSignatureProvider extends SignatureProvider {

	//Utiliza o algoritmo SHA1 com RSA para realizar a
assinatura.
	private static final String ALGORITHM = "SHA1withRSA";

	//O tamanho em bytes da assinatura é de 176 bytes
	private static final int SIGNATURE_SIZE = 176;

	public MedicalSignatureProvider() throws
NoSuchAlgorithmException {
		super(ALGORITHM, SIGNATURE_SIZE);
	}

	/**
	 * A chave privada é obtida em um par de chaves de UserTO
obtido através da ThreadLocal
	 * @see UserTO
	 */
	@Override
	public PrivateKey getPrivateKey() {
		return
MedicalEncryptionProvider.threadLocal.get().keyPair.getPrivate();
	}

	/**
	 * A chave pública é obtida em um par de chaves de UserTO
obtido através da ThreadLocal
	 * @see UserTO
	 */
	@Override
	public PublicKey getPublicKey() {
		return
MedicalEncryptionProvider.threadLocal.get().keyPair.getPublic();
	}

}
```

Quadro 11 –Implementação da classe MedicalSignatureProvider

d) Para garantir o não repúdio quanto ao registro realizado pelo médico foi adicionado ao atributo `doctor` da classe `History` a anotação `SignedField`. Foi adicionado também ao atributo `description` a anotação `SignedField`. O quadro 12 demonstra ao trecho de código relacionado;

```
...
pubic class History implements Serializable {
    ...
        //O médico que realizou o atendimento.
        @SignedField(storeOnField="doctorCodeSignature")
        private Doctor doctor;

        private String doctorCodeSignature;
    ...
        @EncryptedField
        //Contém a descrição do atendimento realizado.
        private String description;
    ...

};
```

Quadro 12 – Declaração de um atributo assinado digitalmente

e) Alterar a classe da aplicação responsável por iniciar a configuração do Hibernate para utilizar a classe SafeConfiguration. A alteração é apresentada no quadro 13.

```
private void configure() {
    Configuration cf = new Configuration().configure();
    this.sf = cf.buildSessionFactory();
}
```

```
private void configure() {
    // SessionFactory deve ser criado uma Única vez durante a
execução da aplicação
    Configuration cf = new
SafeConfiguration(MedicalEncryptionProvider.class,
                  MedicalSignatureProvider.class).configure();
    this.sf = cf.buildSessionFactory();
}
```

Quadro 13 – Alteração no configurador do Hibernate

Após a alteração a aplicação foi empacotada novamente e foi realizado o *deploy* para o servidor de aplicação.

Ao executar a aplicação cliente é apresentada a tela de *login* apresentada na figura 11.

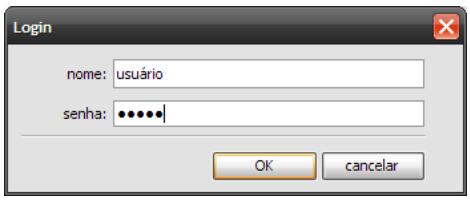


Figura 11 – Tela de *login*

Quando o usuário informa um nome e senha válidos é instanciado e mantido em um objeto estático ThreadLocal uma chave privada obtida através da senha informada utilizando o algoritmo Password Based Encryption (PBE), assim como um par de chaves pré-cadastrado. Esta chaves são utilizadas pela aplicação nas implementações dos *providers* fornecidos ao *framework*.

Quando um médico entra no sistema lhe é apresentada uma tela onde ele pode pesquisar pacientes cadastrados e rever seus históricos. Esta tela é apresentada na figura 12.

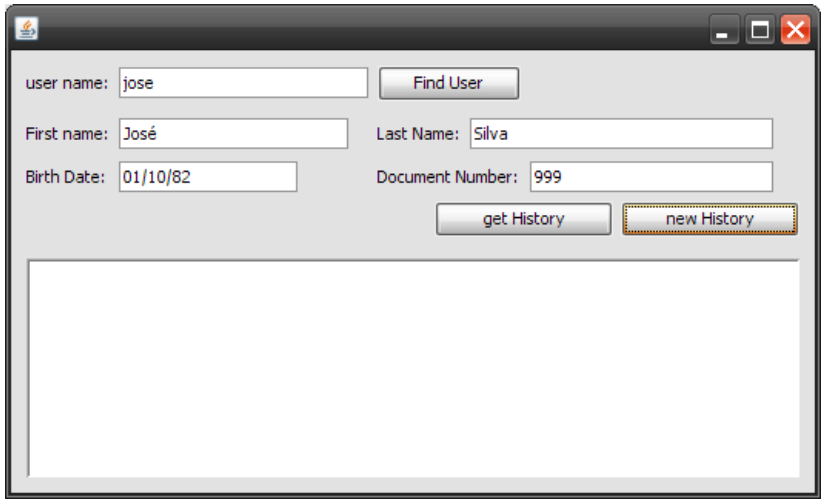


Figura 12 – Tela utilizada pelo médico para consultar dados dos pacientes

No exemplo apresentado foi informado o nome de um usuário que não possuía nenhum histórico médico anterior. Após pesquisar o histórico do paciente o médico aciona no botão `new History` e é apresentada a tela da figura 13.

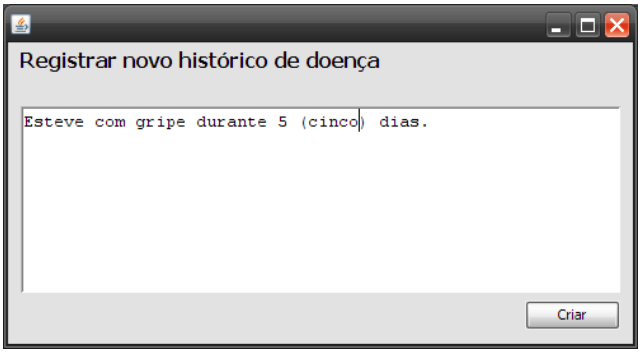


Figura 13 – Tela de registro de históricos médicos

Nesta tela o médico informa uma descrição do histórico e aciona o botão `Criar`. Neste momento é criado e persistido um histórico para o paciente na classe `History`. Esta classe é mapeada através do arquivo `History.hbm.xml` para a persitência no banco de dados. O atributo `description` é criptografado e o atributo `doctor` é assinado digitalmente pelo médico que criou o histórico. A assinatura do médico fica armazenada no atributo `doctorCodeSignature`.

O trecho relevante da classe `History` é apresentada no quadro 14.

```
/**
 * Um histórico é um registro de um atendimento que um médico realizou
 * para um paciente.
 */
public class History implements Serializable {

      private static final long serialVersionUID = -7415123767728136038L;

      //O código do atendimento.
      private Integer code;
```

```
        //O médico que realizou o atendimento.
        @SignedField(storeOnField="doctorCodeSignature")
        private Doctor doctor;

        private String doctorCodeSignature;

        //O paciente atendido.
        private Patient patient;

        @EncryptedField
        //Contém a descrição do atendimento realizado.
        private String description;
        ...
}
```

Quadro 14 – Classe `br.com.medical.model.History`

Na figura 14 é apresentada uma consulta na tabela `history` que mantém os registros dos históricos persistidos onde é evidenciado que os campos `description` e `doctor_signature` foram criptografados.

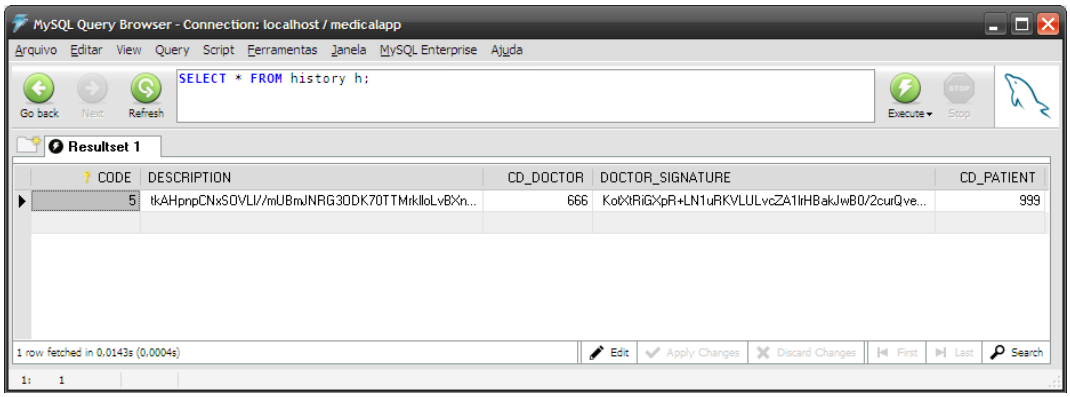


Figura 14 – Consulta a tabela HISTORY

Quando o médico consulta novamente o histórico deste paciente a validade da assinatura digital é verificada e a informação é decriptografada. Na figura 15 é apresentada a tela após a consulta do histórico.

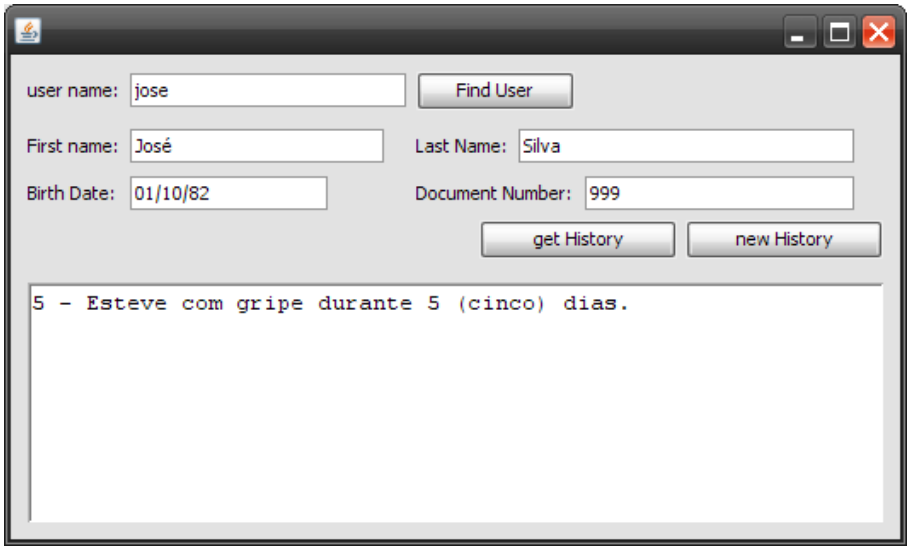


Figura 15 – Consulta do histórico do paciente

3.4 RESULTADOS E DISCUSSÃO

A utilização do recurso de *annotations* implementado à partir da versão 5.0 da linguagem Java facilitou a adoção do *framework* por parte do usuário, já que permite facilmente adicionar as características desejadas sem grandes alterações no código.

O uso da linguagem Java também facilitou a aderência da orientação a objetos, assim como a API bouncy castle forneceu a implementação dos algoritmos de criptografia necessários para a implementação do *framework* e para futuras extensões por parte do usuário, contornando as limitações de tamanho de chave e de algoritmos não fornecidos na distribuição padrão da linguagem Java.

Para o desenvolvimento do estudo de caso utilizando o Hibernate com uma camada de apresentação em Swing foi necessário o emprego dos *design patterns business delegate* e *session façade* para simular uma sessão de usuário na camada de aplicação e fornecer os certificados. Apesar de o *framework* Hibernate ser largamente utilizado em aplicações web utilizando a sessão http para armazenar objetos do usuário, dificilmente é utilizado neste contexto.

O Hibernate apresenta vasta documentação para usuários através de javadoc, html e fóruns, porém é muito limitado quanto as suas capacidades de extensão, o que exigiu recorrer ao código-fonte para identificar a forma correta de implementar o *framework*.

Apesar de ser um recurso importante para a validação do *framework* o uso de assinatura digital nos atributos nem sempre é viável em aplicações reais que demandem que grande performance, no entanto, o uso de anotações distintas para a criptografia e para a assinatura digital permite que este recurso seja utilizado somente quando necessário, sem comprometer a criptografia.

4 CONCLUSÕES

O desenvolvimento de um *framework* para criptografia de dados persistidos não é uma tarefa simples. Apesar de utilizar ferramentas de alta produtividade como o ambiente integrado de desenvolvimento eclipse e um *framework* muito bem fundamentado como o Hibernate, os detalhes arquiteturais envolvidos exigem bastante dedicação.

O objetivo principal deste trabalho foi alcançado com sucesso. O *framework* desenvolvido é capaz de persistir e assinar digitalmente de forma transparente os atributos no estudo de caso apresentado.

Pode-se concluir que o desenvolvimento de um *framework* de persistência de dados com criptografia e assinatura digital é viável. Isto é comprovado pelo estudo de caso apresentado neste trabalho, no qual, através de poucas modificações na aplicação, foi possível habilitar a criptografia e ou assinatura digital dos dados.

4.1 EXTENSÕES

As seguintes sugestões são apresentadas para o *framework* apresentado:

a) construir uma estrutura de chave pública que permita a obtenção de certificados digitais e listas de certificados revogados;
b) verificar a validade e autenticidade dos certificados associados às chaves públicas utilizadas na criptografia;
c) fornecer um meio padrão para o gerenciamento de usuários e de chaves através do *framework*;
d) interceptar consultas e projeções que envolvam somente alguns atributos de objetos e realizar a criptografia e ou assinatura digital nestes.

REFERÊNCIAS BIBLIOGRÁFICAS

ALECRIM, Emerson. **Criptografia**. [S.l.], 2005. Disponível em: <http://www.infowester.com/criptografia.php >. Acesso em: 31 out. 2007.

BAUER, Christian; KING, Gavin. **Hibernate em ação**. Rio de Janeiro: Ciência Moderna, 2005.

BANCO BRADESCO S.A. **O que é segurança da informação.** [S.l.], [2006?]. Disponível em: <http://www.bradesco.com.br/br/seguranca_informacao/oquee.html>. Acesso em: 31 out. 2007.

BAPTISTA, Cláudio. **Banco de dados**. [S.l.], [2006?]. Disponível em: <www.dsc.ufcg.edu.br/~baptista/ cursos/BDadosI/Unidade1.doc>. Aceso em: 31 out. 2007.

BRUNETT, Steve; PAINE, Stefen. **Criptografia e segurança:** o guia oficial RSA. Rio de Janeiro: Campus, 2002.

CARVALHO, Daniel Balparda de. **Segurança de dados com criptografia**: métodos e algoritmos. 2. ed. Rio de Janeiro: Book Express, 2001.

CHAMBERLIN, Donald D. **SEQUEL/2**: a unified approach to data definition, manipulation and control. 6 ed. San Jose: IBM, 1976.

COOD, Edgar F. **The relational model for database management**. Boston: Addison Wesley Publishing Company, 1990.

DATE, Cristopher J. **Banco de dados**: fundamentos. Tradução Contexto Traduções Ltda. Rio de Janeiro: Campus, 1985.

GIANISINI JR., João Batista. **Desenvolvimento de um framework para replicação de dados entre bancos heterogêneos**. 2006. 15 f. Proposta de Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

HIBERNATE. **Relational persistence for Java and .NET**. [S.l.], 2006. Disponível em: <http://www.hibernate.org/>. Acesso em: 31 out. 2007.

HIETEL, Stefan. **Data control language**. [S.l.], [2002?]. Disponível em: <http://rowa.giso.de/oracle/latex/Data_Control_Language.html>. Acesso em: 04 nov. 2007.

HOLOVAC, Srdjan. **Securing data at rest:** database encryption solution using empress RDBMS. [S.l.], [2006?]. Disponível em: <http://www2.empress.com/marketing/encryption/EmpressSecurityWhitePaper.v11.htm>. Acesso em: 31 out. 2007.

INTERNATIONAL TELECOMMUNICATIONS UNION. S**ecurity guidance for ITU-T recommendations**. Geneva, 1991. Disponível em: <www.itu.int/ITU-T/studygroups/com17/security-guidance.doc>. Acesso em: 31 out. 2007.

JAVA COMMUNIT PROCESS. **JSR 175**: a metadata facility for the Java programming language. [S.l.], 2006. Disponível em: <http://jcp.org/en/jsr/detail?id=175>. Acesso em: 4 nov. 2007.

LEOCÁCIO, Paulo. **Teoria de base de dados**. [S.l.], [2005?]. Disponível em: <http://www.uac.pt/~pleocadio/gestao_qualidade/Teoria_Base_Dados.pdf>. Acesso em: 31 out. 2007.

LOURENÇO, Carlos. **Segurança por omissão de sistemas de gestão de bases de dados**. Lisboa, 2007. Disponível em: <www.di.fc.ul.pt/~mpc/pubs/ Lourenco-ConfiguracaoOmissaoSGBD-sinos07.pdf>. Acesso em: 21 out. 2007.

MAINWALD, Eric. **Network security**: a beginner's guide. New York: McGraw-Hill/Osborne, 2001.

MENESES, Alfred; OORSCHOT, Paul van; VANSTONE, Scott. **Handbook of applied cryptography**. [S.l.], 1997. Disponível em: <http://www.cacr.math.uwaterloo.ca/hac/>. Acesso em: 31 out. 2007.

SCHWEBEL, Samuel C. **Frasedare**: framework orientado a objetos para segurança de dados em repouso. 2005. 165 f. Dissertação (Mestrado em Ciência da Computação) – Curso de Pós-Graduação em Ciência da Computação, Universidade Federal de Santa Catarina, Florianópolis.

SILVA JR., Antônio M. **O papel de protagonistas no desenvolvimento de sistemas interativos.** [S.l.], [2005?]. Disponível em: <http://www.espacoacademico.com.br/047/47amsf.htm>. Acesso em: 31 out. 2007.

SILVA, Rafael P. **Integridade e segurança em bancos de dados**. [S.l.], [2006?]. Disponível em: <http://www.nuted.edu.ufrgs.br/biblioteca/texto.php?texto=94&assunto=8>. Acesso em: 31 out. 2007.

STALLINGS, William. **Cryptography and network security**: principles and practice. 3rd. ed. Upper Saddle River: Prentice Hall, 2003.

SUN MICROSYSTEMS. **Java database connectivity (JDBC)**. [S.l.], 2006. Disponível em: <http://java.sun.com/products/jdbc/overview.html>. Acesso em: 31 out. 2007.

TONAHAN, Torin. **Surveillance and security**: technological politics and power in everyday life. New York: Routledge, 2006.

APÊNDICE A – Mapeamento das classes para o modelo relacional

No quadro 15 é apresentado mapeamento da classe History para o modelo relacional.

```
<?xml version="1.0" encoding="UTF-8"?>
 <!DOCTYPE hibernate-mapping
   PUBLIC "-//Hibernate/Hibernate Configuration DTD//EN"
   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
 <hibernate-mapping>
   <class name="br.com.medical.model.History"
       dynamic-update="true"
       dynamic-insert="true"
       select-before-update="true"
       lazy="true"
       table="HISTORY">
       <id name="code" column="CODE" type="java.lang.Integer">
           <generator class="identity"/>
       </id>
       <property name="description" column="DESCRIPTION" type="java.lang.String"
length="200"/>

       <many-to-one name="doctor" class="br.com.medical.model.Doctor"
column="CD_DOCTOR" not-null="true"/>
       <property name="doctorCodeSignature" column="DOCTOR_SIGNATURE"
type="java.lang.String" length="250" not-null="false"/>

       <many-to-one name="patient" class="br.com.medical.model.Patient"
column="CD_PATIENT" not-null="true"/>
   </class>
 </hibernate-mapping>
```

Quadro 15 – Mapeamento da classe History

No quadro 16 é apresentado mapeamento da classe Person para o modelo relacional.

```
<?xml version=”1.0" encoding="UTF-8"?>
 <!DOCTYPE hibernate-mapping
   PUBLIC "-//Hibernate/Hibernate Configuration DTD//EN"
   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
 <hibernate-mapping>
   <class name="br.com.medical.model.Person"
       dynamic-update="true"
       dynamic-insert="true"
       select-before-update="true"
       lazy="true"
       table="PERSON"
       discriminator-value="PERSON">

       <id name="document" column="DOCUMENT" type="int"/>
          <discriminator type="string" length="7" column="PERSON_TYPE"/>

       <property name="firstName" column="FIRST_NAME" type="java.lang.String"
length="40" not-null="true"/>
       <property name="lastName" column="LAST_NAME" type="java.lang.String"
length="40" not-null="true"/>
       <property name="birthDate" column="BIRTH_DATE" type="java.util.Date"/>

       <one-to-one name="user" class="br.com.medical.model.User" property-
ref="person"/>

       <!-- Patient subclass -->
       <subclass name="br.com.medical.model.Patient" discriminator-value="PATIENT">
            <many-to-one name="doctor" class="br.com.medical.model.Doctor"
update="false" insert="false">
            </many-to-one>
             <set name="history" inverse="false" order-by="code">
             <key column="code"/>
                <one-to-many class="br.com.medical.model.History"/>
            </set>
       </subclass>

       <!-- Doctor subclass -->
       <subclass name="br.com.medical.model.Doctor"
                 discriminator-value="DOCTOR">
            <property name="registerNumber" column="REGISTER_NUMBER" type="long"/>
       </subclass>

   </class>
 </hibernate-mapping>
```

Quadro 16 – Mapeamento da classe Person

No quadro 17 é apresentado mapeamento da classe User para o modelo relacional.

```
<?xml version="1.0" encoding="UTF-8"?>
 <!DOCTYPE hibernate-mapping
   PUBLIC "-//Hibernate/Hibernate Configuration DTD//EN"
   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
 <hibernate-mapping>
   <class name="br.com.medical.model.User"
       dynamic-update="true"
       dynamic-insert="true"
       select-before-update="true"
       lazy="true"
       table="USER">
       <id name="name" column="NAME" type="java.lang.String" length="20">
           <generator class="assigned"/>
       </id>
       <property name="password" column="PASSWORD" type="java.lang.String"
length="40"/>
```

```
        <property name="privateKey" column="PRIVATE_KEY" type="binary"/>
        <property name="publicKey" column="PUBLIC_KEY" type="binary"/>
        <property name="secretKey" column="SECRET_KEY" type="binary"/>

        <many-to-one name="person" class="br.com.medical.model.Person"
unique="true" lazy="false" not-null="true">
              <column name="PERSON_ID"></column>
        </many-to-one>

   </class>
 </hibernate-mapping>
```

Quadro 17 – Mapeamento da classe User