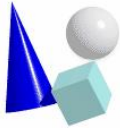


# Ansätze zur Integration eines Sicherheitsmodells in Java Data Objects

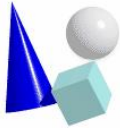
Dipl.-Wirtsch.-Inf. Matthias Merz



# Agenda

1. Motivation
2. Java 2 Sicherheitsarchitektur
3. JDO Security Model
4. Live Demo
5. Fazit und weiteres Vorgehen





# Java Data Objects - Grundlagen

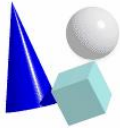
"The Java Data Objects (JDO) API is a standard interface-based Java model abstraction of persistence, developed as Java Specification Request 12 under the auspices of the Java Community Process. Application programmers use JDO to directly store their Java domain model instances into the persistent store (database)."

Quelle: Sun

Ziele:

- Transparente Persistenz
- Datenbankunabhängigkeit
- Transaktionale Semantik
- Interoperabilität

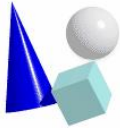




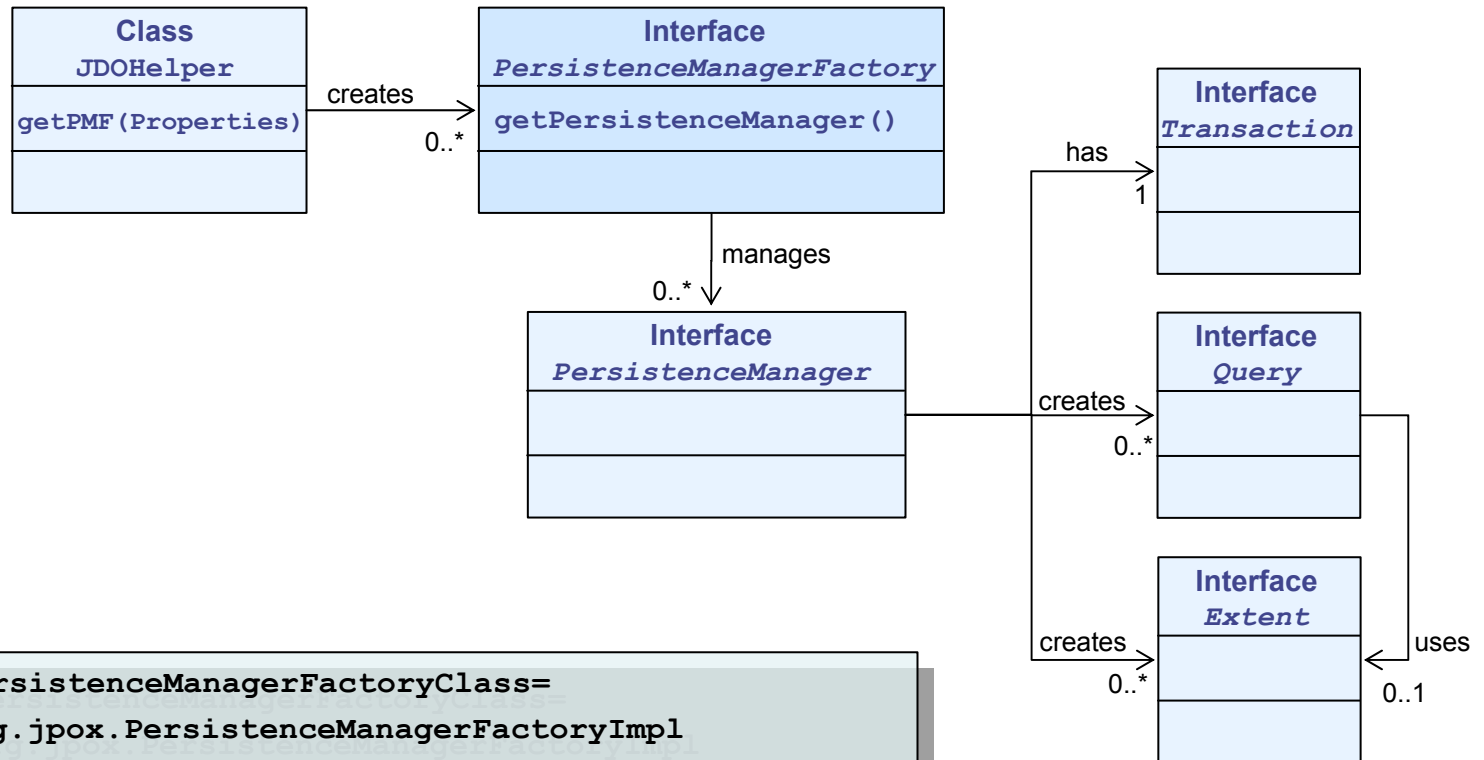
# JDO-API

- *persistence-capable* (persistenzfähige Klassen)
  - Implementieren das `PersistenceCapable`-Interface
- *transient* (nicht persistenzfähig)
  - Klassen, die native Aufrufe verwenden
  - Einige System-Klassen in `java.lang`, `java.io`, `java.net` (z. B.: `Socket`, `Exception`, `Thread`)
- *persistence-aware*
  - Selbst nicht persistenzfähig
  - Greifen direkt auf `public` bzw. `protected` Attribute einer `PersistenceCapable`-Instanz zu





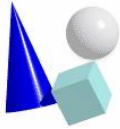
# JDO-API



```

javax.jdo.PersistenceManagerFactoryClass=
    org.jpox.PersistenceManagerFactoryImpl
javax.jdo.option.ConnectionDriverName=com.mysql.jdbc.Driver
javax.jdo.option.ConnectionURL=jdbc:mysql://localhost/jpox
javax.jdo.option.ConnectionUserName=merz
javax.jdo.option.ConnectionPassword=*****
    
```

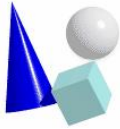




# Sicherheitsaspekte in der JDO-Spezifikation

- Sicherheit nur bezüglich des Auslesens der Metadaten von **PersistenceCapable** Klassen gewährleistet
- Nach Anmeldung an einer Ressource können über die Methoden des **PersistenceManagers** Anwender beispielsweise mittels
  - **getObjectById(...)** auf beliebige Objekte zugreifen bzw. mit
  - **deletePersistence(...)** beliebige persistente Objekte löschen
- Zudem besteht prinzipiell die Möglichkeit
  - Laufende (auch fremde) Transaktionen zu deaktivieren
  - Den gesamten Object-Cache zu löschen

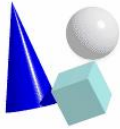




# Sicherheitsaspekte in der JDO-Spezifikation

- Keine Authentifizierung/Autorisierung auf Ebene der Persistenzschicht
  - Alle Anwender/Anwendungen nutzen eine gemeinsame Benutzerkennung zur Ausführung sämtlicher JDO-Operationen
  - Bei erfolgreicher Authentifizierung an der Ressource: Jeder darf alles
  - Werden datenbankspezifische Einschränkungen genutzt (unterschiedliche Benutzerkennungen, spezielle Zugriffsrechte für Tabellen, Trigger, etc.) besteht die Abhängigkeit zu einem konkreten Datenbankprodukt
    - Widerspruch zur Intention von JDO: Datenbankunabhängigkeit!





# JDO Security Model

Ziel:

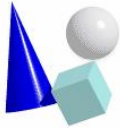
Einschränkung der Zugriffsrechte bei der Nutzung der JDO-API hinsichtlich

- Benutzer bzw. ihren Rollen
- Paketen, Klassen und Objekten
- 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)
- Security Model soll keine Möglichkeit zur Umgehung bieten

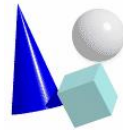




# Agenda

1. Motivation
2. Java 2 Sicherheitsarchitektur
3. JDO Security Model
4. Live Demo
5. Fazit und weiteres Vorgehen

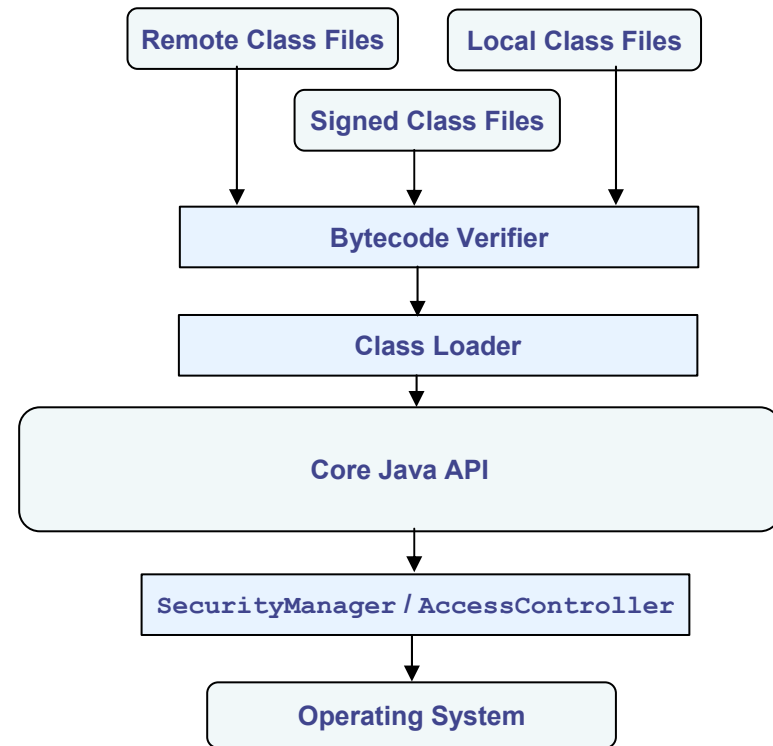




# Java 2 Sicherheitsarchitektur

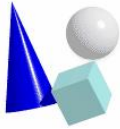
## Grundlagen:

- Sicherheitsarchitektur in Java basiert auf:
  - Bytecode Verifier
  - Class Loader
  - Security Manager
- Java Anwendungen werden i.d.R. ohne **SecurityManager** ausgeführt
- Aktivierbar mittels:
  - Djava.security.manager**



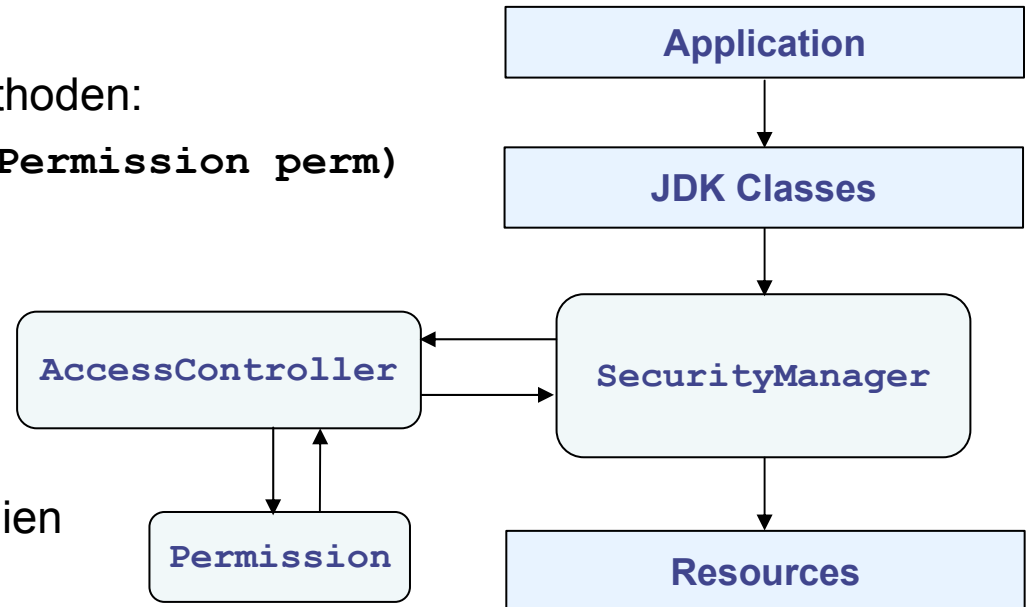
In Anlehnung an S. Oaks:  
"Java Security", O'Reilly

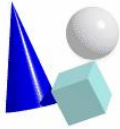




## Java 2 Sicherheitsarchitektur

- AccessController:
  - Sammlung statischer Methoden:  
z.B. `checkPermission(Permission perm)`
- Permission:
  - Berechtigen zur Ausführung einer bestimmten Operation
  - Werden in `.policy`-Dateien gespeichert
  - Sind verknüpft mit dem Herkunftsort des Bytecodes (*Code Sources*) und/oder einer digitalen Signatur



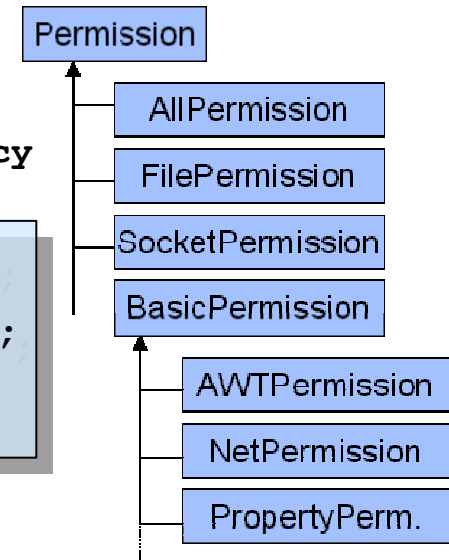


## Java 2 Sicherheitsarchitektur

Beispiel: Aufruf mit

```
-Djava.security.manager -Djava.security.policy=test.policy
```

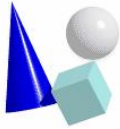
```
Permission p = new FilePermission("/tmp", "read");  
SecurityManager s = System.getSecurityManager();  
if (s != null) s.checkPermission(p);
```



test.policy:

```
grant codeBase "file:/home/merz"{  
    permission java.io.FilePermission "/tmp", "read";  
};
```

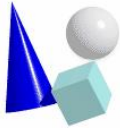




## Java 2 Sicherheitsarchitektur

- Bis JDK 1.2: "Code-Centric Authorization"
  - Zugriff auf Ressourcen basiert auf Herkunft und Verfasser des Programmcodes:
    - Woher wurde der Code geladen?
    - Wer hat den Code signiert?
- Ab JDK 1.4: Erweiterung um Benutzerorientierte Sicherheit:
  - Java Authentication and Authorization Service (JAAS) ermöglicht einen benutzerbasierten Zugriff auf Ressourcen:
    - Unter wessen Namen läuft der Code?





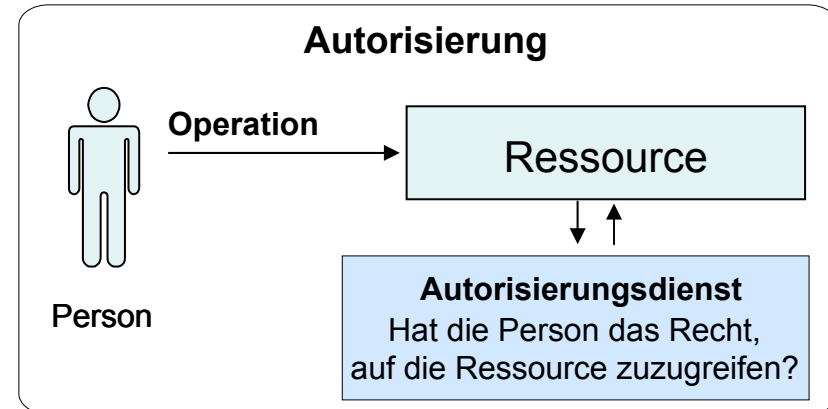
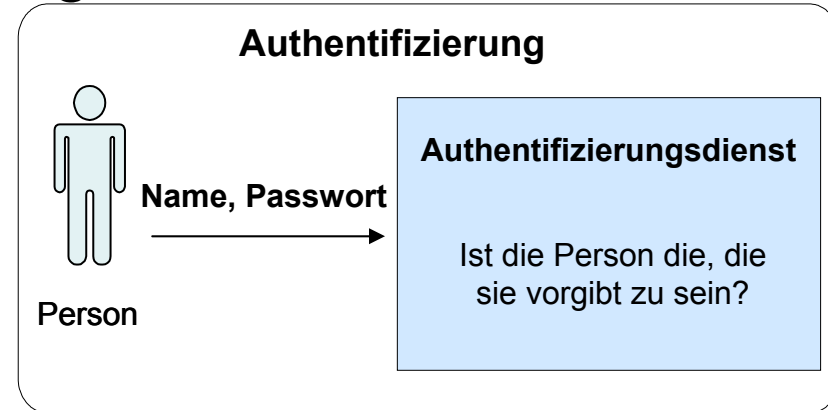
# Authentifizierung vs. Autorisierung

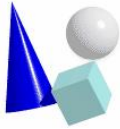
## Authentifizierung:

- Vorgang, der die Identität einer Person mittels eines bestimmten Merkmals überprüft

## Autorisierung:

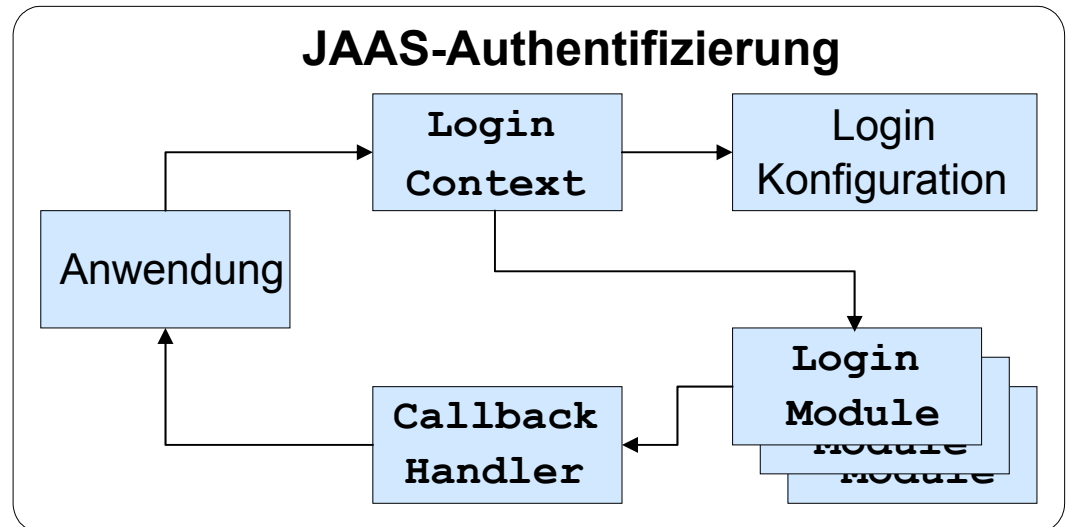
- Nach erfolgreicher Authentifizierung
- Einschränkung der Aktionen eines legitimierten Benutzers durch Zuweisung und Prüfung von Rechten
- Ziel: Aktivitäten verhindern, die Sicherheitsbestimmungen verletzen

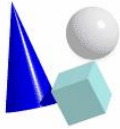




# JAAS-Authentifizierung

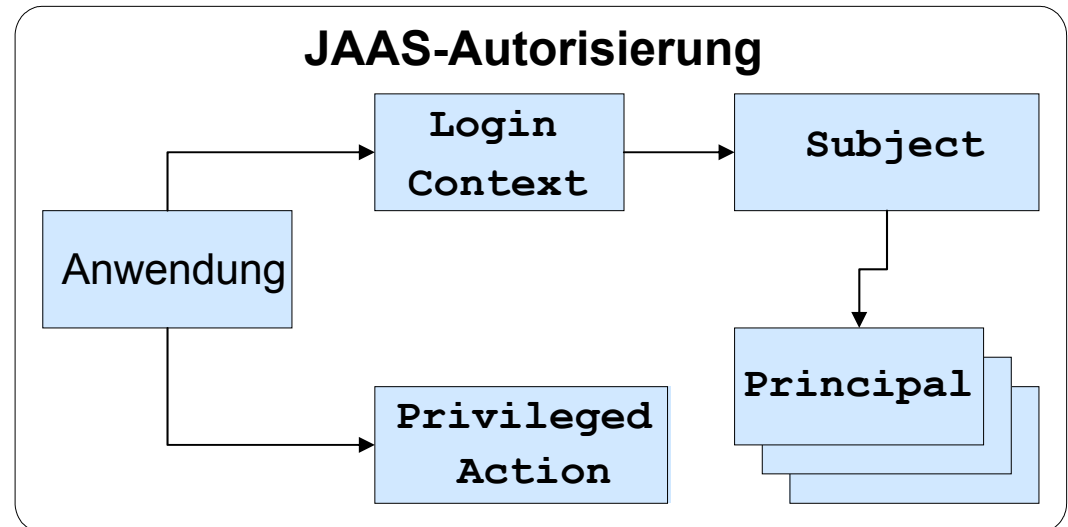
1. Die Anwendung erzeugt ein `LoginContext` und ruft `login()` auf
2. Anhand der Login Konfiguration delegiert der `LoginContext` die Authentifizierung an `LoginModules`
3. `LoginModules` verwenden `CallbackHandler` zur Kommunikation mit der Anwendung, der Umgebung oder dem Anwender
4. Ist die Authentifizierung nicht erfolgreich wird eine `SecurityException` ausgeworfen

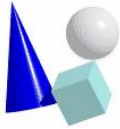




# JAAS-Autorisierung

1. Über den **LoginContext** kann ein **Subject**-Objekt bezogen werden
2. Ein **Subject** kann mit mehreren **Principals** verknüpft sein
3. Die Anwendung kann nun über die Methode **doAs(subject, action)** eine Aktion ausführen.
4. Hierbei kann der **AccessController** prüfen, ob mindestens ein dem **Subject** zugeordneter **Principal** die notwendigen Rechte zur Ausführung dieser Aktion besitzt

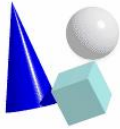




# JAAS-Autorisierung

```
LoginContext lc = new LoginContext ("configName");  
try {  
    lc.login();  
    // authentication successful  
    Subject subject = lc.getSubject();  
    Subject.doAs(subject, someAction);  
    ...  
    lc.logout();  
} catch (LoginException e) {  
    System.out.println("Login failed");  
}
```



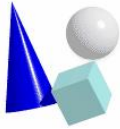


# JAAS-Autorisierung

Erweiterte policy-Datei:

```
grant signedBy "signer_names", codeBase "URL",  
    principal principal_class_name "principal_name",  
    ...  
{  
    permission permission_class_name "target_name",  
        "action", signedBy "signer_names";  
    ...  
};
```

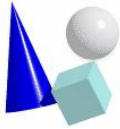




# Agenda

1. Motivation
2. Java 2 Sicherheitsarchitektur
3. JDO Security Model
4. Live Demo
5. Fazit und weiteres Vorgehen





# JDO Security Model

Ziel:

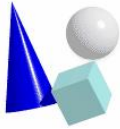
Einschränkung der Zugriffsrechte bei der Nutzung der JDO-API hinsichtlich

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

Nebenbedingungen:

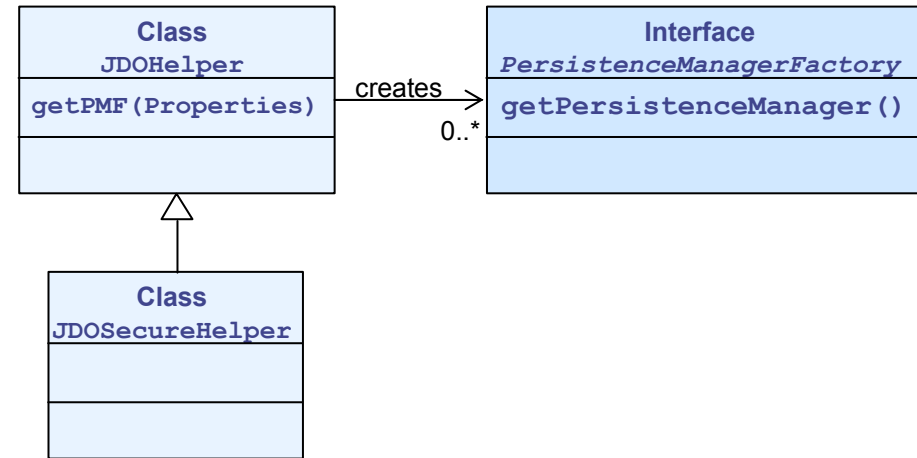
- Kompatibilität zur JDO-Spezifikation
- Unabhängigkeit bezüglich der verwendeten JDO-Implementierung
- Verwaltung der notwendigen Informationen in einer beliebigen JDO-Ressource (z. B. Datenbank oder LDAP-Server)
- Security Model soll keine Möglichkeit zur Umgehung bieten





## Idee

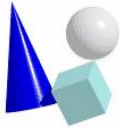
- Eine von `JDOHelper` abgeleitete Klasse (`JDOSecureHelper`) bildet den Einstiegspunkt
- Diese überschreibt die Methode



**`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 des Benutzernamen und Passworts (soll die Umgehung des JDO Security Models verhindern)



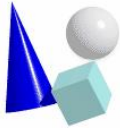


## Idee

- Jeder Methodenaufruf über den **PersistenceManager**, wie z. B. **makePersistent()**, **newQuery()**, **deletePersistent()** sollen nur nach positiver Prüfung von Benutzerrechten erfolgen
- Entsprechende Permissions sollen für den jeweiligen Anwender separat gesetzt werden, z. B.:  

```
permission JDOMakePersistentPermission "java.lang.*";
```



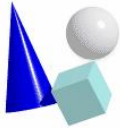


# Realisierung der Authentifizierung

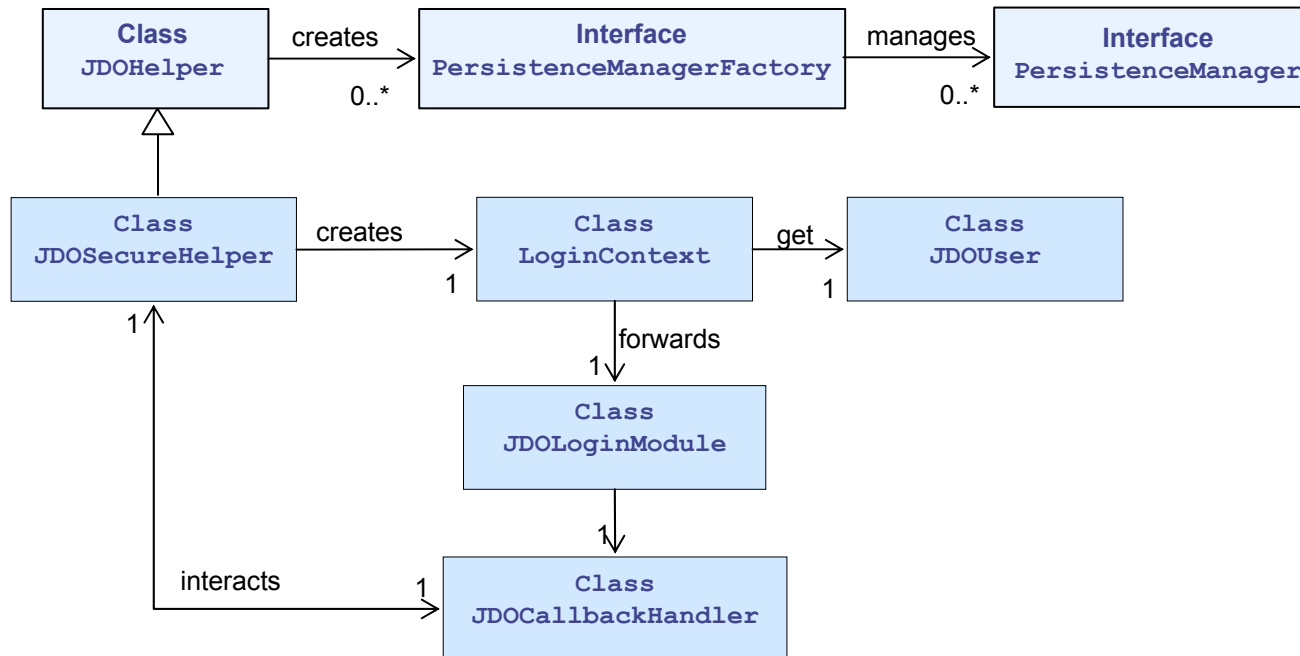
Pakete:

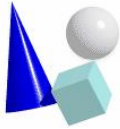
- `de.unimannheim.wifo.jdo.securityextension`
  - `JDOSecureHelper`
  - ...
  
- `de.unimannheim.wifo.jdo.securityextension.authentication`
  - `JDOLoginModule`
  - `JDOUser`
  - `JDOCallbackHandler`





# Realisierung der Authentifizierung





# Realisierung der Autorisierung

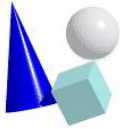
Problem:

- Wie ist der Eingriff in den **PersistenceManager** möglich, ohne sich dabei auf eine konkrete JDO-Implementierung festlegen zu müssen?

Lösung:

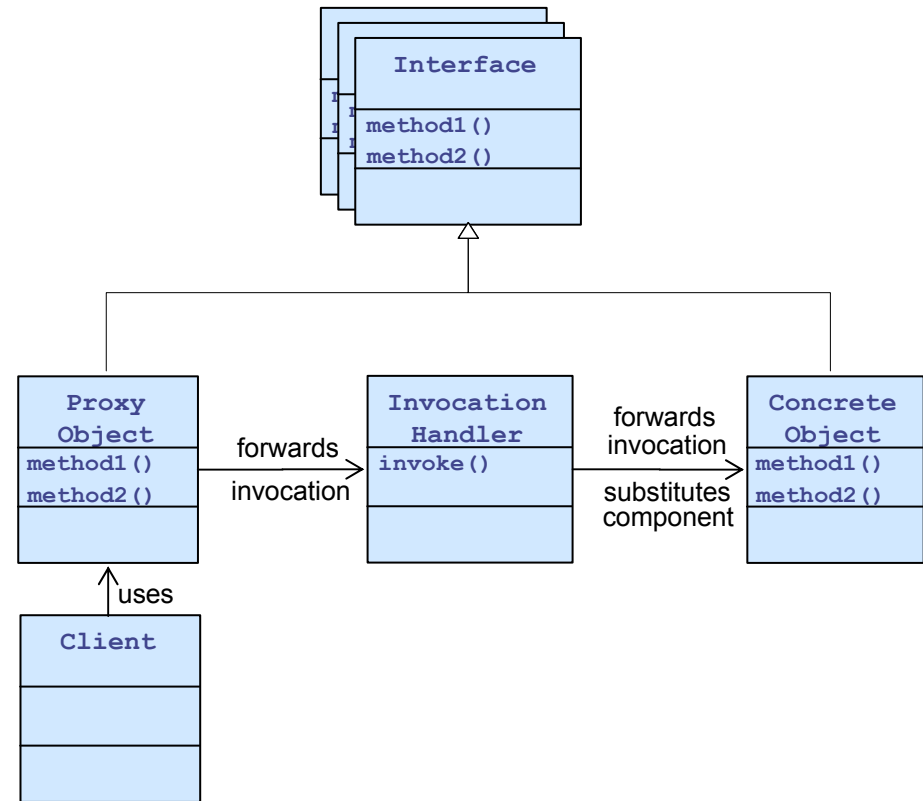
- Durch Verwendung von Dynamic Proxies

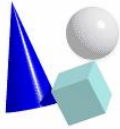




# Dynamic Proxies

- Die Klasse **Proxy** ermöglichen die Erzeugung einer Instanz, die eine Menge von Interfaces zur Laufzeit implementiert
- Aufrufe der Interface-Methoden werden an eine **InvocationHandler**-Instanz weitergeleitet



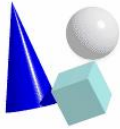


# Realisierung der Authentifizierung

Pakete:

- `de.unimannheim.wifo.jdo.securityextension`
  - `JDOSecureHelper`  
Einstiegspunkt für Anwendungen
  - `PMFProxy` und `PMFInvocationHandler`  
Überschreibt `getPersistenceManagerFactory()` zur Rückgabe eines `PMPProxy` anstelle eines `PersistenceManagers`
  - `PMPProxy` und `PMInvocationHandler`  
Hier erfolgt die Berechtigungsprüfung vor Ausführen einer `PersistenceManager` Methode
  - `QueryProxy` und `QueryInvocationHandler`  
Vor dem Erzeugen eines Query-Objekts erfolgt ebenfalls eine Berechtigungsprüfung



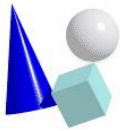


# Realisierung der Authentifizierung

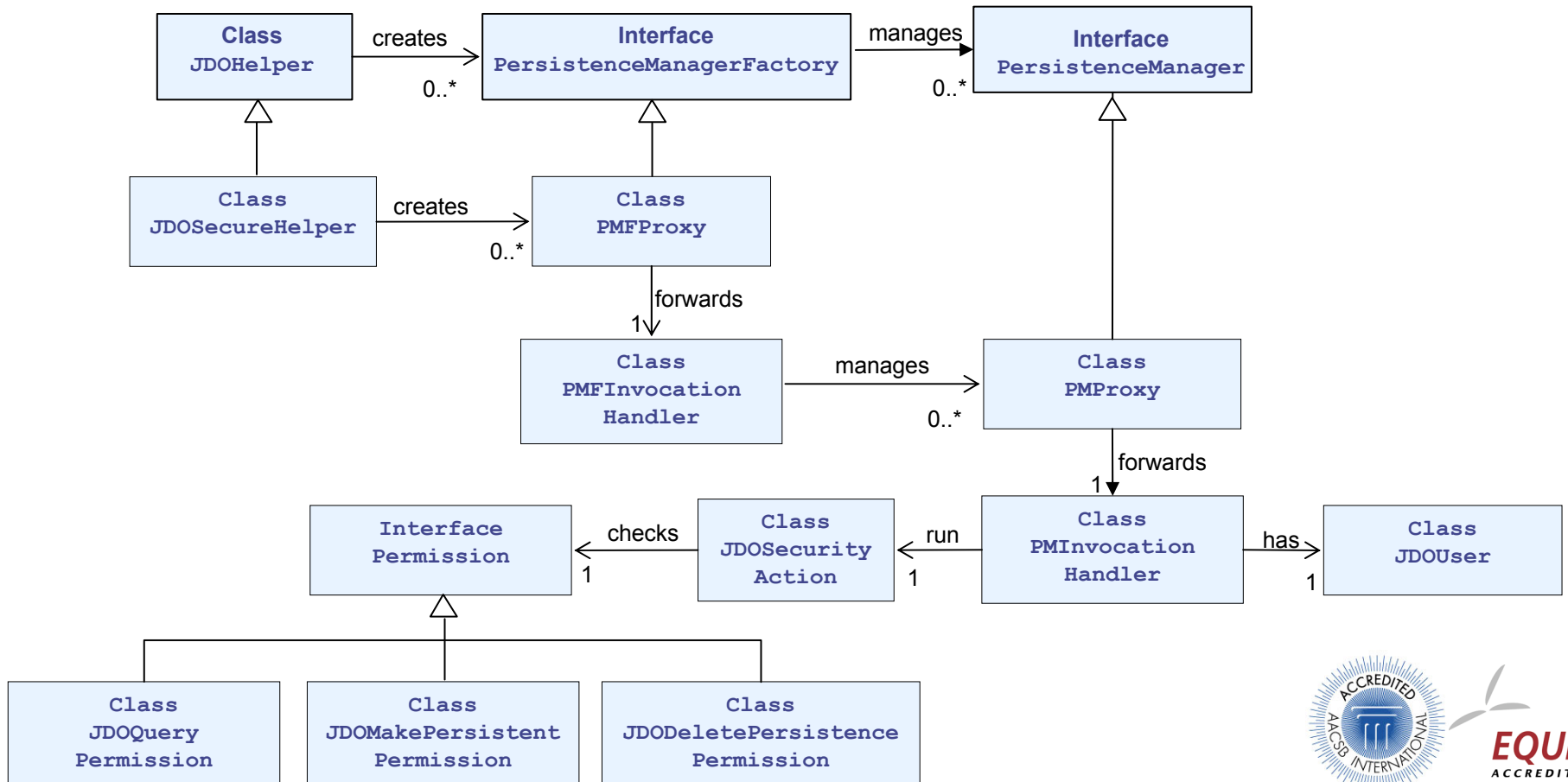
Pakete:

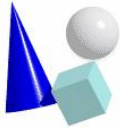
- `de.unimannheim.wifo.jdo.securityextension.permission`
  - `JDOMakePersistentPermission`
  - `JDOQueryPermission`
  - `JDODeletePersistencePermission`
- `de.unimannheim.wifo.jdo.securityextension.authorization`
  - `JDOSecurityAction`  
Zur Prüfung der entsprechenden `Permissions`





# Realisierung der Autorisierung

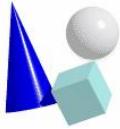




# Agenda

1. Motivation
2. Java 2 Sicherheitsarchitektur
3. JDO Security Model
4. Live Demo
5. Fazit und weiteres Vorgehen

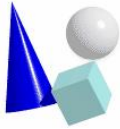




# Agenda

1. Motivation
2. Java 2 Sicherheitsarchitektur
3. JDO Security Model
4. Live Demo
5. Fazit und weiteres Vorgehen



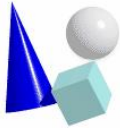


## Fazit

Das vorgestellte JDO Security Model erfüllt die gestellten Anforderungen:

- Einschränkung der Zugriffsrechte bei der Nutzung der JDO-API hinsichtlich
  - Benutzer bzw. ihren Rollen
  - Objekten, 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



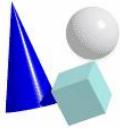


## Noch offen

Ein besonders Problem stellt die Behandlung von Updates dar:

```
product = new Product(...);  
try {  
    tx = pm.currentTransaction();  
    tx.begin();  
    pm.makePersistent(product);  
    ...  
    product.setValue(9.99);  
    tx.commit();  
  
} catch (Exception e) {  
    ...  
}
```

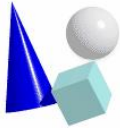




## Weiteres Vorgehen

- Lösung für das Update-Problem finden
- 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 Anwender, der ein Objekt erzeugt hat, darf es verändern und löschen"





# Danke für die Aufmerksamkeit!

Gibt es Fragen?

