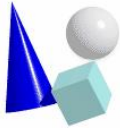


# JDOSecure: Eine Sicherheitsarchitektur für die Java Data Objects-Spezifikation

Dipl.-Wirtsch.-Inf. Matthias Merz

08. Dezember 2005

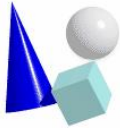




# Agenda

1. Aktuelle Trends
2. Motivation
3. JDOSecure: Bisheriger Stand
4. Änderung von Objektattributen
5. Performance Messung
6. Fazit und weiteres Vorgehen

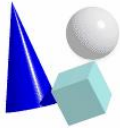




## Aktuelle Trends

- Hersteller von Persistenzlösungen gehen vermehrt dazu über, in ihrer Software sowohl den JDO 2.0 als auch EJB 3.0 Standard zu implementieren. Bsp.:
  - SolarMetric: Kode 4.0
  - Xcalia: Xcalia Core
  - Versant: Eclipse JSR220-ORM
  - JPOX (Open Source)
- Übernahme von SolarMetric durch BEA Systems
  - BEA Systems: J2EE-Server WebLogic, Workshop Development Tool
  - SolarMetric: Spezialisiert auf Datenbank-Entwickler-Tools, JDO-Implementation Kodo

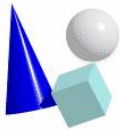




# Agenda

1. Aktuelle Trends
2. Motivation
3. JDOSecure: Bisheriger Stand
4. Änderung von Objektattributen
5. Performance Messung
6. Fazit und weiteres Vorgehen



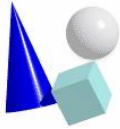


# Motivation

„Sicherheitsaspekte“ in der JDO-Spezifikation:

- Zugriff auf persistente Objekte nur durch Datenbank Authentifizierung eingeschränkt.
- Alle Anwender/Anwendungen verwenden i.d.R. eine gemeinsame Datenbank-Kennung.
- Nach Anmeldung an einer Ressource können über die Methoden des **PersistenceManagers** Anwender beispielsweise mittels
  - **makePersistent (...)** beliebig persistente Objekte erzeugen
  - **getObjectById (...)** , **getExtent (...)** , **newQuery (...)** auf beliebige Objekte der Datenbank zugreifen
  - **deletePersistence (...)** sämtliche Objekte löschen.
- Die Änderung von Objekt-Attributen ist ebenfalls uneingeschränkt möglich.

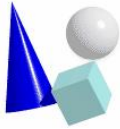




# Agenda

1. Aktuelle Trends
2. Motivation
3. JDOSecure: Bisheriger Stand
4. Änderung von Objektattributen
5. Performance Messung
6. Fazit und weiteres Vorgehen





# JDOSecure

Ziel:

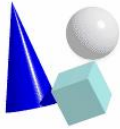
Einschränkung der Zugriffsrechte bei der Nutzung der JDO-API hinsichtlich

- Benutzer bzw. ihren Rollen,
- Klassen und Paketen,
- Funktionalen Aspekte: CRUD (Create, Read, Update, Delete).

Nebenbedingungen:

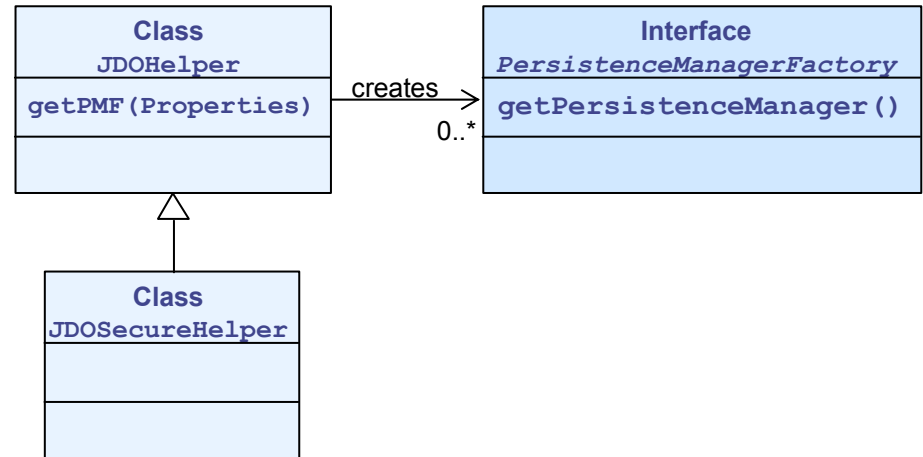
- Kompatibilität zur JDO-Spezifikation.
- Unabhängigkeit bezüglich der verwendeten JDO-Implementierung.
- Verwaltung der notwendigen Informationen in einer beliebigen JDO-Ressource (z. B. Datenbank oder LDAP-Server).
- Keine Möglichkeit zur Umgehung von JDOSecure durch direkte Nutzung der JDO-API.





## Bisheriger Stand

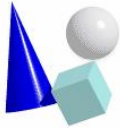
- Eine von `JDOHelper` abgeleitete Klasse (`JDOSecureHelper`) bildet den Einstiegspunkt.
- Diese überschreibt die Methode



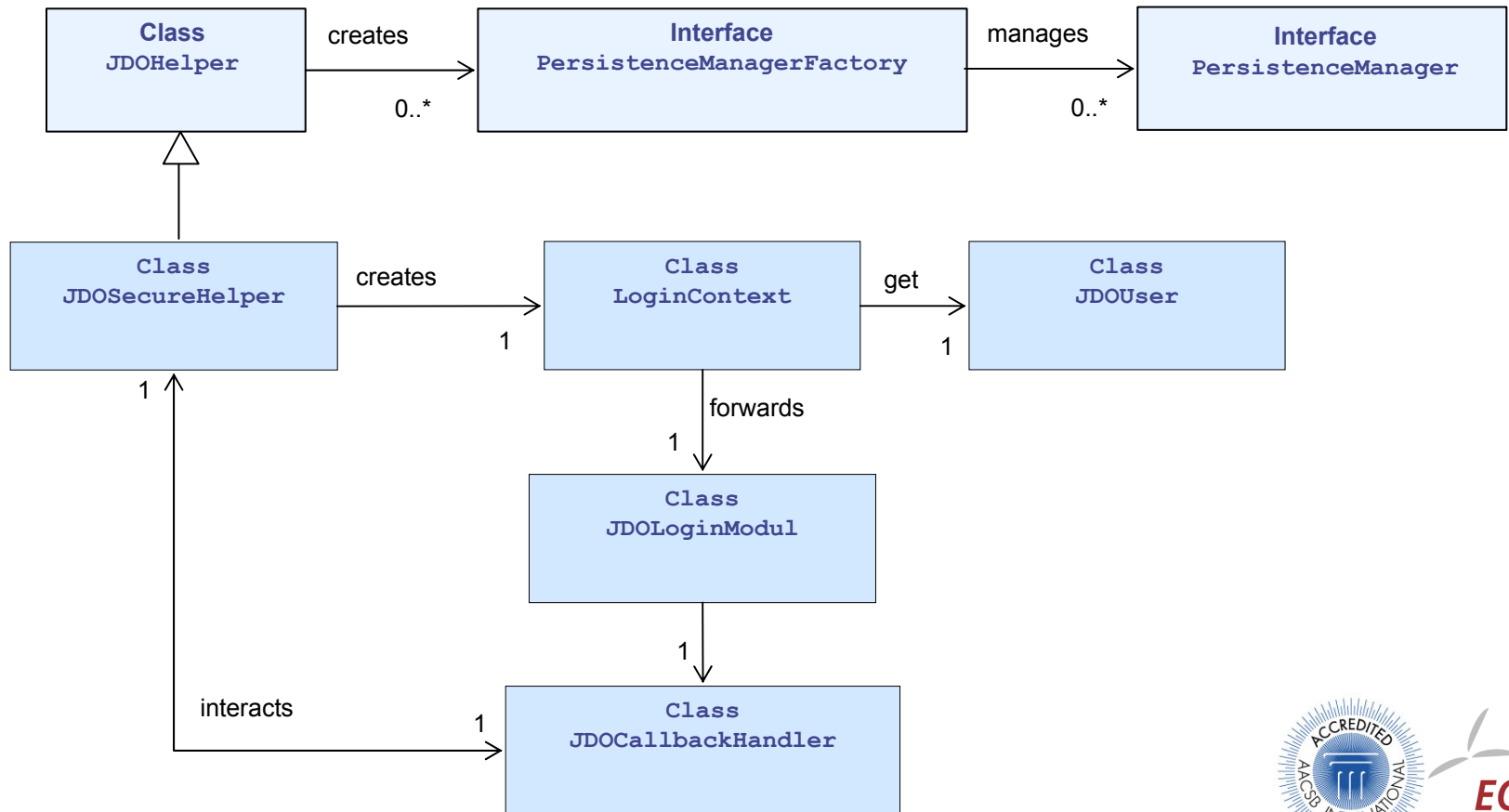
`getPersistenceManagerFactory(Properties props):`

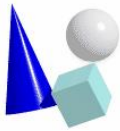
- Anhand der übergebenen `Properties` (u. a. Benutzername und Passwort) erfolgt die Authentifizierung.
- Falls erfolgreich, wird eine `PersistenceManagerFactory` erzeugt.
- Zuvor erfolgt jedoch der Austausch von Benutzername und Passwort; dies soll die Umgehung von `JDOSecure` verhindern.



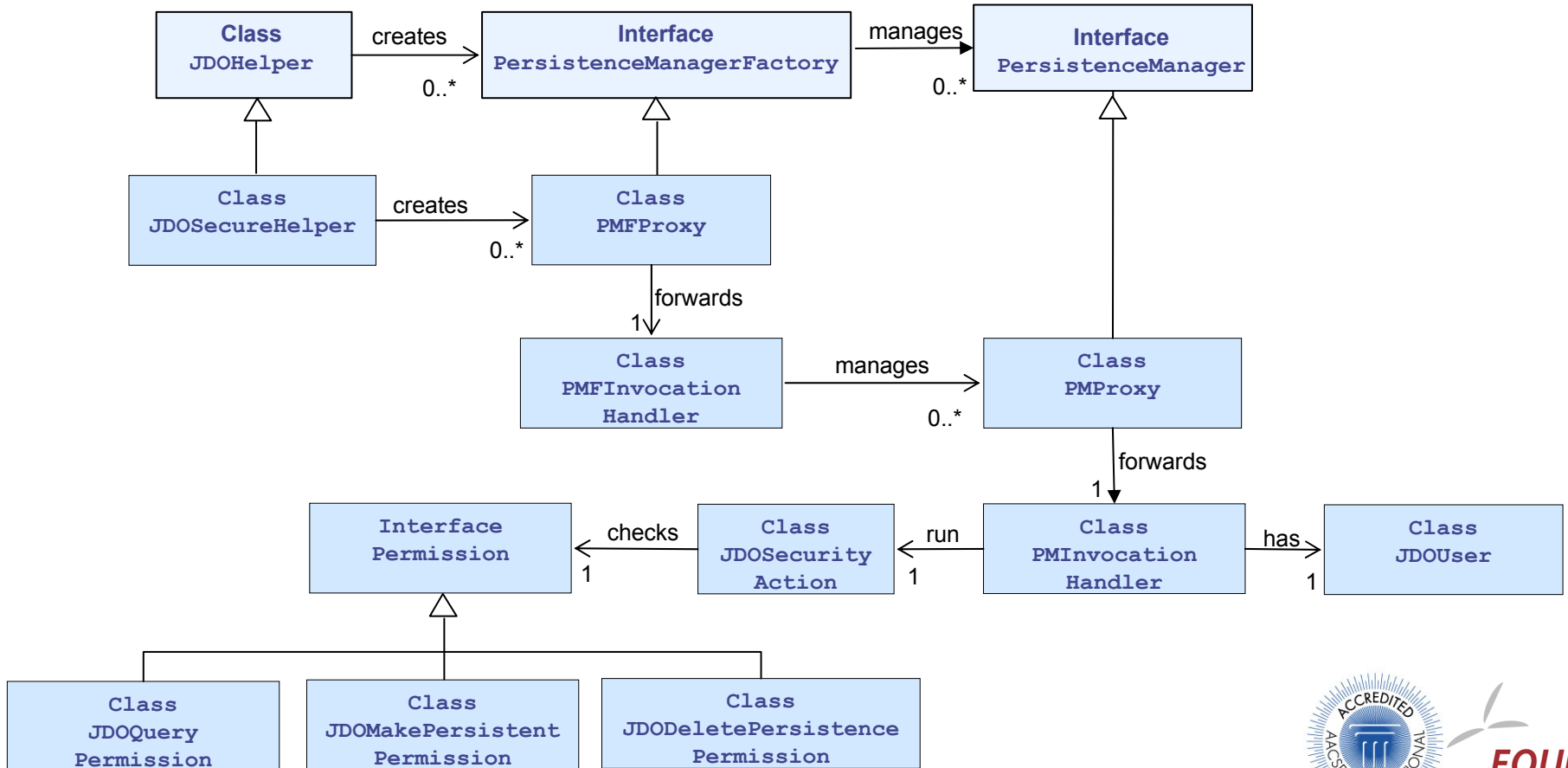


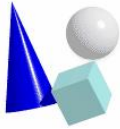
# Realisierung der Authentifizierung





# Realisierung der Autorisierung





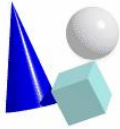
# JDOSecure

- Bisher definierte Permissions:

Methoden des PersistenceMangerS / Query:	Notwendige Permissions:
<code>makePersistent(..)</code> <code>makePersistentAll(..)</code>	<code>JDOMakePersistentPermission &lt;Klasse&gt;</code>
<code>deletePersistent(..)</code> <code>deletePersistentAll(..)</code>	<code>JDODeletePersistencePermission &lt;Klasse&gt;</code>
<code>getExtent(..)</code> <code>Query.execute(..)</code>	<code>JDOQueryPermission &lt;Klasse&gt;</code>

- Neu: `JDOUpdatePermission <Klasse>`

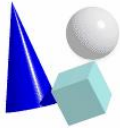




# Agenda

1. Aktuelle Trends
2. Motivation
3. JDOSecure: Bisheriger Stand
4. Änderung von Objektattributen
5. Performance Messung
6. Fazit und weiteres Vorgehen

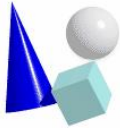




## Änderung von Objektattributen

- Bei JDO ergeben sich aus Sicht der Anwender keine Änderungen der Schnittstelle zum Objektsystem (*transparente Persistenz*).
- Wird z. B. einem Objektattribut ein neuer Wert zugewiesen, erfolgt ein Datenbank-Update implizit nach positiver Beendigung einer Transaktion.

```
tx.begin();  
...  
Extent e      = pm.getExtent(Product.class, false);  
Iterator it = e.iterator();  
...  
product = (Product) iterator.next();  
product.setPrice(9.99);  
...  
tx.commit();
```



# Änderung von Objektattributen

- Durch den Enhancement-Prozess wird der Bytecode der Klasse `Product` beispielsweise wie folgt geändert:

```
public void setPrice(double price) {  
    this.price = price;  
}
```

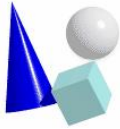
Original

```
public void setPrice(double price) {  
    jdoSetprice(this, price);  
}
```

Nach dem Enhancement Vorgang

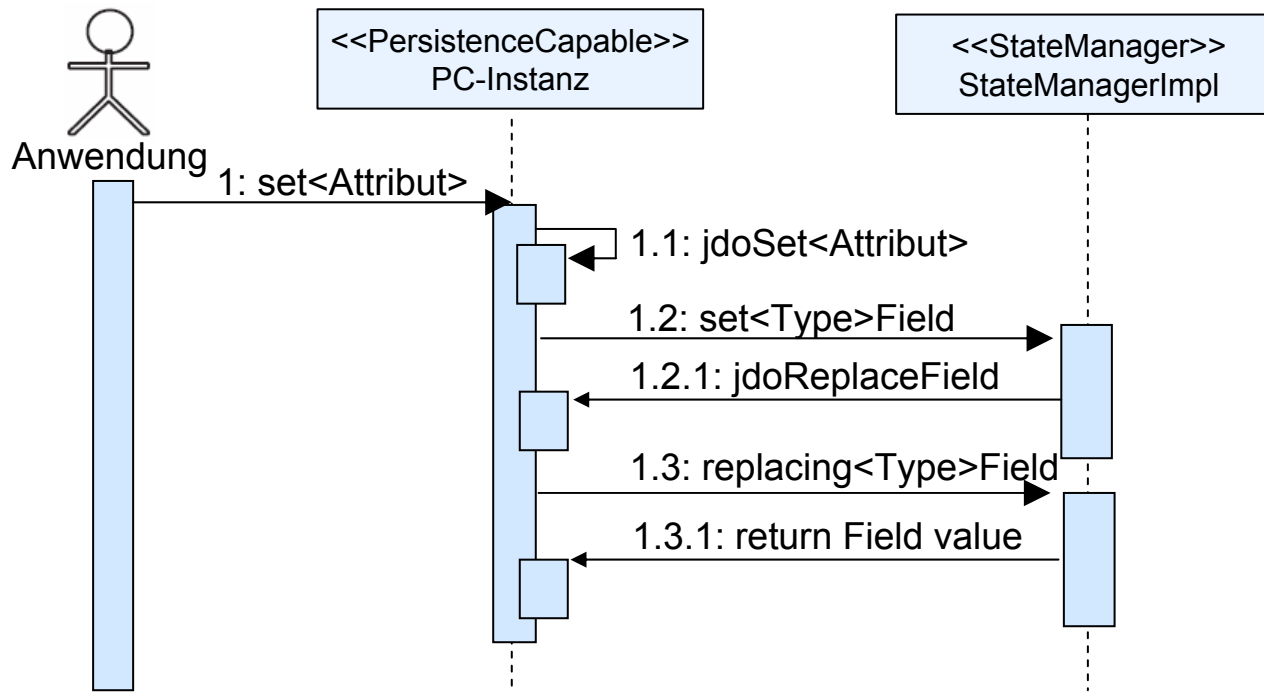
- Die `set<Attribut>` Methoden werden so abgewandelt, dass nun ein `StateManager` über die Änderung von Objektattributen informiert wird.
- In der `PersistenceCapable` Instanz werden diese Änderungen nicht automatisch übernommen.

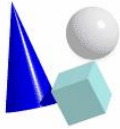




# Änderung von Objektattributen

- Ein `PersistenceCapable` Objekt ist mit einem `StateManager` Objekt verbunden. Bei Änderungen wird dieser benachrichtigt, der die Änderungen an die Datenbank automatisch propagiert.



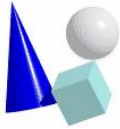


# Änderung von Objektattributen

Mögliche Ansätze zur Realisierung der Überprüfung von Update-Rechten:

- Veränderung des Binärcodes von `PersistenceCapable` Objekten
  - Vorteil: Gute Performance zur Laufzeit
  - Nachteil: Spezieller Enhancer, nicht JDO Standardkonform, unsicher
- Abfangen der `commit()` Methode
  - Vorteil: Binärcode wird nicht verändert, standardkonform
  - Nachteil: Welche Objekte wurden in einer Transaktion geändert?
- Austausch des `stateManagers` durch einen Proxy
  - Vorteil: Binärcode wird nicht verändert, standardkonform
  - Nachteil: Geringere Performance, wie erfolgt der Austausch?



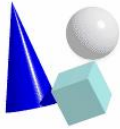


# Änderung von Objektattributen

Setzen des `stateManagerS` in der `PersistenceCapable` Klasse:

```
public synchronized void jdoReplaceStateManager(StateManager sm) {
    if (jdoStateManager != null){
        jdoStateManager = jdoStateManager.
            replacingStateManager(this, sm);
    }
    else{
        SecurityManager securityManager =
            System.getSecurityManager();
        if(securityManager != null) securityManager.
            checkPermission(
                JDOPermission.
                SET_STATE_MANAGER);

        jdoStateManager = sm;
    }
}
```

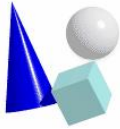


# Änderung von Objektattributen

Austausch des `stateManager`s durch einen Proxy:

- Die Erzeugung einer `stateManager` Instanz obliegt ausschließlich der jeweiligen JDO-Implementierung.
  - Zur Erzeugung des `dynamic proxies` muss bereits eine `stateManager` Instanz erzeugt worden sein.
  - Dies ist der Fall nach dem Aufruf der Methoden des `PersistenceManager`s
    - `makePersistent (...)`, `makePersistentAll (...)`
    - `getExtent (...)`
    - `getObjectById (...)`
    - sowie der `execute (...)` Methode von `query` Instanzen.
- ➔ Daher erfolgt der Austausch nach Aufruf dieser Methoden im `PMInvocationHandler` bzw. `QueryInvocationHandler`.





# Änderung von Objektattributen

Problem 1:

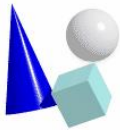
- Wie lässt sich der aktuelle `StateManager` einer `PersistenceCapable` Klasse auslesen? – Das Feld `jdoStateManager` ist `private`.

Lösung:

- Über die `java.lang.reflection` API

```
Class c = obj.getClass();  
  
Field      jdoStateManagerField = c.getDeclaredField("jdoStateManager");  
StateManager jdoStateManagerField.setAccessible(true);  
  
jdoStateManagerValue = (StateManager) jdoStateManagerField.get(obj);
```

(Nur möglich, wenn die Permission `suppressAccessChecks` gesetzt ist)



# Änderung von Objektattributen

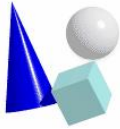
## Problem 2:

- Die JDO-Implementierung kann jederzeit wieder die Methode `jdoReplaceStateManager (...)` aufrufen und den Proxy ersetzen.

## Lösung:

- Durch die entsprechenden Methoden des `PersistenceManagerS` wurde bereits der `StateManager` durch den Proxy ersetzt.
- Im zugehörigen `InvocationHandler` muss daher nur die Methode `replacingStateManager (...)` abgefangen werden.
- Hier erfolgt die erneute Konstruktion eines dynamic proxy, der an den `StateManagerS` zur Ersetzung übergeben wird.



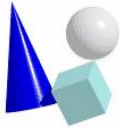


# Änderung von Objektattributen

Fazit:

- Nach dem Aufruf der Methoden des `PersistenceManagerS`
  - `makePersistent(...)`, `makePersistentAll(...)`
  - `getExtent(...)`
  - `getObjectById(...)`
  - sowie der `execute(...)` Methode von `query` Instanzen wird bei den `PersistenceCapable` Instanzen der zugeordnete `StateManager` durch einen Proxy ausgetauscht.
- Konkret erfolgt der Austausch nach Aufruf dieser Methoden im `PMInvocationHandler` bzw. `QueryInvocationHandler`.
- Der Anwender bzw. die Anwendung bekommt davon nichts mit.

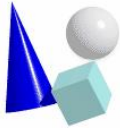




# Agenda

1. Aktuelle Trends
2. Motivation
3. JDOSecure: Bisheriger Stand
4. Änderung von Objektattributen
5. Performance Messung
6. Fazit und weiteres Vorgehen



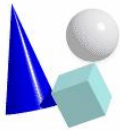


# JDOSecure: Performance Messung

CRUD-Analyse basierend auf

- dem Paket: `org.jpox.samples.store`
  - Klasse `Product` (`getPrice()`, `setPrice()`, ...)
  - 7 Subklassen, darunter:
    - Klasse `Book` (`getIsbn()`, `getAuthor()`, ...)
    - Klasse `CompactDisc` (`getArtist()`, ...)
- der JDO-Implementierung Xcalia Core 4.0.1 build 836,
- MySQL 5.0.15-nt,
- einer Beispiel-Anwendung, die jede Operation mit 500, 5.000 und 20.000 Objekten testet.





# JDOSecure: Performance Messung

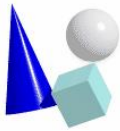
## CRUD-Analyse: Create

`pm.makePersistent (product) ;`

Create / JDO	T1	T2	T3	Mittelwert	Varianz [s <sup>2</sup> ]	Zeitlicher Overhead [%]
500 Obj.	5,078	5,047	5,047	5,06	0,00	
5.000 Obj.	22,75	22,079	22,016	22,28	0,17	
20.000 Obj.	78,953	79,047	77,75	78,58	0,52	

Create / JDOSecure						
500 Obj.	6,328	6,344	6,375	6,35	0,00	25,5
5.000 Obj.	28,766	28,484	28,532	28,59	0,02	28,3
20.000 Obj.	98,187	98,000	98,187	98,12	0,01	24,9

(Jeweils 500 Objekte sind einer Transaktion zugeordnet, 2/5 der Objekte sind vom Typ `Product`, 2/5 `Book` und 1/5 `CompactDisc`)



# JDOSecure: Performance Messung

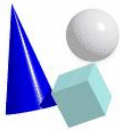
## CRUD-Analyse: Read

```
e = pm.getExtent(org.jpox.samples.store.Product.class, true);
q = pm.newQuery(e, "price < 150.00");
q.setOrdering("price ascending");
```

Read / JDO	T1	T2	T3	Mittelwert	Varianz [s <sup>2</sup> ]	Zeitlicher Overhead [%]
500 Obj.	3,625	3,547	3,641	3,60	0,00	
5.000 Obj.	23,125	22,562	23,14	22,94	0,11	
20.000 Obj.	104,703	103,563	102,203	103,49	1,57	

Read / JDOSecure						
500 Obj.	3,860	3,812	3,828	3,83	0,00	6,4
5.000 Obj.	27,390	27,360	27,422	27,39	0,00	19,4
20.000 Obj.	122,172	120,375	120,484	121,01	1,02	16,9

(Es werden alle Produkte gesucht, für die gilt: price < 150)



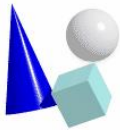
# JDOSecure: Performance Messung

## CRUD-Analyse: Update

```
Extent e = pm.getExtent(Product.class, true);
while(iterator.hasNext()) {
    product = (Product) iterator.next();
    product.setPrice(product.getPrice()*1.05);
}
```

Update / JDO	T1	T2	T3	Mittelwert	Varianz [s <sup>2</sup> ]	Zeitlicher Overhead [%]
500 Obj.	5,61	5,625	5,765	5,67	0,01	
5.000 Obj.	44,109	42,797	42,844	43,25	0,55	
20.000 Obj.	168,3	171,84	169,61	169,92	3,21	

Update / JDOSecure						
500 Obj.	7,015	6,953	7,062	7,01	0,00	23,7
5.000 Obj.	50,328	50,328	50,156	50,27	0,01	16,2
20.000 Obj.	196,08	194,28	194,78	195,05	0,86	14,8



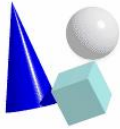
# JDOSecure: Performance Messung

## CRUD-Analyse: Delete

```
Extent e = pm.getExtent(Product.class, true);
while(iterator.hasNext()) {
    product = (Product) iterator.next();
    pm.deletePersistent(product);
}
```

Delete / JDO	T1	T2	T3	Mittelwert [s]	Varianz [s <sup>2</sup> ]	Zeitlicher Overhead [%]
500 Obj.	1,047	1,016	1,094	1,05	0,00	
5.000 Obj.	9,297	9,219	9,531	9,35	0,03	
20.000 Obj.	35,656	35,235	34,922	35,27	0,14	

Delete / JDOSecure	T1	T2	T3	Mittelwert [s]	Varianz [s <sup>2</sup> ]	Zeitlicher Overhead [%]
500 Obj.	1,844	1,875	1,828	1,85	0,00	75,7
5.000 Obj.	13,797	13,859	13,812	13,82	0,00	47,9
20.000 Obj.	52,859	52,14	52,359	52,45	0,14	48,7

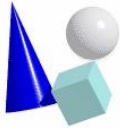


# JDOSecure: Performance Messung

## Fazit:

- Sicherheitsprüfung hat wie erwartet negative Auswirkung auf die Performance.
- Mit Ausnahme des Delete-Vorgangs sind die Performanceeinbußen  $\leq 28\%$ .
- Bisher wurde JDOSecure nicht bezüglich Geschwindigkeit optimiert.

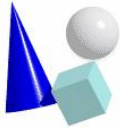




# Agenda


1. Aktuelle Trends
2. Motivation
3. JDOSecure: Bisheriger Stand
4. Änderung von Objektattributen
5. Performance Messung
6. Fazit und weiteres Vorgehen



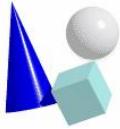


## Fazit

Die vorgestellte JDOSecure Architektur erfüllt die gestellten Anforderungen:

- Einschränkung der Zugriffsrechte bei der Nutzung der JDO-API hinsichtlich
  - Benutzer bzw. ihren Rollen
  - Klassen und Paketen
  - Funktionalen Aspekte: CRUD (Create, Read, Update, Delete) 
- JDO-Kompatibilität
- JAAS gestattet Anbindung beliebiger PAM-Authentifizierungsmodule
- Integration über Dynamic Proxies ermöglicht die Einbindung in jede JDO-Implementierung.

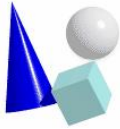




## Weiteres Vorgehen

- Ablage der Policies und Benutzer-Anmeldedaten in eine separate JDO-Ressource.
- Login-Modul und Tool zum konfigurieren der Benutzerrechte.
- Evt. Verwendung von MD5-Hashes/Digest Authentication.
- Erweiterung der Zugriffsrechte auf Objekt-Ebene, z. B.:  
„Nur ein Anwender, der ein Objekt erzeugt hat, darf es verändern und löschen“.
- Logging





# Danke für die Aufmerksamkeit!

Gibt es Fragen?

