JDOSecure: Ein Sicherheitsmodell für die Java Data Objects-Spezifikation

Matthias Merz

Arbeitspapier 3/2005
Juli 2005

Arbeitspapiere in der Wirtschaftsinformatik

Matthias Merz
Universität Mannheim
Lehrstuhl für Wirtschaftsinformatik III
Schloss, D-68131 Mannheim
merz@uni-mannheim.de

JDOSecure: Ein Sicherheitsmodell für die Java Data Objects-Spezifikation

Matthias Merz
Universität Mannheim
Lehrstuhl für Wirtschaftsinformatik III
Schloss, D-68131 Mannheim
merz@uni-mannheim.de

5. August 2005

Zusammenfassung

Dieser Artikel zeigt zunächst die konzeptuellen Schwächen der JDO Architektur auf. Es wird deutlich, dass auch die JDO-Spezifikation in der Version 2.0 keinen Mechanismus zur Authentifizierung und Autorisierung auf Basis von Benutzergruppen, Rollen und Rechten bietet. Hierdurch bedingt lassen sich über Methoden des `PersistenceManager`s u. a. ungeprüft Transaktionen zurücksetzen, persistente Objekte löschen sowie beliebige Objekte aus dem Hintergrundspeicher rekonstruieren.

Das Sicherheitsmodell JDOSecure gestattet die Einführung benutzerspezifischer Zugriffsrechte bei Verwendung der JDO-API. Die Rechte lassen sich für einen Anwender bzw. dessen Rollen sowie für eine bestimmte Paket- bzw. Klassen-Ebene individuell definieren. JDOSecure kann ohne Modifikation des Quellcodes mit einer beliebigen JDO-Implementation eingesetzt werden. Die Realisierung der Authentifizierung und Autorisierung erfolgt über den Java Authentication and Authorization Service, wodurch auch die Anbindung externer Authentifizierungsmodule zur Nutzung weiterer Dienste wie Kerberos, Radius oder LDAP möglich wird.

1 Forschungsüberblick

1.1 Grundlagen der Objekt-Persistenz

Objektorientierte Programmiersprachen gestatten die Konstruktion komplexer Objekte, deren Existenz jedoch im allgemeinen an den Lebenszyklus einer Ablaufumgebung gebunden ist. Objekte, die dieser Beschränkung nicht unterliegen, werden *persistente Objekte* genannt (Schmidt 1991, S. 23-24). In der Programmiersprache Java können Objekte eine Terminierung der *Java Virtual Machine* überdauern, wenn zuvor Objekt-Zustand, Objektidentität und Typ-Information beispielsweise in eine Datei serialisiert werden. Objekte lassen sich so zu einem späteren Zeitpunkt *(zeitliche Persistenz)* oder außerhalb des aktuellen Adressraums *(räumliche Persistenz)* rekonstruieren.

In Java stellt die Objektserialisierung eine einfache Möglichkeit zur Realisierung von Objekt-Persistenz dar. Anwendungsentwickler können Objekte über einen expliziten Methodenaufruf serialisieren, sofern diese die Schnittstelle `java.io.serilizable` implementieren. Hierbei wird automatisch der vollständige Objektgraph berücksichtigt, d. h. sämtliche direkt und indirekt

erreichbaren Objekte werden ebenfalls serialisiert. Ein direkter Zugriff auf einzelne Objekte im Bytestrom, beispielsweise zur Änderung eines einzigen Objekt-Attributs, ist jedoch nicht möglich. Ist die Änderung eines solchen erforderlich muss zunächst der vollständige Bytestrom eingelesen und sämtliche Objekte müssen im Hauptspeicher rekonstruiert werden. Nach erfolgter Änderung ist anschließend das erneute Serialisieren des gesamten Objektgraphs notwendig. Weitere Nachteile der Objektserialisierung wie z. B. das nicht Berücksichtigen statischer Klassenvariablen, fehlende Versionskontrolle oder mögliche Auswirkungen einer Deserialisierung auf die Semantik der Domäne werden von Atkinson näher untersucht (Atkinson 2001).

Die Ansätze zur Realisierung von Objektpersistenz, welche wie die Objektserialisierung auf einfache Dateisysteme basieren, bieten in der Regel keine Leistungsmerkmale wie Mehrbenutzerzugriff, Transaktionskontrolle oder Ad-hoc Anfragen. Datenbanksysteme sind daher zur Realisierung von Objektpersistenz von besonderem Interesse. Zugleich ist der Einsatz relationaler Datenbanken aufgrund unterschiedlicher Konzepte (relationales Datenmodell vs. objektorientiertem Paradigma) besonders problematisch. Dieser Sachverhalt wird in der Literatur unter dem Stichwort *Impedance Mismatch* diskutiert (Atkinson, Bancilhon, DeWitt, Dittrich, Maier und Zdonik 1989; Schäfer 2003). Es zeigt sich, dass generell Klassen auf Tabellen und Attribute auf Tupel abgebildet werden können. Einfache Objekt-Beziehungen lassen sich zudem über Fremdschlüssel realisieren. Allerdings bereitet die Umsetzung objektorientierter Konzepte wie z. B. Kapselung, Polymorphismus, Vererbung und Beziehungen besondere Probleme. Bei der Persistenzierung eines komplexen Objekts ist zudem die zeitlich aufwändige Zerlegung des Objekts in einzelne Bestandteile notwendig. Analog folgt das zeitaufwändige Zusammensetzen dieser Teile bei der Rekonstruktion des Objekts.

Bereits 1989 haben die Autoren des *Object-Oriented Database System Manifesto* daher die Verwendung objektorientierter Datenbanksysteme (OODB) zur Realisierung von Objekt-Persistenz postuliert (Atkinson et al. 1989). Sie begründen die nach ihrer Ansicht gegebene Vorteilhaftigkeit objektorientierter Datenbanken gegenüber relationaler Datenbanken u. a. mit der „Durchgängigkeit“ des objektorientierten Ansatzes. Die ersten OODB ermöglichten bereits die transparente Persistenzierung komplex strukturierter Objekte und verfügten u. a. über Methoden zum navigierenden Zugriff auf diese. Das Fehlen übergreifender Standards führte jedoch bald zu gravierenden Unterschieden einzelner OODB bezüglich des zugrundeliegenden Objektmodells und der jeweils verwendeten Datendefinitions- und Anfragesprachen. Dieses Defizit versuchte die 1991 gegründete *Object Database Management Group* (ODMG) als Zusammenschluss führender OODB-Hersteller durch Verabschiedung des ODMG-Standards auszugleichen (Object Data Management Group 2000). In diesem wurden u. a. ein sprachneutrales Objektmodell, eine Objektspezifikationssprache, ein Objektaustauschformat, die Anfragesprache OQL sowie unabhängige Sprachanbindungen für unterschiedliche Programmiersprachen festgeschrieben. Obgleich der ODMG-Standard zwischenzeitlich von mehreren Herstellern implementiert wurde, konnten sich OODB, entgegen der in der Literatur festgestellten Vorteilhaftigkeit, bis heute nicht am Markt durchsetzen. Die ODMG stellte folglich die Weiterentwicklung des ODMG-Standards ein und löste sich 2001 auf. Lediglich die Java-Sprachanbindung wurde dem Java Community Process (JCP) als Ausgangsbasis zur Erarbeitung eines neuen Persistenz-Standards für Java übergeben (siehe Java Data Objects in Abschnitt 1.2).

Die konzeptionelle Lücke zwischen den nach wie vor weit verbreiteten relationalen Datenbanken und objektorientierten Programmiersprachen versuchen *Object Relational Mapping Tools* (O/R-Mapper) zu überwinden. Diese verdecken den Low-Level Zugriff auf relationale Daten-

banken und sorgen aus Sicht der Anwendungsentwickler für eine automatische und transparente Persistenzierung der Objekte. Allerdings sind O/R-Mapper wie z. B. TopLink oder Hibernate in der Regel nicht standardisiert, wodurch sich Anwender häufig in Abhängigkeit des gewählten Tools oder einer konkreten Datenbank begeben.

Aufgrund dieser unbefriedigenden Situation wurde insbesondere in der Java Community die Forderung nach einem einfachen, einheitlichen und transparenten Standard für Objektpersistenz erhoben, der zugleich die Anbindung beliebiger transaktionaler *Persistenzmedien*[1] ermöglichen soll. Dieser Anforderung versucht die Java Data Objects Spezifikation gerecht zu werden (Java Community Process 2004, S. 20).

1.2 Die Java Data Objects-Spezifikation

Java Data Objects (JDO) ist ein Industriestandard für Objekt-Persistenz und wurde auf Initiative von Sun Microsystems innerhalb des Java Community Process erarbeitet. JDO liegt seit Mai 2003 in der Version 1.0.1 vor und ermöglicht Java-Entwicklern einen transparenten Umgang mit persistenten Objekten. Die Umsetzung der JDO-Spezifikation in ein konkretes Software-Produkt wird als *JDO-Implementierung* bezeichnet. Die JDO-Spezifikation stellt hierbei die Austauschbarkeit unterschiedlicher JDO-Implementierungen untereinander sowie den Wechsel eines Persistenzmediums sicher. Eine Modifikation des Java-Quellcodes ist hierzu nicht notwendig. Neben der Verwendung von JDO als Persistenzlösung für die J2SE-Plattform (Java 2 Standard Edition) kann JDO auch in *Managed Environments* wie der J2EE-Umgebung (Java 2 Enterprise Edition) eingesetzt werden.

Die JDO-Spezifikation definiert zwei Pakete: Das JDO-Application Programming Interface (API) stellt Anwendungsentwicklern Interfaces für den Zugriff auf persistente Objekte zur Verfügung. Die Klassen und Interfaces des Service Provider Interface (SPI) werden dagegen ausschließlich von einer JDO-Implementierung genutzt.

Die Interfaces `PersistenceManager`, `Transaction` und `Query` sind in der JDO-API (`javax.jdo`) definiert. Der `PersistenceManager` dient als primäre Anwendungsschnittstelle und stellt Methoden zur Verwaltung des Lebenszyklus persistenter Objekte bereit. Das Interface `Transaction` dient zur Realisierung der Transaktionsverwaltung unter Anwendungskontrolle. Über das `Query`-Interface lässt sich die Suche nach persistenten Objekten anhand spezifischer Kriterien realisieren. Die JDO-Spezifikation sieht hierzu die Verwendung einer an Java angelehnten Anfragesprache, JDO Query Language (JDOQL), vor. Abbildung 1 zeigt den schematischen Aufbau des `javax.jdo`-Pakets.

Im SPI-Paket (`javax.jdo.spi`) ist u. a. das `StateManager`-Interface definiert, über das eine JDO-Implementierung den Zustand persistenter Instanzen bzw. deren Felder verwalten kann. Über die `JDOImplHelper`-Klasse können Instanzen persistenzfähiger Klassen zur Laufzeit ohne Methoden der Reflection-API (`java.lang.reflect`) konstruiert und deren Meta-Informationen ausgelesen werden. Um Sicherheitsrisiken auszuschließen, ist der Zugriff auf diese Klasse nur mit entsprechenden JDO-Zugriffsrechten gestattet, die über die Klasse `JDOPermission` realisiert werden.

Damit eine JDO-Implementierung in den Objektlebenszyklus einer Instanz eingreifen kann und Änderungen einzelner Objekt-Attribute automatisch an den Hintergrundspeicher propagiert

[1] Speichermedium, wie z. B. ein Dateisystem, relationale- oder objektorientierte Datenbank, in dem Objekte dauerhaft vorgehalten werden können.

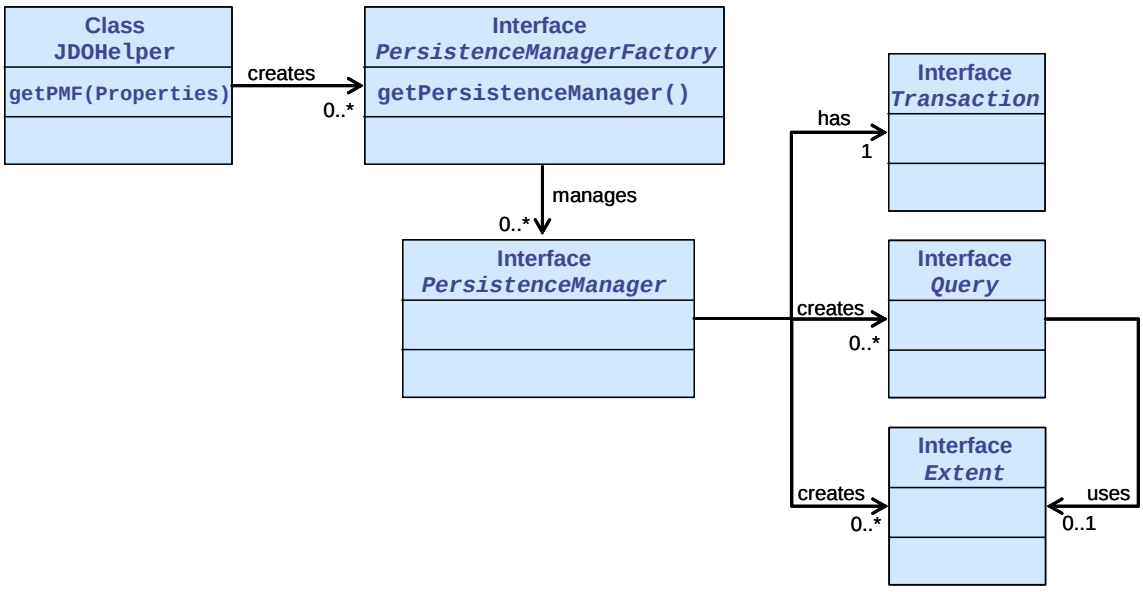

Abbildung 1: JDO-API

werden können, müssen persistenzfähige Klassen das `PersistenceCapable`-Interface implementieren. Dieses Interface ist Teil des SPI-Pakets. Diese Zuordnung deutet bereits an, dass die Implementierung des Interfaces nicht durch Anwendungsentwickler zu erfolgen hat. Stattdessen sieht die JDO-Spezifikation die Verwendung eines *JDO-Enhancers* – auch *Enhancment-Tool* genannt – vor, welches die Implementierung des Interfaces übernimmt. In der Regel handelt es sich hierbei um einen Postprozessor, der den Java-Bytecode einer Klasse nachträglich erweitert. Hierzu werden einfache Java Klassen, die auch als *Plain Old Java Objects* (POJOs) bezeichnet werden, wie üblich in Bytecode übersetzt. In einem XML basierten *persistence descriptor* werden durch den Anwendungsentwickler alle persistenzfähigen Klassen aufgeführt. Diese Informationen wertet der JDO-Enhancer aus und modifiziert den Bytecode nachträglich. Die JDO-Spezifikation schreibt in diesem Zusammenhang die Kompatibilität des erzeugten Bytecodes unterschiedlicher JDO-Implementierungen vor. Greifen Klassen eines Anwendungsentwicklers entgegen dem Prinzip der Objekt-Kapselung direkt auf `public` deklarierte Attribute von `PersistenceCapable`-Instanzen zu, muss auch der Bytecode dieser *persistence-aware* Klassen durch den JDO-Enhancer modifiziert werden. Anderenfalls könnten Änderungen von Objekt-Attributen nicht automatisch durch die JDO-Implementierung erkannt werden und zusätzliche Schritte, wie z. B. ein `makeDirty()`-Methodenaufruf, wären notwendig (Bishop, Mitchell, Bell, Holm und Ayers 2001, S. 907; Roos 2002, S. 95).

1.3 Kritische Betrachtung der Java Data Objects-Spezifikation

Selten wurde die Entwicklung eines Standards in Java so kritisch begleitet. Insbesondere das Konzept des JDO-Enhancers sorgt bis heute für intensive Auseinandersetzungen. Der JDO Expert Group wird u. a. vorgeworfen, mit Einführung des JDO-Enhancers Fehler der ODMG zu wiederholen (TheServerSide.COM 2003). Kritiker befürchten die Zunahme der Komplexität des

Entwicklungsprozesses und beanstanden die notwendige Konfiguration der persistence descriptors. Zudem führe die Modifikation am Bytecode zu Problemen bei der Fehleranalyse, da sich die Modifikationen nicht im Java-Quellcode widerspiegeln und z. B. die Änderung von Methodensignaturen für einen Anwendungsentwickler intransparent sind. Andere Persistenzlösungen wie db4o oder Hibernate nutzen anstelle eines Postprozessors Methoden der Reflection-API um Objekt-Attribute zur Laufzeit auszulesen und kommen daher ohne persistence descriptors aus (vgl. Database for Objects 2005; Hibernate 2005). Die JDO-Expert Group hält diesem Ansatz die hohe Performance ihres Enhancement-Prozesses zur Laufzeit entgegen.

Die Anfragesprache JDOQL gestattet über einen Filterausdruck das Auffinden persistenter Objekte, die anwendungsspezifischen Kriterien genügen. Dabei wird insbesondere die Umsetzung des Filterausdrucks als typunsicherer String kritisiert, da Tipp-Fehler durch den Java-Compiler zunächst unbemerkt bleiben und Ausnahmen erst während der Laufzeit ausgeworfen werden. Dagegen verfolgt beispielsweise die Simple Object Database Access-Query (S.O.D.A.) den typsicheren Query-By-Example-Ansatz, bei dem eine Anfrage über ein zur Laufzeit konstruiertes Beispiel-Objekt realisiert wird (vgl. SourceForge 2005). Ein weiterer Nachteil der Anfragesprache JDOQL wird in der geringen Mächtigkeit gesehen, die in der JDO-Spezifikation 1.0.1 weder Projektionen noch Aggregatfunktionen vorsieht.

Die Überschneidung zwischen dem Persistenzmodell der Enterprise JavaBeans-Spezifikation (Java Community Process 2003a) und JDO wird ebenfalls häufig kritisiert. Beide Standards gehen zwar auf Initiative von Sun Microsystems zurück, sind untereinander jedoch inkompatibel. Anwendungsentwickler stehen so vor der schwierigen Entscheidung, je nach Fall den jeweils geeigneteren Ansatz auszuwählen. Korthaus und Merz, welche die Vorteilhaftigkeit beider Konzepte einsatzspezifisch analysiert haben, bieten in diesem Zusammenhang eine Hilfestellung und geben entsprechende Empfehlungen (Korthaus und Merz 2003). Weitere Kritik richtet sich auch gegen technische Details der JDO-Spezifikation. Hierzu zählen beispielsweise die drei unterschiedlichen Konzepte zur Verwaltung der Identitäten persistenter Instanzen, der Zugriff auf beliebige Objekte über die `getObjectByID()`-Methode, sowie die zahlreichen *optional features* der JDO-Spezifikation.

Neben diesen technisch orientierten Kritikpunkten wird häufig auch der leichtgewichtige Ansatz der JDO-Architektur kritisiert. Beispielsweise bietet die JDO-Spezifikation keinen verteilten Zugriff auf persistente Objekte. Auch die Kommunikation einer JDO-Implementierung über die Grenzen des aktuellen Speicherbereichs hinaus, beispielsweise zur Synchronisation verteilter Objekt-Caches, ist in der aktuellen JDO-Spezifikation nicht vorgesehen (vgl. TheServerSide.COM 2001). Zudem fehlt ein rollenbasiertes Sicherheitsmodell, weshalb sich sicherheitskritische Vorgänge in JDO nicht durch Vergabe expliziter Rechte benutzerorientiert einschränken lassen. So haben Anwender nach dem Aufbau der Datenbankverbindung und der Konstruktion des `PersistenceManager`s über dessen Methoden Zugriff auf sämtliche persistenten Objekte. So lassen sich u. a. beliebig persistente Objekte löschen und ferner ungeprüft laufende Transaktionen deaktivieren oder der Objekt-Cache leeren.

Einige der hier deutlich gewordenen Defizite der JDO-Spezifikation werden im Rahmen der Weiterentwicklung des JDO-Standards behoben. Der nächste Abschnitt gibt hierzu einen Überblick und beschreibt weitere Entwicklungstendenzen.

1.4 Aktuelle Entwicklungen

Kontrovers diskutiert wird derzeit der als Java Specification Request (JSR) 243 eingebrachte Entwurf „Java Data Objects 2.0 - An Extension to the JDO specification“ (Java Community Process 2005b), der auf eine Erweiterung des JDO-Standards abzielt. Die Erweiterungsvorhaben umfassen u. a. die Vereinfachung und Ergänzung der Anfragesprache JDOQL (Java Data Objects Query Language), eine für alle JDO Implementierungen verbindliche, einheitliche Abbildungsvorschrift für relationale Datenbanken, sowie einen Mechanismus zur Abkopplung persistenter Objekte von der JDO-Laufzeitumgebung (*detached instances*).

Insbesondere letzteres Erweiterungsvorhaben wurde innerhalb der JDO-Expert Group intensiv diskutiert. IBM und BEA Systems warnen in diesem Zusammenhang vor einer Überschneidung mit dem von ihnen eingebrachten *Service Data Objects*-Standard (SDO), der den Austausch vollständiger Objekt-Graphen zwischen unabhängigen Rechnern ermöglicht (BEA Systems, Inc. und IBM Corp. 2005; Java Community Process 2003b). Oracle kritisiert als Mitglied der JDO und Enterprise JavaBeans-Expert Group die parallele Entwicklung zweier unterschiedlicher Persistenz-Standards für die J2EE-Plattform. Zwei Spezifikationen die mit unterschiedlichen APIs den gleichen Problembereich adressieren (persistente und transaktionale Semantik, Mapping-Definitionen und eine eigene Anfragesprache) wirken laut Oracle den Bemühungen zur Vereinfachung des EJB-Standards in der Version 3.0 entgegen (Java Community Process 2005a). Bei einer im April 2004 durchgeführten Abstimmung innerhalb des JCP (*Review Ballot*) votierten IBM, BEA Systems und Oracle daher gegen die geplanten Erweiterungen der JDO-Spezifikation.

Aufgrund dieser Kritik sah sich Sun Microsystems als Initiator beider Standards zum Handeln veranlasst. In einem gemeinsamen Brief der beiden Vorsitzenden (*Specification-Leads*) DeMichiel und Russell wurde im September 2004 die gemeinsame Entwicklung eines einheitlichen Persistenz-Standards angekündigt (DeMichiel und Russell 2004). Dieser wird derzeit als eigenständige Spezifikation innerhalb von EJB 3.0 erarbeitet, weshalb die EJB Expert Group um Mitglieder der JDO Expert Group erweitert wurde.

Ausgehend von der Anforderung nach Persistenzmedien-Unabhängigkeit wurde in Kapitel 1 deutlich, dass es derzeit zu einer Persistenzlösung auf Basis der JDO-Spezifikation keine annehmbaren Alternativen gibt. Bis zur Verabschiedung und Etablierung eines neuen, Java-einheitlichen Persistenz-Standards gilt JDO weiterhin als *state-of-the-art*. Zwei Punkte tragen hierzu wesentlich bei: Erstens, die Mitteilung zahlreicher JDO-Hersteller, den JDO-Standard auch zukünftig voll zu unterstützen und sobald erforderlich Migrationslösungen anzubieten. Zweitens, die Ankündigung durch Sun Microsystems, weiterhin die Verabschiedung erforderlicher JDO-*Maintenance Releases* innerhalb des JCP zu tolerieren. Wie in Abschnitt 1.3 jedoch auch deutlich wurde, stellt das fehlende Sicherheitsmodell ein schwerwiegendes Defizit dar, das einer breiteren Akzeptanz von JDO entgegen wirkt.

Das Sicherheitsmodell JDOSecure, das in Abschnitt 3 vorgestellt wird, soll helfen, dieses Defizit durch Einführung benutzerspezifischer Zugriffsrechte auszugleichen und so zu einer Steigerung der Akzeptanz von JDO beizutragen. Zum besseren Verständnis des JDOSecure Sicherheitskonzepts werden im folgenden Abschnitt zunächst die Sicherheitsdefizite der JDO-Spezifikation aufgezeigt und anschließend die Java 2 Sicherheitsarchitektur skizziert.

2 Sicherheitsdefizite der JDO-Spezifikation und die Java 2 Sicherheitsarchitektur

2.1 Sicherheitsdefizite der JDO-Spezifikation

Wie bereits in Abschnitt 1.3 deutlich wurde, sieht die JDO-Spezifikation derzeit keine Möglichkeit zur Authentifizierung und Autorisierung auf Ebene der Persistenzschicht vor. Lediglich das Auslesen der Metadaten von `PersistenceCapable`-Instanzen über Klassen und Methoden der JDO-SPI wird durch die Vergabe spezieller Rechte (`JDOPermission`-Objekte) reglementiert, über die ausschließlich eine JDO-Implementierung verfügt.

Die Konstruktion einer `PersistenceManagerFactory`-Instanz erfolgt in der Regel über die statische Methode `getPersistenceManagerFactory(Properties props)` der `JDOHelper`-Klasse (vgl. Abbildung 1). Die Verwendung dieser Methode ermöglicht den Austausch einer JDO-Implementierung, ohne dass eine Änderung am Java-Quellcode einer Anwendung notwendig wird. Dieser Methode werden hierzu Informationen über die verwendete JDO-Implementierung, datenbankspezifische Parameter sowie Benutzerkennung und Passwort übergeben. Nach Aufbau der Datenbankverbindung und Konstruktion des `PersistenceManager`s lassen sich über dessen Schnittstellen beispielsweise `Query`-Objekte zur Anfrage einer Datenbank konstruieren. Die JDO-Spezifikation sieht hierbei keine Einschränkungen von Anfragen oder dem Ausführen weiterer Methoden der JDO-API vor. Nach Konstruktion einer `PersistenceManager`-Instanz kann ein Anwender über dessen Methoden u. a. mit dem Aufruf `getObjectById()` auf beliebige Objekte zugreifen oder mittels `deletePersistence()` persistente Objekte aus dem Hintergrundspeicher löschen. Desweiteren besteht jederzeit die Möglichkeit, aktive Transaktionen zu deaktivieren oder den Object-Cache zu löschen.

Eine Verbesserung dieser Situation lässt sich auf den ersten Blick über unterschiedliche Wege realisieren. Beispielsweise können durch das Setzen datenbankspezifischer Einschränkungen sowie die Vergabe unterschiedlicher Benutzerkennungen erste Sicherheitsrestriktionen realisiert werden. Wird jedem Anwender eine individuelle Benutzerkennung z. B. in einer Datenbank angelegt, lässt sich über diese die Konstruktion unterschiedlicher `PersistenceManagerFactory`-Instanzen realisieren. Ist jedoch davon auszugehen, dass alle Anwender Zugriff auf eine gemeinsame Datenbasis benötigen, müssen entsprechende Zugriffsrechte in der Datenbank individuell gepflegt werden. Bei der Verwendung eines Dateisystems als Persistenzmedium entfällt jedoch diese Möglichkeit. Werden relationale Datenbanksysteme verwendet, lassen sich u. a. über Trigger oder Views die Einschränkungen des Datenbankzugriffs noch feiner realisieren. Die genannten Möglichkeiten haben jedoch alle den entscheidenden Nachteil, dass sie eine starken Abhängigkeit gegenüber der verwendeten Datenbank implizieren. Dies steht jedoch in offensichtlichem Widerspruch zur Intention der JDO-Spezifikation, persistente Transparenz und Datenbankunabhängigkeit zu gewährleisten.

Einen anderen Weg verfolgt das Sicherheitsmodell SecureJDO, welches in Kapitel 3 vorgestellt wird. Aufbauend auf der Java 2 Sicherheitsarchitektur sollen Konstruktion, Zugriff und Destruktion persistenter Objekte mit spezifischen Rechten versehen werden. Hierbei stehen die Kompatibilität zur JDO-Spezifikation, die Unabhängigkeit bezüglich konkret verwendeter JDO-Implementierungen sowie die Verwaltung notwendiger Informationen in einer JDO-Ressource im Vordergrund. Die zum Verständnis von SecureJDO notwendigen Grundlagen der Java 2 Sicherheitsmechanismen werden im folgenden Abschnitt skizziert.

2.2 Die Java 2 Sicherheitsarchitektur

Die Sicherheitsarchitektur in Java basiert auf drei Komponenten: Bytecode Verifier, Class Loader und Security Manager (vgl. Sun Microsystems 2005 und Gong 2002). Der Class Loader lädt Java Klassen und erzwingt Namensraum- und Speichergrenzen. Der Bytecode-Verifier prüft den Code vor Ausführung auf mögliche Verletzungen der Java Sprachregeln, potentielle Adressüberschreitungen und unzulässige Typumwandlungen. Der `SecurityManager`, bzw. der `AccessController` prüft die Ausführung sicherheitsrelevanter Operationen, wie z. B. den Zugriff auf das Dateisystem, das Setzen und Lesen von Systemproperties sowie die Netzwerkkommunikation über Sockets (vgl. Abbildung 2).

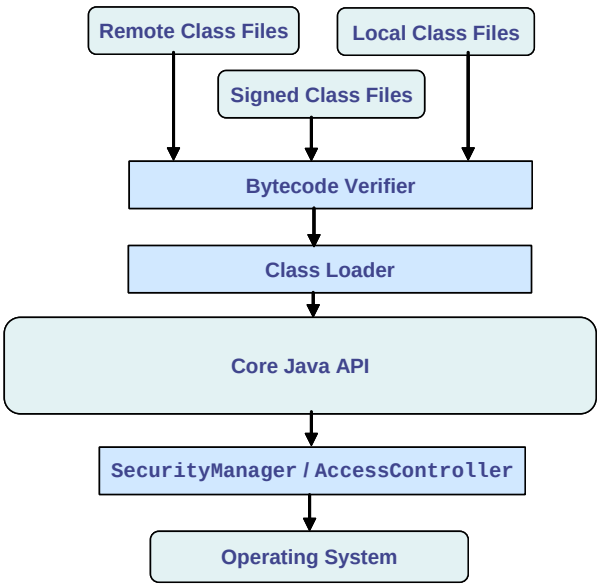

Abbildung 2: Java 2 Sicherheitsarchitektur, in Anlehnung an (Oaks 2001)

Bis Java 1.2 lies sich der Zugriff auf eine Systemressource nur in Abhängigkeit von Autor/Unterzeichner und Herkunft des Bytecodes (lokales Dateisystem vs. entfernte Web-Ressource) regeln (*Code-Centric Authorization*). Erst mit Einführung des Java Authentication and Authorization Service (JAAS) in Java 1.4 ist es möglich, den Zugriff auf Ressourcen in Abhängigkeit vom ausführenden Anwender zu reglementieren (*User-Centric Authorization*). Hierzu folgt zunächst die Authentifizierung eines Anwenders gefolgt von der Phase der Autorisierung.

Die Authentifizierung bezeichnet den Vorgang, die Identität eines Anwenders mit Hilfe bestimmter Merkmale zu überprüfen (siehe Abbildung 3). Neben der Authentifizierung mittels Benutzerkennung und Passwort bieten sich weitere Merkmale und Verfahren zum Nachweis der Identität, wie beispielsweise *Public-Private-Key*-Verfahren auf Basis von Chipkarten oder Prüfungen biometrischer Merkmale an.

Ist die Identität eines Anwenders erfolgreich festgestellt, folgt die Zuordnung und Überprüfung von individuellen Benutzerrechten in der Phase der Autorisierung (siehe Abbildung 4). Vor einem sicherheitskritischen Zugriff auf eine Systemressource werden die Rechte eines

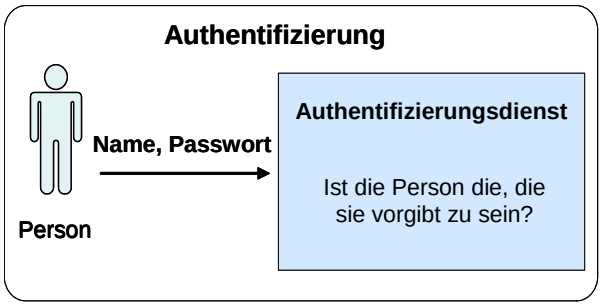

Abbildung 3: Prozess der Authentifizierung

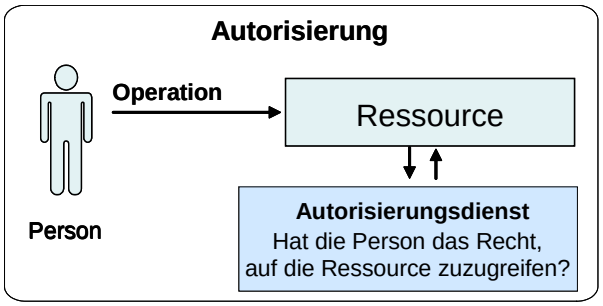

Abbildung 4: Prozess der Autorisierung

Anwenders durch einen Autorisierungsdienst geprüft. In Java übernimmt diese Aufgabe der `SecurityManager`, der die Prüfung der Rechte an den `AccessController` delegiert, welcher dem Anwender den Zugriff auf eine Systemressource individuell gewährt oder verweigert.

Abbildungen 5 und 6 bilden den Vorgang der Authentifizierung und Autorisierung für den JAAS graphisch ab. Bei der JAAS-Authentifizierung erzeugt die Anwendung zunächst einen `LoginContext` und ruft für diese Instanz die Methode `login()` auf. Anhand der Login-Konfiguration delegiert der `LoginContext` die Authentifizierung an ein oder mehrere `LoginModule`s. Diese verwenden `CallbackHandler` zur Kommunikation mit der Anwendung, der Umgebung oder dem Anwender, beispielsweise zur Abfrage einer Benutzerkennung und eines Passworts. JAAS ermöglicht über die Einbindung von PAM-Authentifizierungsmodulen (Pluggable Authentication Modules) die Nutzung weiterer Dienste, mit der die Nutzung von Kerberos, Radius oder LDAP möglich wird. Scheitert ein Authentifizierungsversuch endgültig, wird eine `SecurityException` ausgeworfen.

Bei der JAAS-Autorisierung kann über den in der Authentifizierung-Phase konstruierten `LoginContext` und der Methode `getSubject()` eine `Subject`-Instanz bezogen werden. Diese repräsentiert einen authentifizierten Anwender und ist mit einem oder mehreren `Principal`s (konkrete Rolle eines Anwenders) verknüpft. Eine Anwendung kann über die statische Methode `Subject.doAs(subject, action)` eine Aktion (`PrivilegedAction`) ausführen, bei der zur Laufzeit die Rechte des Anwenders geprüft werden. Dazu stellt der `AccessController` fest, ob mindestens ein dem `Subject` zugeordneter `Principal` die benötigten Rechte zur Ausführung dieser Aktion besitzt. Die Zuordnung der Rechte wird über `Permission`-Objekte realisiert, die in einer Policy-Datei den einzelnen Principals zugeordnet sind.

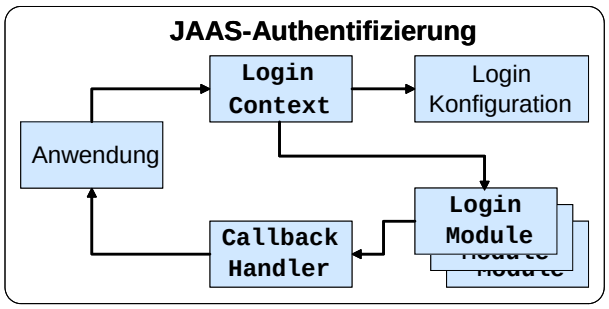

Abbildung 5: JAAS-Authentifizierung

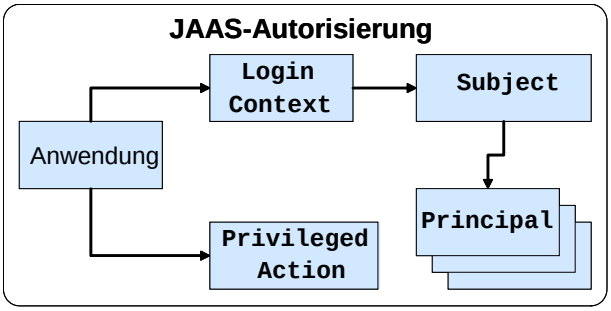

Abbildung 6: JAAS-Autorisierung

3 SecureJDO: Ein Sicherheitsmodell für die JDO-Spezifikation

In diesem Kapitel werden zunächst die Konzepte des Sicherheitsmodells zur Authentifizierung und Autorisierung erläutert. Anschließend wird auf die Möglichkeiten zur Integration von SecureJDO eingegangen.

3.1 Authentifizierung und Autorisierung

Die Konstruktion einer `PersistenceManagerFactory`-Instanz erfolgt, wie in Abschnitt 2.1 deutlich wurde, über die statische Methode `getPersistenceManagerFactory(Properties props)` der `JDOHelper`-Klasse. Die Verwendung dieser Methode ermöglicht den Austausch einer JDO-Implementierung, ohne dass hierzu die Modifikation des Java-Quellcodes einer Anwendung notwendig ist.

Um das Sicherheitsmodell unabhängig von der konkret verwendeten JDO-Implementierung einsetzen zu können, baut die SecureJDO-Architektur auf dem genannten Konzept auf. Eine von `JDOHelper` abgeleitete `JDOSecureHelper`-Klasse bildet den Einstiegspunkt für Anwendungen zur Nutzung des Sicherheitsmodells. Hierzu überschreibt diese Klasse die `getPersistenceManagerFactory()`-Methode (vgl. Abbildung 7). Anhand der übergebenen Benutzerkennung und des Passworts erfolgt die Authentifizierung des Anwenders. Diese werden derzeit noch mit Einträgen einer Konfigurationsdatei abgeglichen, die künftig in eine eigene JDO-Ressource ausgelagert werden sollen. Nach erfolgreicher Authentifizierung wird eine `PersistenceManagerFactory`-Instanz für das in den Properties angegebene Datenbanksystem erzeugt. Hierbei werden die über das `Properties`-Objekt übergebenen Daten wie z. B. Benutzerkennung und das Passwort gegen solche ersetzt, die dem Anwender unbekannt sind. Dieses Verfahren soll den Aufbau einer direkten Verbindung zwischen Anwender und Persistenzmedium und somit eine mögliche Umgehung der `JDOSecureHelper`-Klasse verhindern.

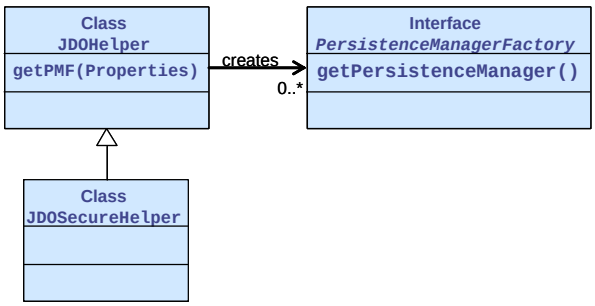

Abbildung 7: Klassendiagramm zur Veranschaulichung der Integration der JDOSecureHelper Klasse in die JDO-API

Zur Realisierung der Authentifizierung wird JAAS verwendet (vgl. Abschnitt 2.2). Abbildung 8 veranschaulicht die Umsetzung der Authentifizierung mit Hilfe eines vereinfachten UML-Klassendiagramms. Über den Methodenaufruf `getPersistenceManagerFactory()` der `JDOSecureHelper`-Klasse wird zunächst eine `LoginContext`-Instanz erzeugt. Diese leitet die Authentifizierung über ein `JDOLoginModule` an einen `JDOCallbackHandler` weiter. Dieser nutzt die in dem `Properties`-Objekt übergebene Benutzerkennung sowie das Passwort zur

Authentifizierung. Sofern dieser Vorgang ohne `SecurityException` abgeschlossen wird, erhält die `JDOSecureHelper`-Instanz über den `LoginContext` den authentifizierten `JDOUser`.

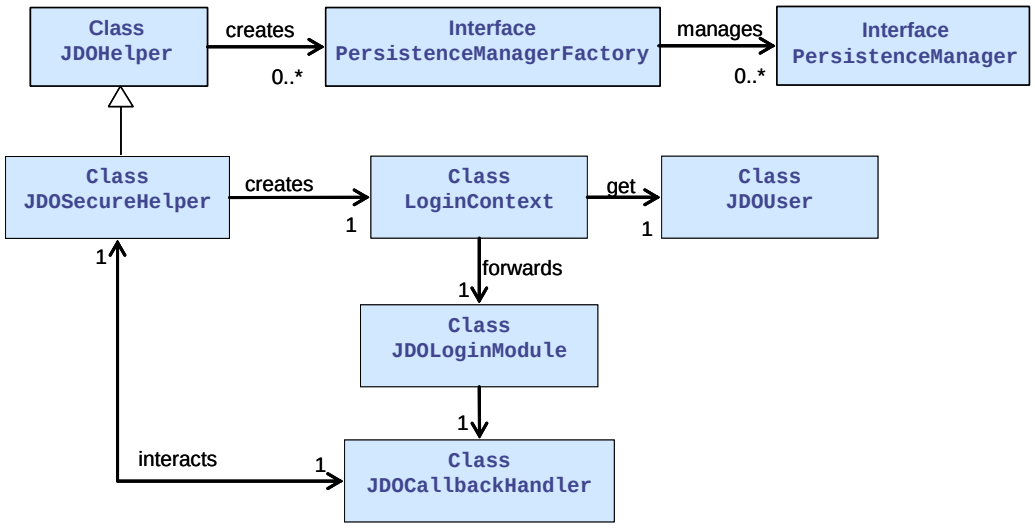

Abbildung 8: Realisierung der Authentifizierung

Nach erfolgreicher Authentifizierung folgt die Phase der Autorisierung. In dieser wird überprüft, ob ein Anwender über die notwendigen Rechte zur Ausführung von Methoden des `PersistenceManager`-Interfaces verfügt. Die Rechte eines Anwenders werden durch Einträge in der Policy-Datei individuell für jeden Methodenaufruf, wie z. B. `makePersistent()`, `newQuery()` und `deletePersistent()` gesetzt. Darüber hinaus lassen sich die Rechte derzeit noch weiter auf ein bestimmtes Paket oder eine konkrete Klasse einschränken. Die Rechte zum Aufruf der Methode `MakePersistent()` lassen sich beispielsweise für ein Paket `sample` und einen Principal „sampleuser“ wie folgt definieren:

```
grant Principal JDOUser ¨sampleuser¨
        permission JDOMakePersistentPermission ¨sample.*¨;
}
```

Offen an dieser Stelle ist zunächst noch, wie SecureJDO zwischen JDO-Implementation und Anwendung zur Prüfung der Benutzerrechte integriert werden kann, ohne eine spezielle JDO-Implementation anpassen zu müssen. Diese Thematik ist Inhalt des folgenden Abschnitts.

3.2 Integration des SecureJDO-Sicherheitsmodells

Wesentliche Voraussetzung für die Akzeptanz des vorgestellten Sicherheitsmodells ist die Unabhängigkeit von einer konkreten JDO-Implementation. Ansätze, die lediglich einzelne Implementationen um Sicherheitsfunktionen erweitern, stellen keine akzeptable Lösung dar. Andererseits sollte ein allgemeiner Ansatz nicht gegen Vorschriften der JDO-Spezifikation verstoßen. Eine Möglichkeit, mit der sich das vorgestellte Sicherheitsmodell in Java entsprechend umsetzen lässt, ist das *Dynamic Proxies*-Konzept (vgl. Abbildung 9).

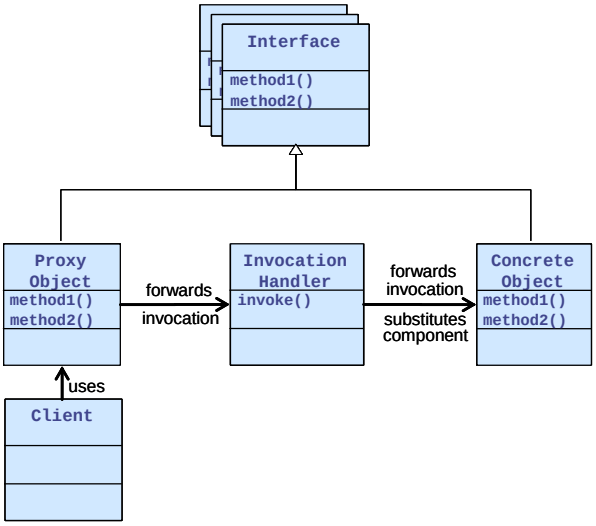

Abbildung 9: Das Konzept der Dynamic Proxies

Ein Proxy bezeichnet ein Stellvertreter-Objekt, das einem anderen Objekt vorgelagert ist. Der Zugriff auf das original Objekt erfolgt ausschließlich über den Proxy, der eingehende Anfragen an dieses weiterleitet. In Java lassen sich solche *statischen Proxies* durch Klassen realisieren, welche die Schnittstellen des original-Objekts implementieren. Ein Nachteil dieses Ansatzes ist das zwingende Vorhandensein einer konkreten Proxy-Klasse zu Beginn der Laufzeit.

Dynamic Proxies gestatten dagegen die Erzeugung einer Proxy-Instanz dynamisch zur Laufzeit (Blosser 2000). Zur Erzeugung eines solchen Proxies wird die statische Methode `newProxyInstance()` der Klasse `java.lang.reflect.Proxy` aufgerufen, welcher ein `ClassLoader`-Objekt, ein Array mit Interfaces sowie ein `InvocationHandler`-Objekt übergeben werden. Eingehende Anfragen werden von der erzeugten Proxy-Instanz an den `InvocationHandler` weitergeleitet, der diese je nach Implementierung entweder selbst bearbeitet oder an das ursprüngliche Objekt delegiert.

Die Einbindung des vorgestellten Sicherheitsmodells in JDO wird, wie Abbildung 10 verdeutlicht, über das Konzept der Dynamic Proxies realisiert. Ausgehend von der `JDOSecureHelper`-Klasse erhält ein Anwender nach Aufruf der `getPersistenceManagerFactory()`-Methode anstelle einer `PersistenceManagerFactory`-Instanz ein korrespondierendes Proxy-Objekt. Über den `PMFInvocationHandler` besteht in diesem Fall die Möglichkeit, Methodenaufrufe, die ursprünglich an die `PersistenceManagerFactory`-Instanz gerichtet sind, abzufangen und gezielt zu manipulieren. Wird über die Proxy-Instanz die Methode `getPersistenceManager()` aufgerufen, wird dem Anwender anstelle eines `PersistenceManager`-Objekts eine weitere Proxy-Instanz zurückgeliefert. Diese Proxy-Instanz leitet auch hier sämtliche Methodenaufrufe an einen `PMInvocationHandler` weiter, über den letztendlich die Rechteprüfung realisiert wird (vgl. Abschnitt 2.1).

Der `PMInvocationHandler` kann für einen zuvor authentifizierten `JDOUser` prüfen, ob dieser über die Rechte zur Ausführung der entsprechenden Methode und einer übergebenen Klasse oder eines konkreten Objekts verfügt. Intern wird dazu über eine `JDOSecurityAction`-Instanz die statische Methode `Subject.doAs(subject, action)` aufgerufen und die eigentliche Prüfung

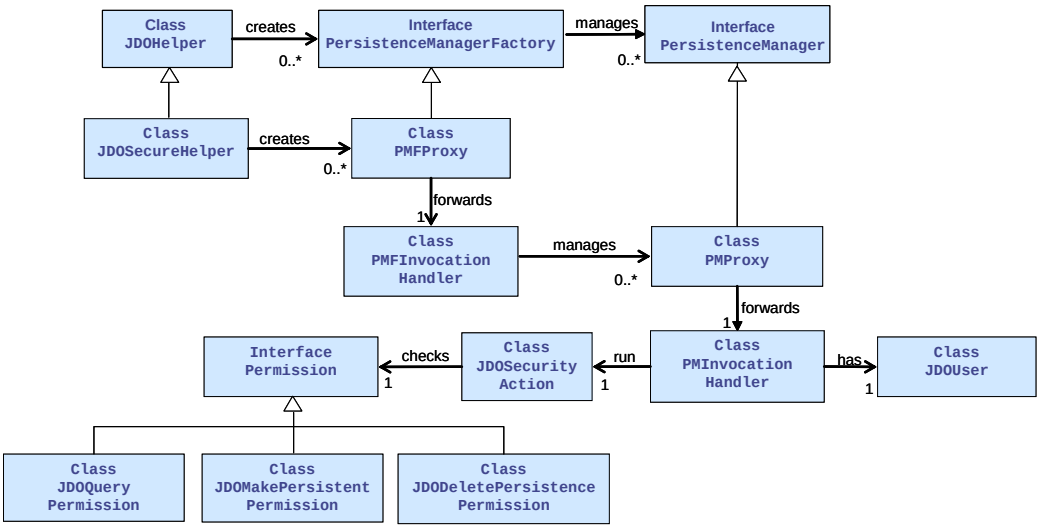

Abbildung 10: Realisierung der Autorisierung

an den AccessController delegiert. Sofern der Anwender über die geforderten Rechte verfügt, wird der Methodenaufruf an die ursprüngliche `PersistenceManager`-Instanz weitergeleitet. Andernfalls wird eine `JavaSecurity`-Exception ausgeworfen und der Vorgang beendet.

4 Offene Punkte

In diesem Artikel wurden die Implementation des JDO-Sicherheitsmodells JDOSecure vorgestellt und die grundlegenden Konzepte zur Realisierung der Authentifizierung und Autorisierung aufgezeigt. Auch wenn bereits in dieser frühen Phase der Implementierung zahlreiche Probleme als gelöst betrachtet werden können, sind derzeit noch einige Punkte offen. Hierzu zählt beispielsweise, wie der JDO Update-Mechanismus in das Sicherheitsmodell integriert werden kann. JDO sieht im Rahmen transparenter Persistenz keine speziellen Methodenaufrufe zum Ändern von Objekt-Attributen vor, an die eine Rechteprüfung gekoppelt werden könnte. Stattdessen werden Änderungen wie bei gewöhnlichen Java-Objekten (POJOs) über ihre *setter*-Methoden direkt vorgenommen. Über einen Update-Vorgang wird eine JDO-Implementierung direkt von der jeweiligen `PersistenceCapable`-Instanz benachrichtigt. Der `StateManager` nimmt daraufhin für den Benutzer transparent die Änderungen auf dem Hintergrundspeicher vor.

Zukünftig soll JDOSecure auch um Funktionen zur Ablage von `Policies` und Benutzer-Anmeldedaten in einer separaten JDO-Ressource erweitert werden. Die Entwicklung eines JDO-basierten Tools für den Security-Administrator zum Konfigurieren der Benutzerrechte ist ebenfalls vorgesehen. Rechte, die sich derzeit ausschließlich an Pakete und Klassen binden lassen, sollen zukünftig auch feingranular auf Objekt-Ebene spezifiziert werden können. Dies würde u. a. die Definition von Rollen gestatten, die Anwendern lediglich die Rechte zum Verändern oder Löschen selbst erzeugter Objekte gewährt.

Hinsichtlich der Vergabe von benutzerspezifischen Zugriffsrechten und deren Überprüfung, erfüllt JDOSecure dagegen bereits die gestellten Anforderungen. Die Rechte lassen sich bezüglich Anwendern und ihrer Rollen individuell definieren und sowohl auf Paket- bzw. Klassen-Ebene

als auch auf einzelne Methoden der JDO-API einschränken.

Darüber hinaus lässt sich JDOSecure durch den Dynamic Proxy-Ansatz ohne Änderungen am Source-Code mit jeder beliebigen JDO-Implementation nutzen. Die Realisierung der Authentifizierung und Autorisierung über JAAS gestattet zudem die Anbindung externer PAM-Authentifizierungsmodule, wodurch die Nutzung weiterer Dienste wie z. B. Kerberos, Radius oder LDAP möglich wird.

Das als *add-on* für beliebige JDO-Implementationen entwickelte Sicherheitsmodell SecureJDO wird mit wachsender Reife dazu beitragen, ein wichtiges Hindernis für den effektiven Einsatz der Java Data Objects-Spezifikation abzubauen und zugleich die Sicherheit von JDO-Persistenz-Architekturen zu erhöhen.

Literatur

Atkinson, Malcolm (2001): *Persistence and Java - A Balancing Act.* In: Proceedings of the International Symposium on Objects and Databases, Springer-Verlag, London, UK, 2001, S. 1–31.

Atkinson, Malcolm; Bancilhon, François; DeWitt, D.; Dittrich, Klaus; Maier, David und Zdonik, Stanley (1989): *The Object-Oriented Database System Manifesto.* In: Proceedings of the First International Conference on Deductive and Object-Oriented Databases (DOOD'89), North-Holland/Elsevier Science Publishers, Kyoto, Japan, 1989, S. 223–240.

BEA Systems, Inc. und IBM Corp. (2005): *Service Data Objects,* 2005, http://ftpna2.bea.com/pub/downloads/Commonj-SDO-Specification-v2.0.pdf.

Bishop, Thomas; Mitchell, Glenn E.; Bell, John; Holm, Bjarki und Ayers, Danny (2001): *Professional Java Data*, 1st edition, Birmingham, Wrox Press Ltd, 2001.

Blosser, Jeremy (2000): *Explore the Dynamic Proxy API,* 2000, http://java.sun.com/developer/technicalArticles/DataTypes/proxy/.

Database for Objects (2005): *Product Information: db4o Open Source Object Database,* 2005, http://www.db4o.com/about/productinformation/db4oV4.5.pdf.

DeMichiel, Linda und Russell, Craig (2004): *A Letter to the Java Technology Community,* 2004, http://java.sun.com/j2ee/letter/persistence.html.

Gong, Li (2002): *Java 2 Platform Security Architecture,* 2002, http://java.sun.com/j2se/1.4.2/docs/guide/security/spec/security-spec.doc.html.

Hibernate (2005): *Hibernate Reference Documentation,* 2005, http://www.hibernate.org/hib_docs/v3/reference/en/pdf/hibernate_reference.pdf.

Java Community Process (2003a): *JSR 153: Enterprise JavaBeans 2.1,* 2003, http://www.jcp.org/en/jsr/detail?id=153.

Java Community Process (2003b): *JSR 235: Service Data Objects,* 2003, http://www.jcp.org/en/jsr/detail?id=235.

Java Community Process (2004): *JSR-012: Java Data Objects (JDO) Specification, (Maintenance Draft Review),* 2004, http://www.jcp.org/en/jsr/detail?id=12.

Java Community Process (2005a): *JSR 220: Enterprise JavaBeans 3.0, (Public Review),* 2005, http://www.jcp.org/en/jsr/detail?id=220.

Java Community Process (2005b): *JSR 243: Java Data Objects 2.0 - An Extension to the JDO specification,* 2005, http://www.jcp.org/en/jsr/detail?id=243.

Korthaus, Axel und Merz, Matthias (2003): *A Critical Analysis of JDO in the Context of J2EE.* In: Al-Ani Ban; H. R. Arabnia und Mun Youngsong (Hrsg.): Proceedings of the 2003 International Conference on Software Engineering Research and Practice (SERP '03, Volume I), CSREA Press 2003, S. 34–40.

Oaks, Scott (2001): *Java Security.* The Java Series, Second, Sebastopol, CA, USA, O'Reilly & Associates, Inc., 2001.

Object Data Management Group (2000): *ODMG 3.0,* 2000, http://www.odmg.org/.

Roos, Robin (2002): *Java Data Objects*, 1st edition, London u. a., Addison-Wesley, Pearson Education, 2002.

Schäfer, Alexandra (2003): *Der „Impedance Mismatch": Im Spannungsfeld zwischen objektorientiertem und relationalem Ansatz, Objektspektrum*, 2003, *3*, S. 33–36, http://www.intersystems.de/pdf/objektspektrum_0303.pdf.

Schmidt, Duri (1991): *Persistente Objekte und Objektorientierte Datenbanken*, München, Wien, Hanser-Verlag, 1991.

SourceForge (2005): *Simple Object Database Access,* 2005, http://sodaquery.sourceforge.net/.

Sun Microsystems (2005): *The Java Language Specification*, Third Edition, Addison-Wesley Professional, 2005, http://java.sun.com/docs/books/jls/.

TheServerSide.COM (2001): *Craig Russell Responds to Roger Sessions' Critique of JDO,* 2001, http://www.theserverside.com/articles/article.tss?l=RusselvsSessions.

TheServerSide.COM (2003): *A criticism of Java Data Objects (JDO),* 2003, http://www.theserverside.com/news/thread.tss?thread_id=8571.