Dependency Injection
Facts, Fictions, and Dangers

Markus Knauß
www.iste.uni-stuttgart.de/se

Institut für Softwaretechnologie, Abteilung Software Engineering
Universität Stuttgart

Inhalt

- Einleitung und Motivation
- Abhängigkeiten und Dependency Injection
- Facts
  - JSR 330: Dependency Injection for Java
  - JSR 299: Contexts and Dependency Injection for the Java EE Platform
- Fictions and Dangers

Inhalt

- **Einleitung und Motivation**
- Abhängigkeiten und Dependency Injection
- Facts
  - JSR 330: Dependency Injection for Java
  - JSR 299: Contexts and Dependency Injection for the Java EE Platform
- Fictions and Dangers

Kontext

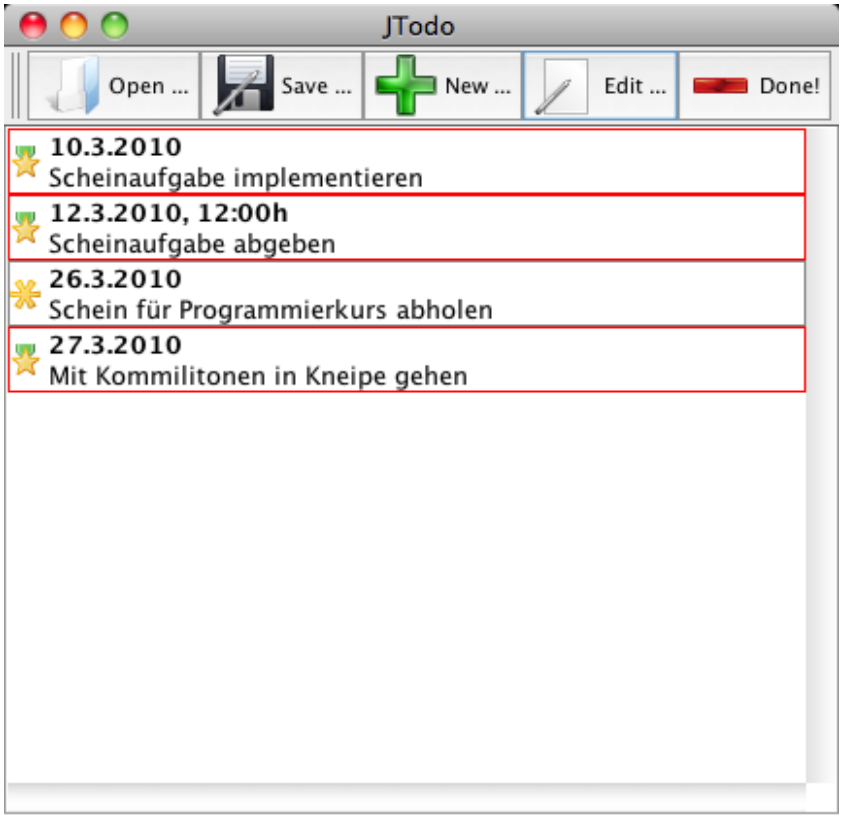
- Im Dezember 2009 wurde die JEE 6 veröffentlicht (JSR 316).
- Der **JSR 330 Dependency Injection for Java** und der **JSR 299 Contexts and Dependency Injection for the Java EE Platform** sind Teil der JEE 6 Plattform.

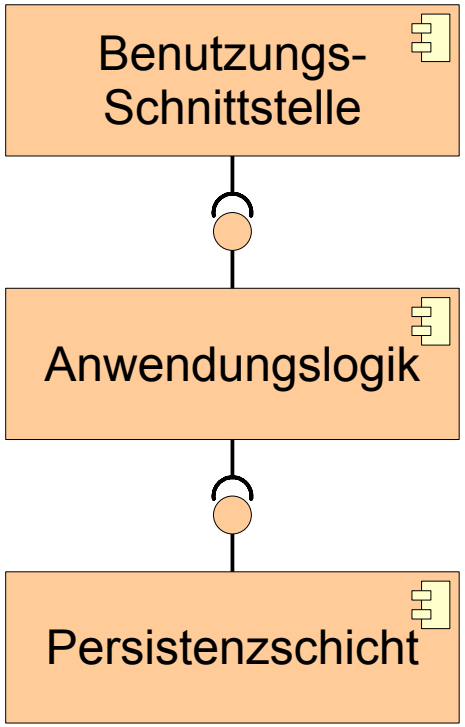
Kontext

- Im Dezember 2009 wurde die JEE 6 veröffentlicht (JSR 316).
- Der **JSR 330 Dependency Injection for Java** und der **JSR 299 Contexts and Dependency Injection for the Java EE Platform** sind Teil der JEE 6 Plattform.

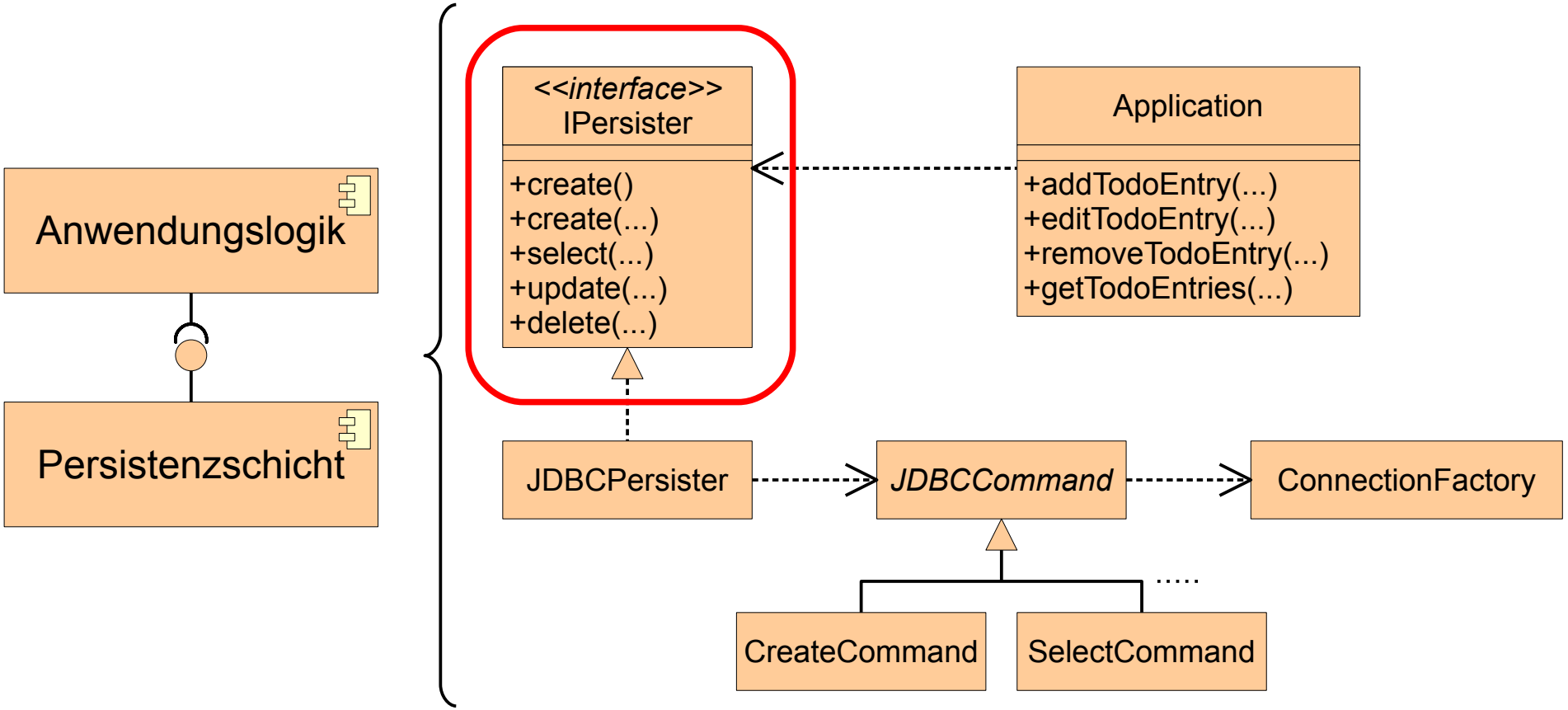
- Was ist Dependency Injection?
- Wie kann Dependency Injection genutzt werden?
- Stimmt das, was versprochen wird?
- Welche Schwierigkeiten sind zu erwarten?

Szenario

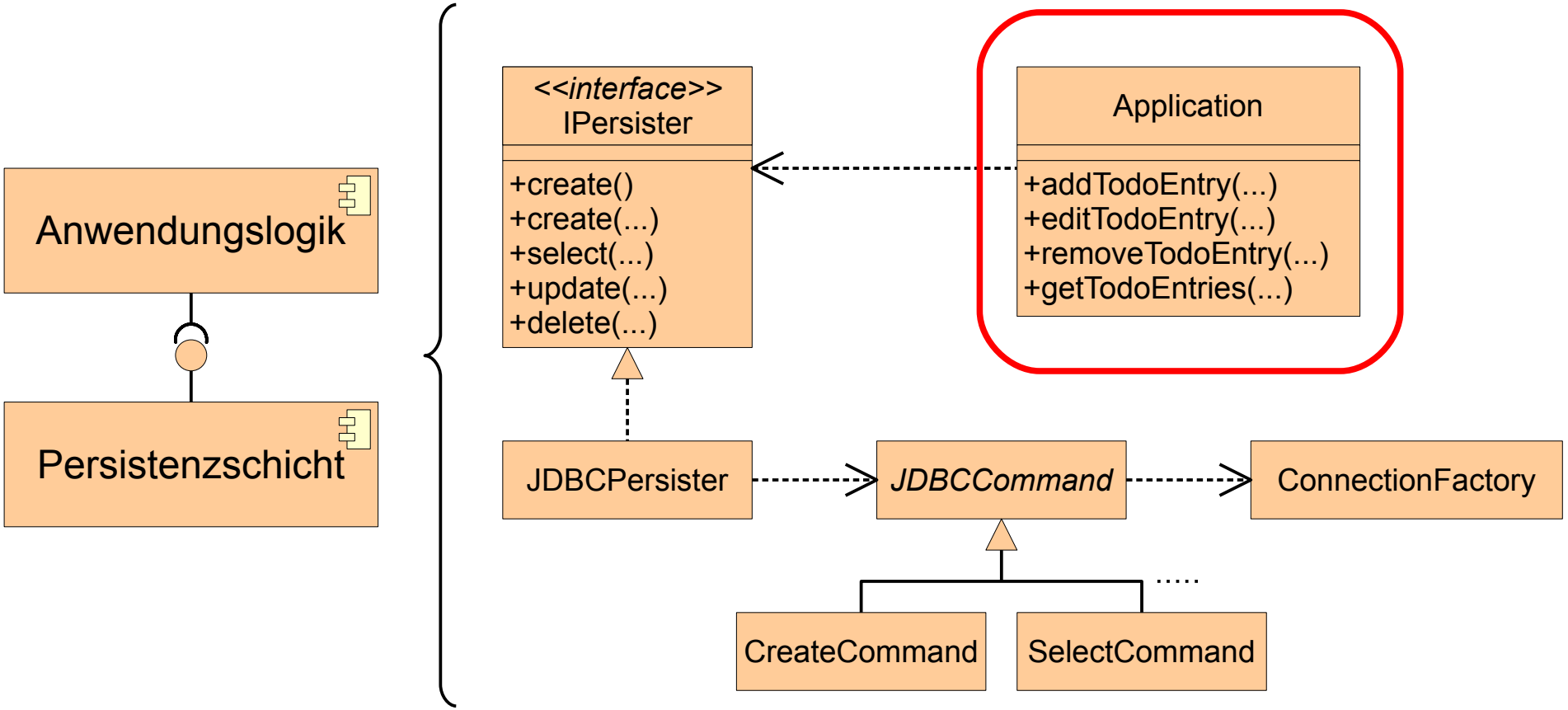

JTodo
Open ...
Save ...
New ...
Edit ...
Done!
10.3.2010
Scheinaufgabe implementieren
12.3.2010, 12:00h
Scheinaufgabe abgeben
26.3.2010
Schein für Programmierkurs abholen
27.3.2010
Mit Kommilitonen in Kneipe gehen


Benutzungs-
Schnittstelle
Anwendungslogik
Persistenzschicht

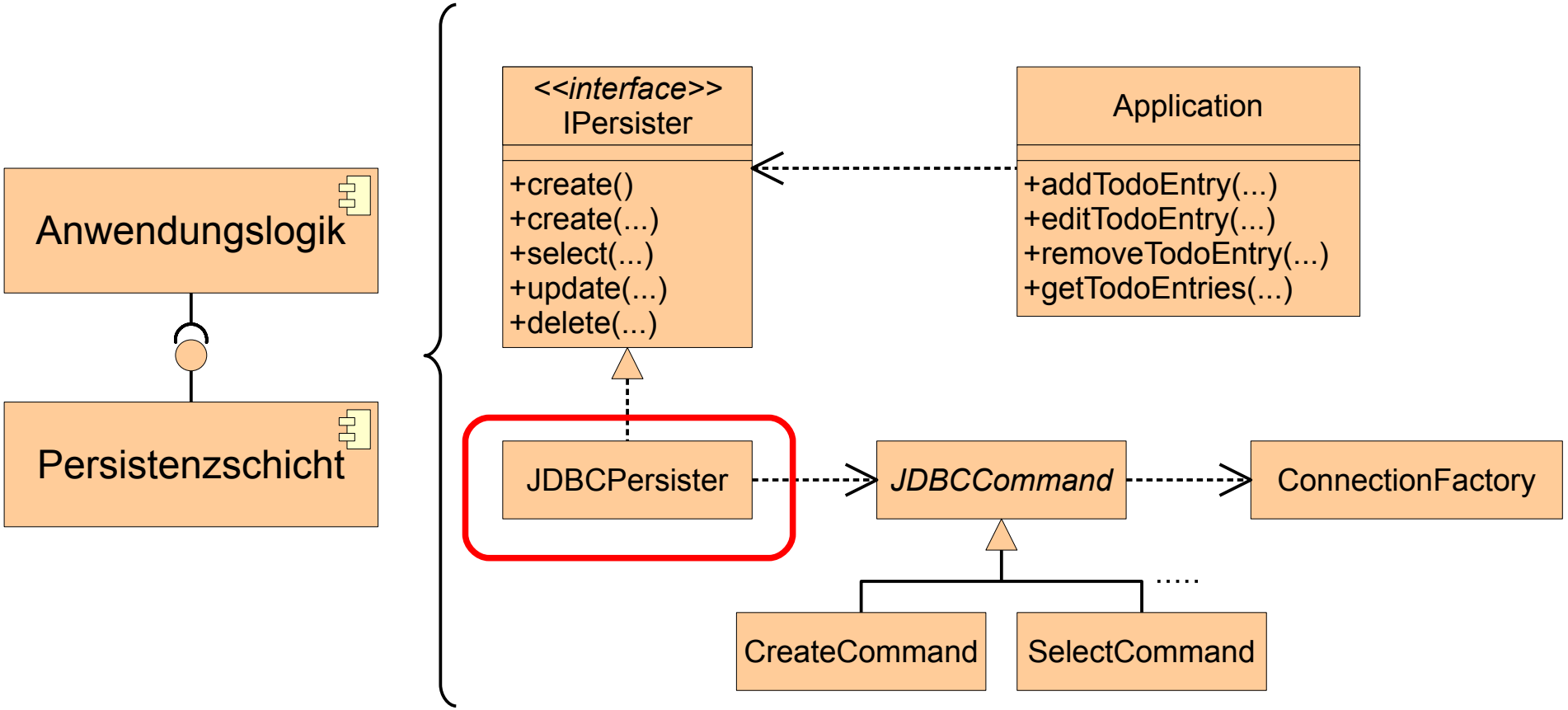
Szenario


Anwendungslogik
Persistenzschicht
<<interface>>
IPersister
+create()
+create(...)
+select(...)
+update(...)
+delete(...)
Application
+addTodoEntry(...)
+editTodoEntry(...)
+removeTodoEntry(...)
+getTodoEntries(...)
JDBCPersister
JDBCCommand
ConnectionFactory
.....
CreateCommand
SelectCommand

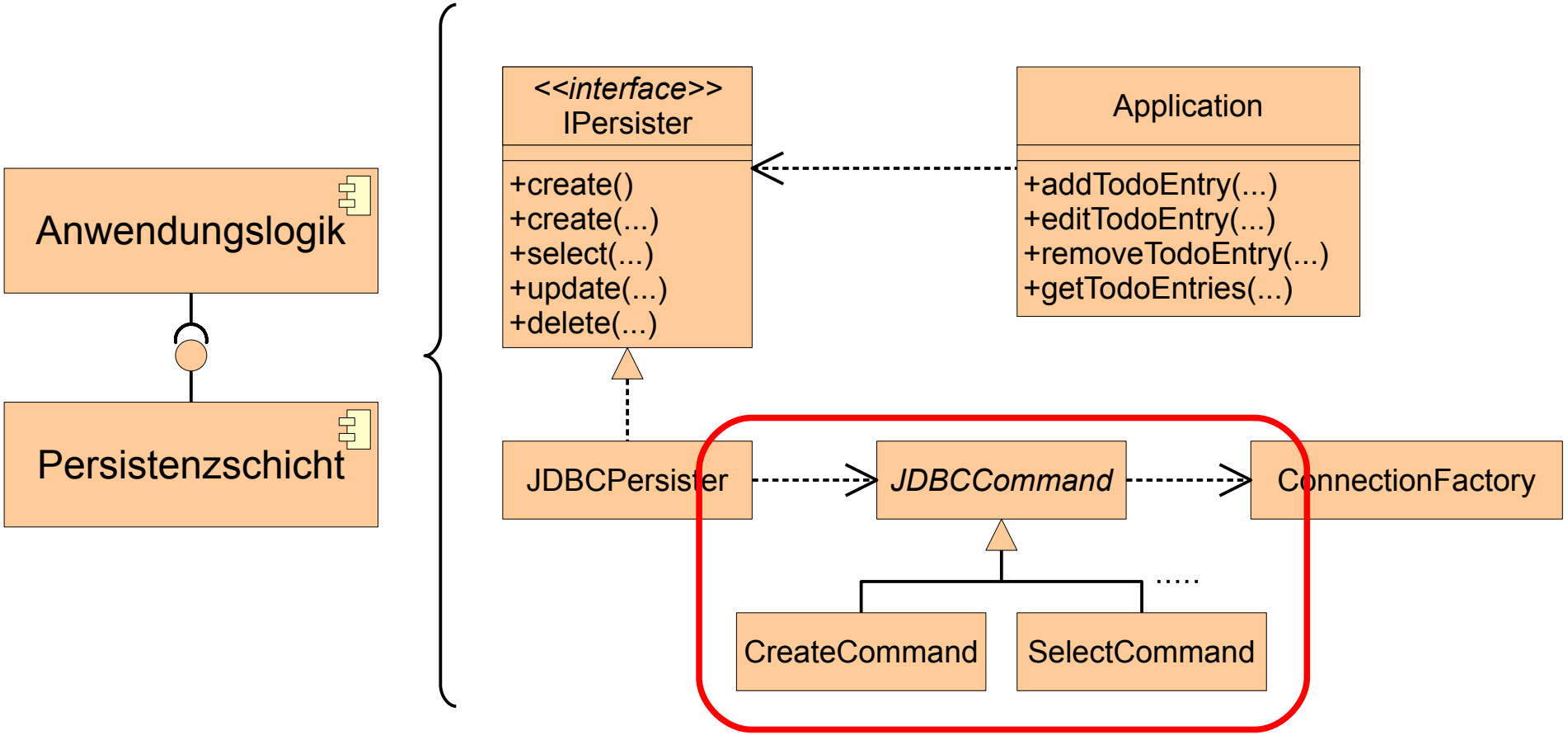
Szenario


Anwendungslogik
Persistenzschicht
<<interface>>
IPersister
+create()
+create(...)
+select(...)
+update(...)
+delete(...)
Application
+addTodoEntry(...)
+editTodoEntry(...)
+removeTodoEntry(...)
+getTodoEntries(...)
JDBCPersister
JDBCCommand
ConnectionFactory
.....
CreateCommand
SelectCommand

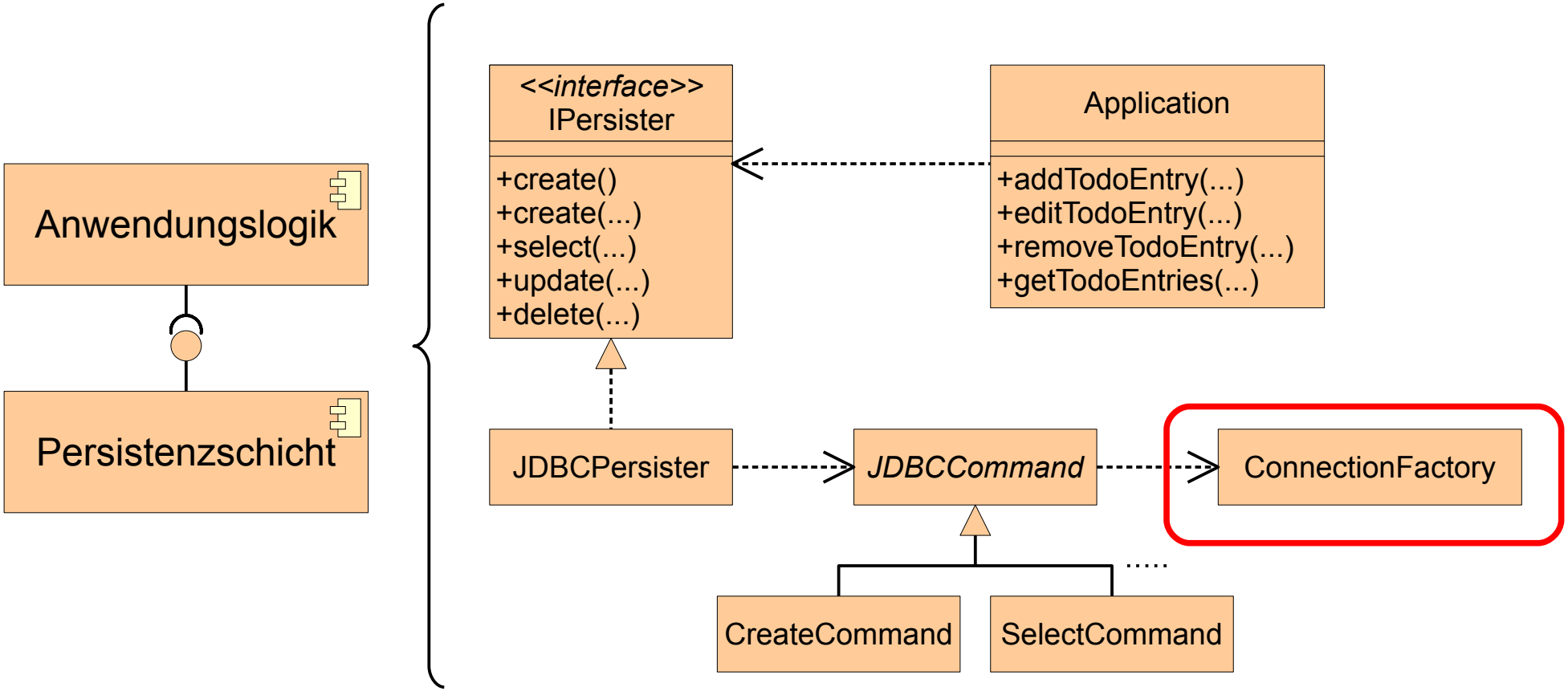
Szenario


Anwendungslogik
Persistenzschicht
<<interface>>
IPersister
+create()
+create(...)
+select(...)
+update(...)
+delete(...)
Application
+addTodoEntry(...)
+editTodoEntry(...)
+removeTodoEntry(...)
+getTodoEntries(...)
JDBCPersister
JDBCCommand
ConnectionFactory
.....
CreateCommand
SelectCommand

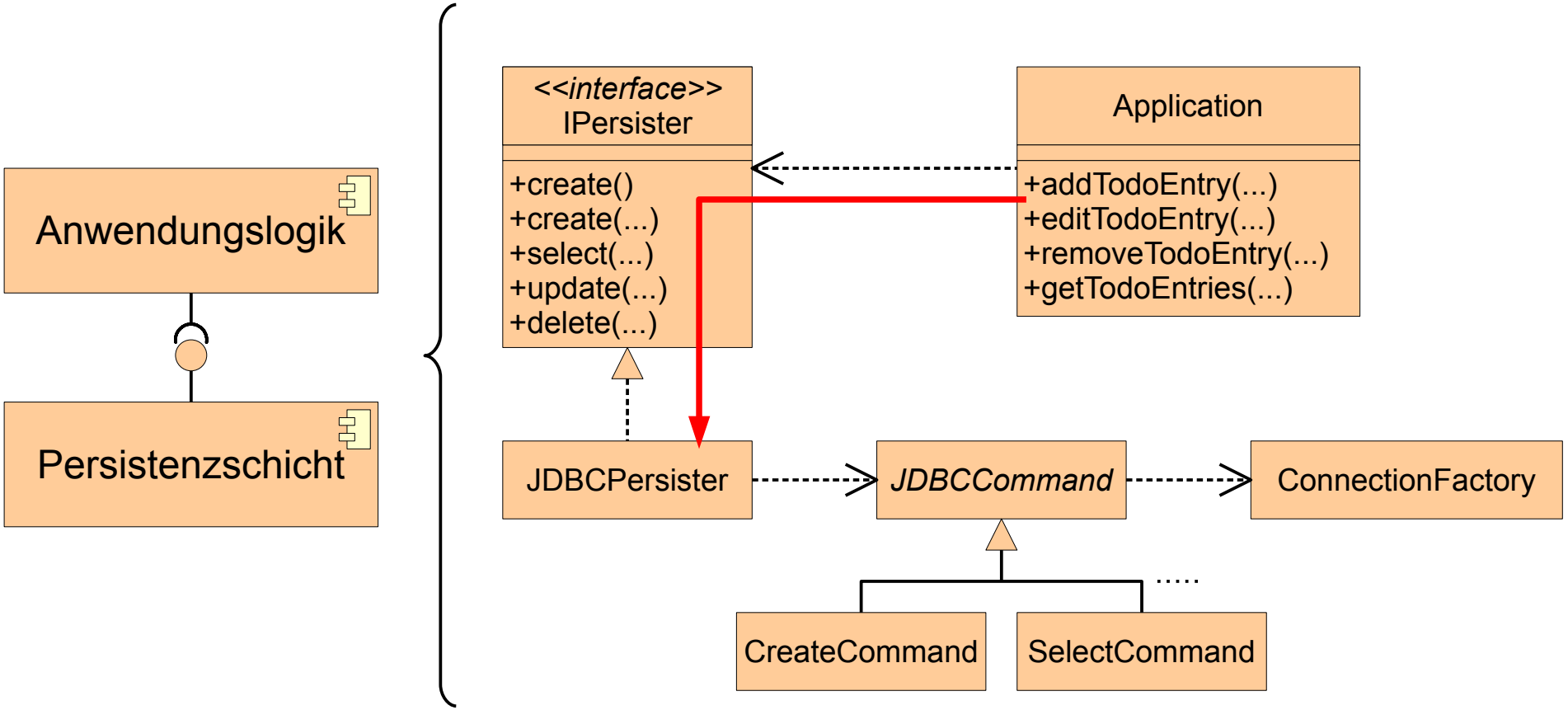
Szenario


Anwendungslogik
Persistenzschicht
<<interface>>
IPersister
+create()
+create(...)
+select(...)
+update(...)
+delete(...)
Application
+addTodoEntry(...)
+editTodoEntry(...)
+removeTodoEntry(...)
+getTodoEntries(...)
JDBCPersister
JDBCCommand
ConnectionFactory
.....
CreateCommand
SelectCommand

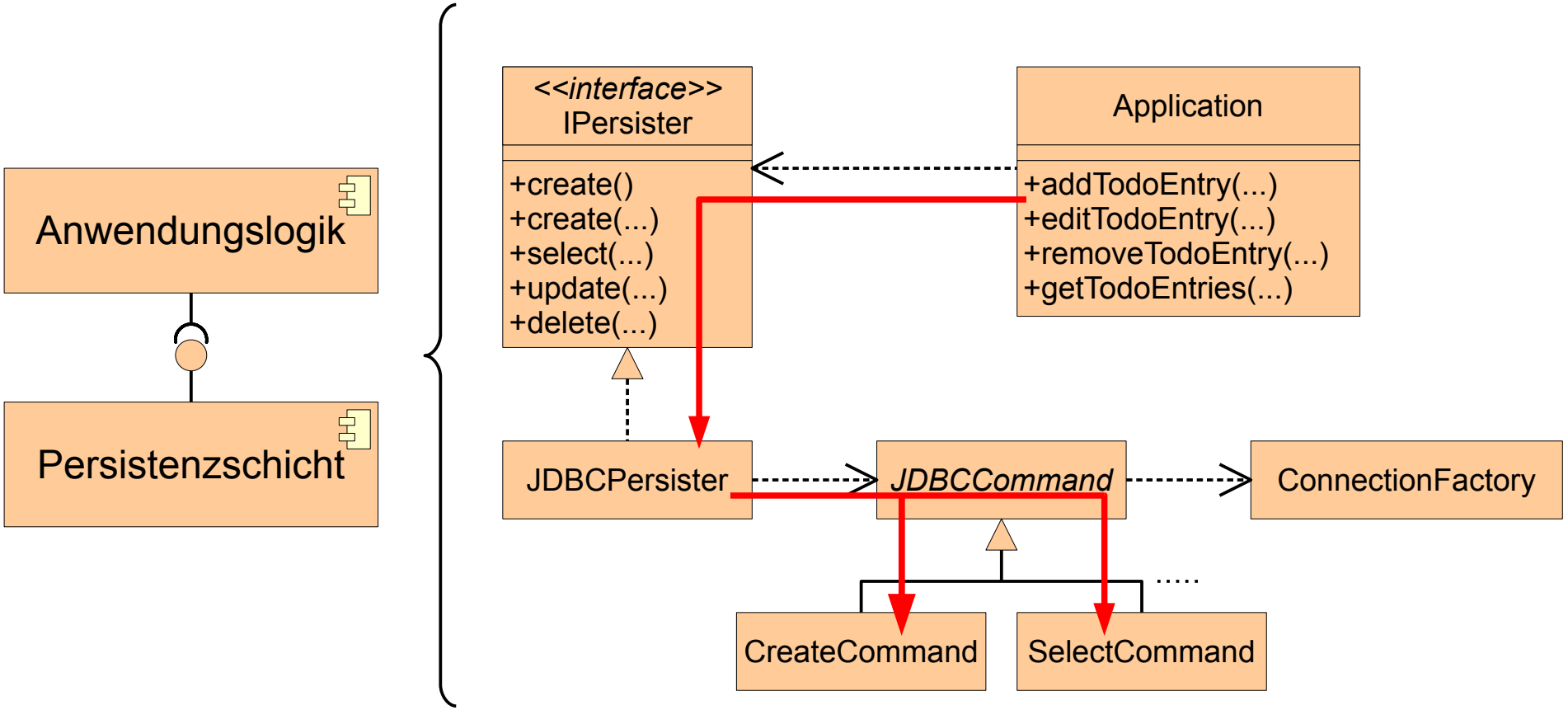
Szenario


Anwendungslogik
Persistenzschicht
<<interface>>
IPersister
+create()
+create(...)
+select(...)
+update(...)
+delete(...)
Application
+addTodoEntry(...)
+editTodoEntry(...)
+removeTodoEntry(...)
+getTodoEntries(...)
JDBCPersister
JDBCCommand
ConnectionFactory
.....
CreateCommand
SelectCommand

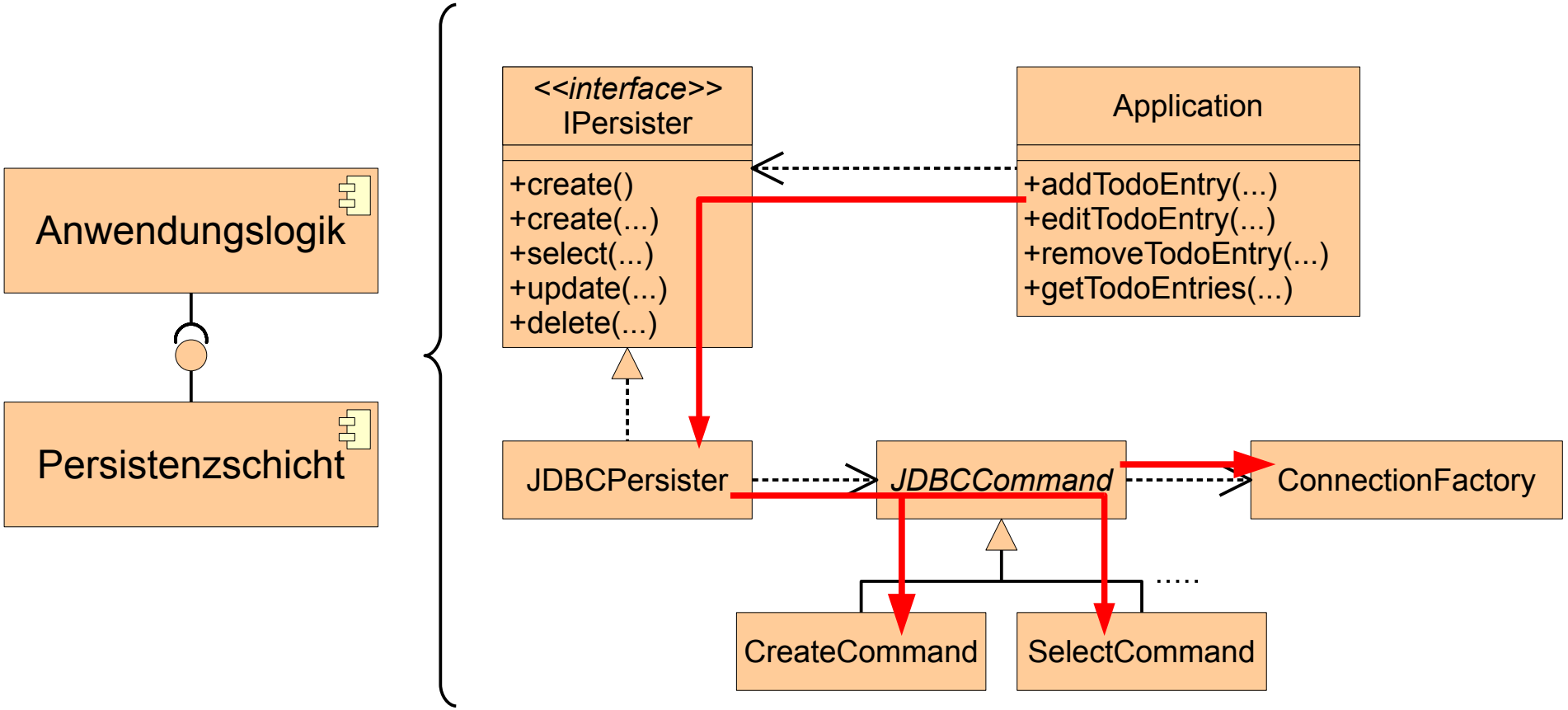
Szenario


Anwendungslogik
Persistenzschicht
<<interface>>
IPersister
+create()
+create(...)
+select(...)
+update(...)
+delete(...)
Application
+addTodoEntry(...)
+editTodoEntry(...)
+removeTodoEntry(...)
+getTodoEntries(...)
JDBCPersister
JDBCCommand
ConnectionFactory
.....
CreateCommand
SelectCommand

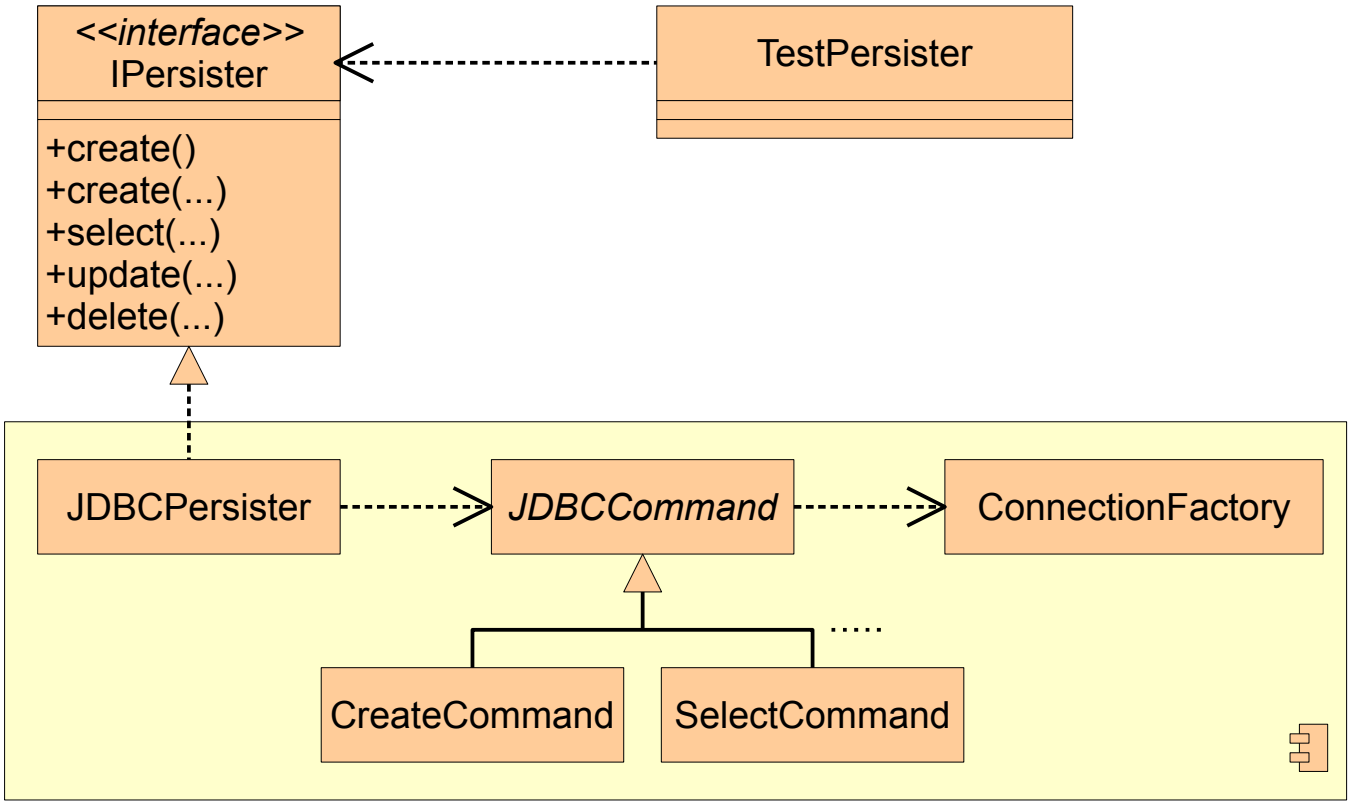
Szenario


Anwendungslogik
Persistenzschicht
<<interface>>
IPersister
+create()
+create(...)
+select(...)
+update(...)
+delete(...)
Application
+addTodoEntry(...)
+editTodoEntry(...)
+removeTodoEntry(...)
+getTodoEntries(...)
JDBCPersister
JDBCCommand
ConnectionFactory
.....
CreateCommand
SelectCommand

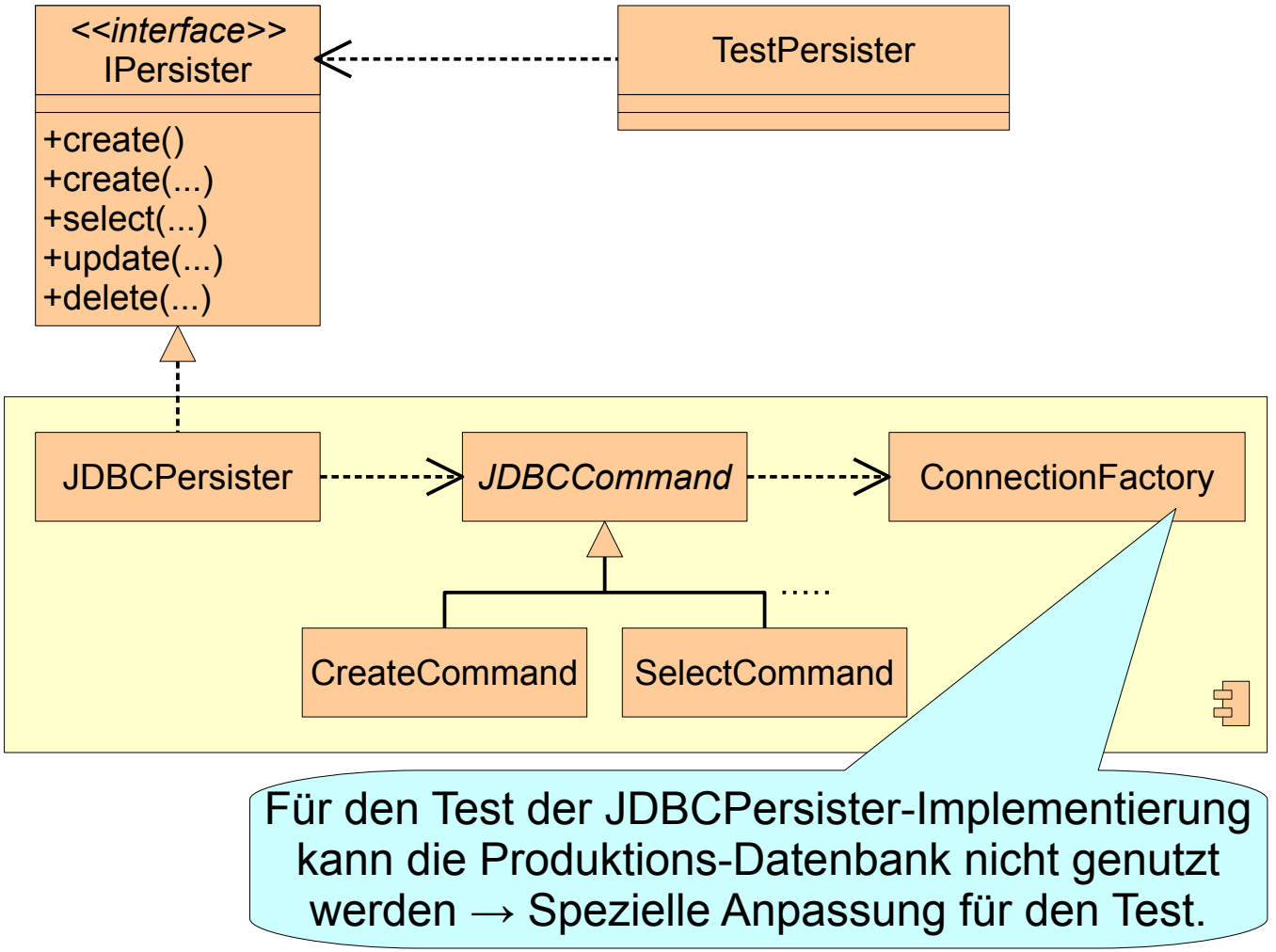
Szenario


Anwendungslogik
Persistenzschicht
<<interface>>
IPersister
+create()
+create(...)
+select(...)
+update(...)
+delete(...)
Application
+addTodoEntry(...)
+editTodoEntry(...)
+removeTodoEntry(...)
+getTodoEntries(...)
JDBCPersister
JDBCCommand
ConnectionFactory
.....
CreateCommand
SelectCommand

Implementierung und Test der Persistenzschicht

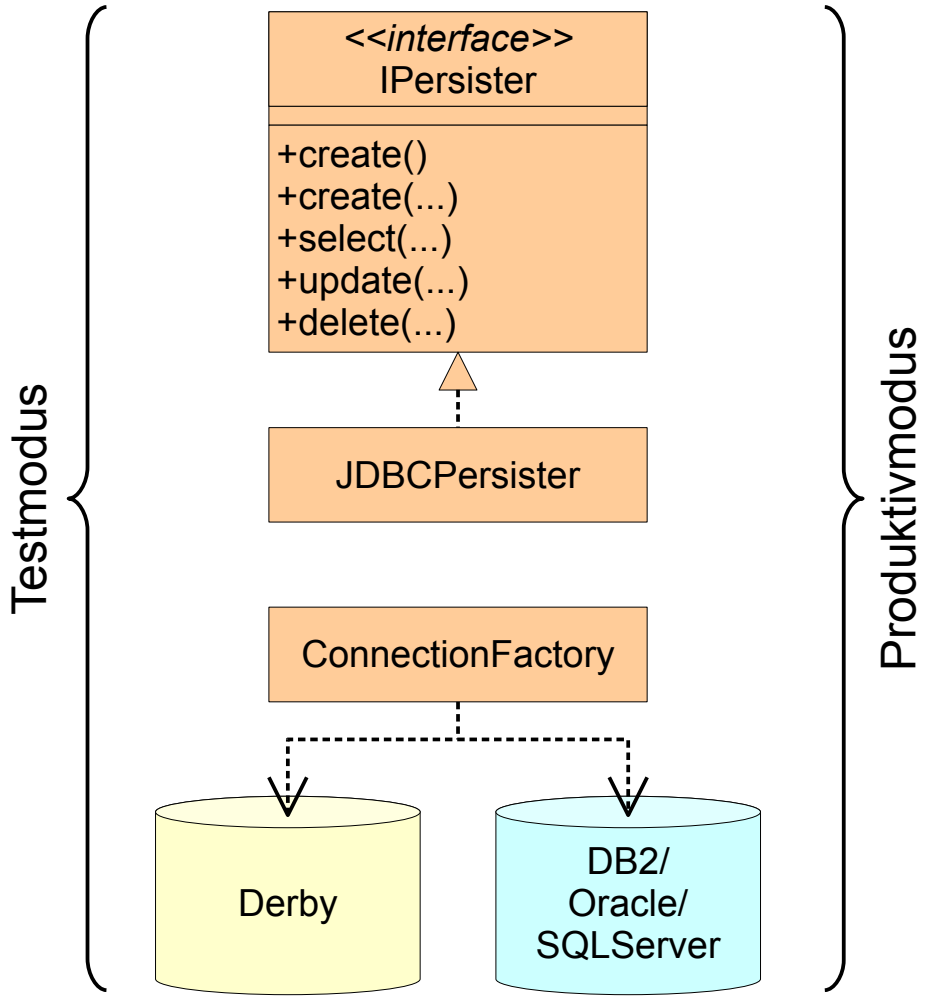


Implementierung und Test der Persistenzschicht



Motivation

- **Dependency Injection** bietet eine Lösung für die **einfache, variable Konfiguration** und die **automatische Auflösung der Abhängigkeiten** in einem Programm.



Inhalt

- Einleitung und Motivation
- **Abhängigkeiten und Dependency Injection**
- Facts
  - JSR 330: Dependency Injection for Java
  - JSR 299: Contexts and Dependency Injection for the Java EE Platform
- Fictions and Dangers

Abhängigkeiten traditionell konfiguriert

- **Direkt konfiguriert:** Das Attribut, in dem die Verbindung zur Datenbank gespeichert wird, wird direkt mit der Verbindung initialisiert.

Abhängigkeiten traditionell konfiguriert

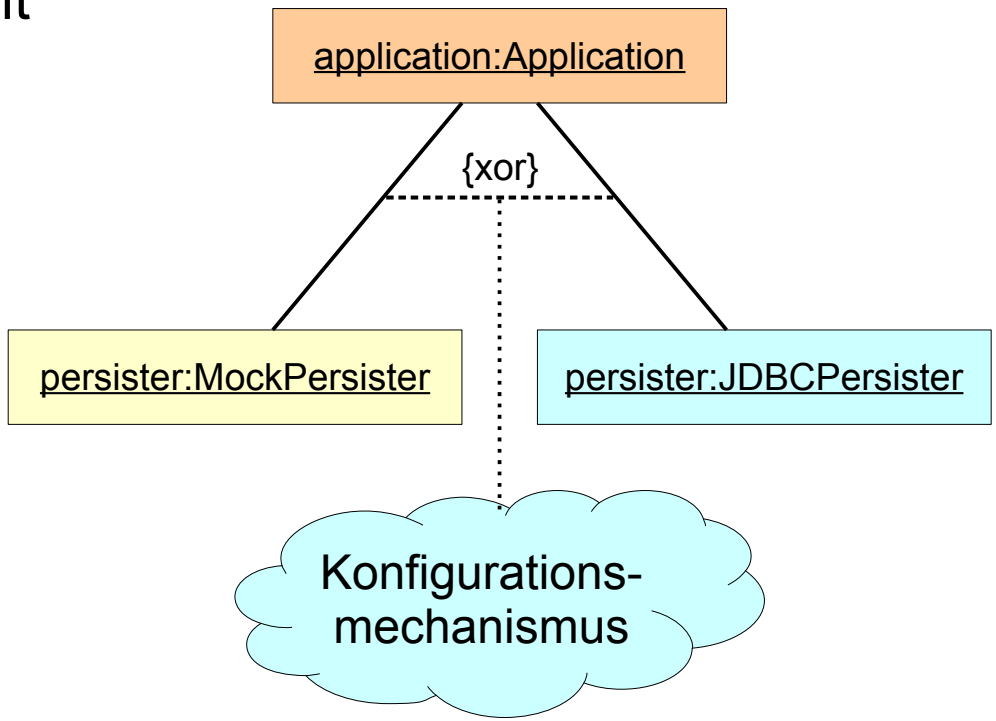
- **Direkt konfiguriert:** Das Attribut, in dem die Verbindung zur Datenbank gespeichert wird, wird direkt mit der Verbindung initialisiert.
- **Indirekt konfiguriert:** Die Verbindung zur Datenbank wird über eine Factory oder einen Service Locator initialisiert.

Abhängigkeiten traditionell konfiguriert

- **Direkt konfiguriert:** Das Attribut, in dem die Verbindung zur Datenbank gespeichert wird, wird direkt mit der Verbindung initialisiert.
- **Indirekt konfiguriert:** Die Verbindung zur Datenbank wird über eine Factory oder einen Service Locator initialisiert.
- **Extern konfiguriert:** Die Verbindung zur Datenbank wird bei der Instantiierung jedes `JDBCCommands` mitgegeben.

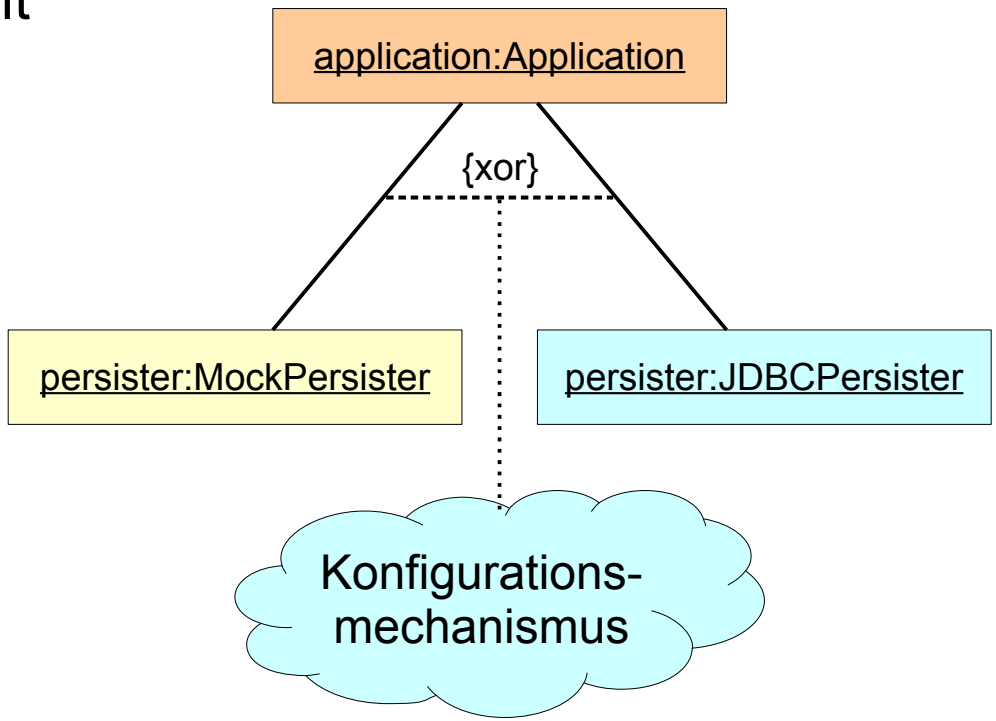
Dependency Injection

- Abhängigkeiten werden implizit oder zentral definiert.



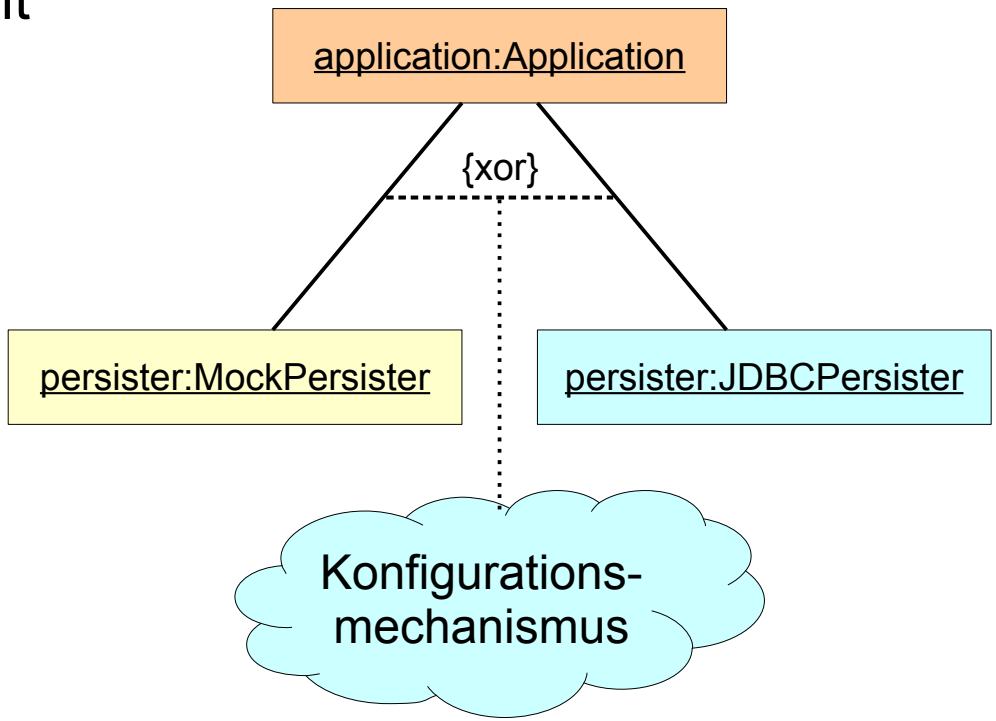
Dependency Injection

- Abhängigkeiten werden implizit oder zentral definiert.
- Abhängigkeiten werden von einem zentralen, für die Beteiligten nicht sichtbaren Mechanismus aufgelöst.



Dependency Injection

- Abhängigkeiten werden implizit oder zentral definiert.
- Abhängigkeiten werden von einem zentralen, für die Beteiligten nicht sichtbaren Mechanismus aufgelöst.
- Instanziierung und Nutzung eines Objekts sind getrennt.



Inhalt

- Einleitung und Motivation
- Abhängigkeiten und Dependency Injection
- **Facts**
  - **JSR 330: Dependency Injection for Java**
  - JSR 299: Contexts and Dependency Injection for the Java EE Platform
- Fictions and Dangers

JSR 330: Dependency Injection for Java

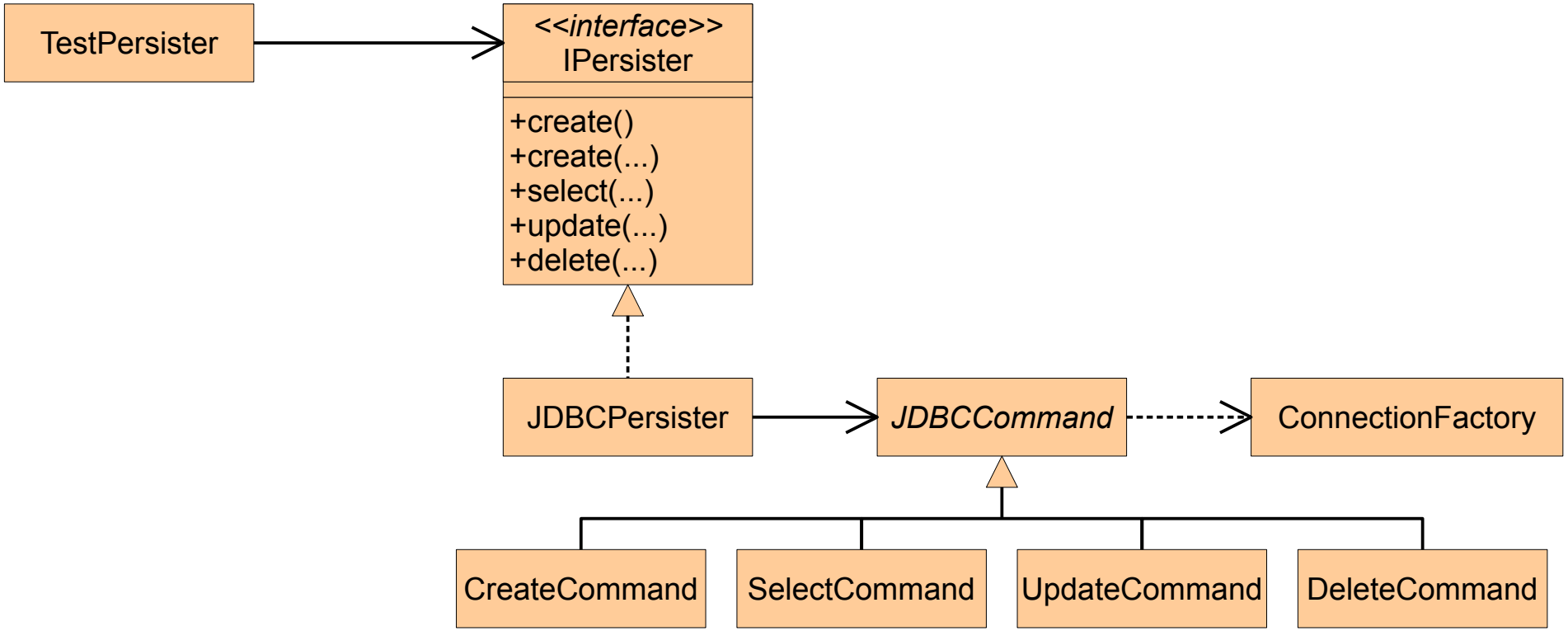
- Definiert fünf Annotation:
  - `@Inject`: Markiert aufzulösende Abhängigkeiten
  - `@Named`, `@Qualifier`: Einschränkung der injizierten Objekte
  - `@Singleton`, `@Scope`: Definition der Lebenszeit injizierter Instanzen
- Zusätzlich definiert der JSR das Interface `Provider<T>`
- Der **Konfigurationsmechanismus** wird im JSR 330 **nicht spezifiziert**.

JSR 330: Dependency Injection for Java

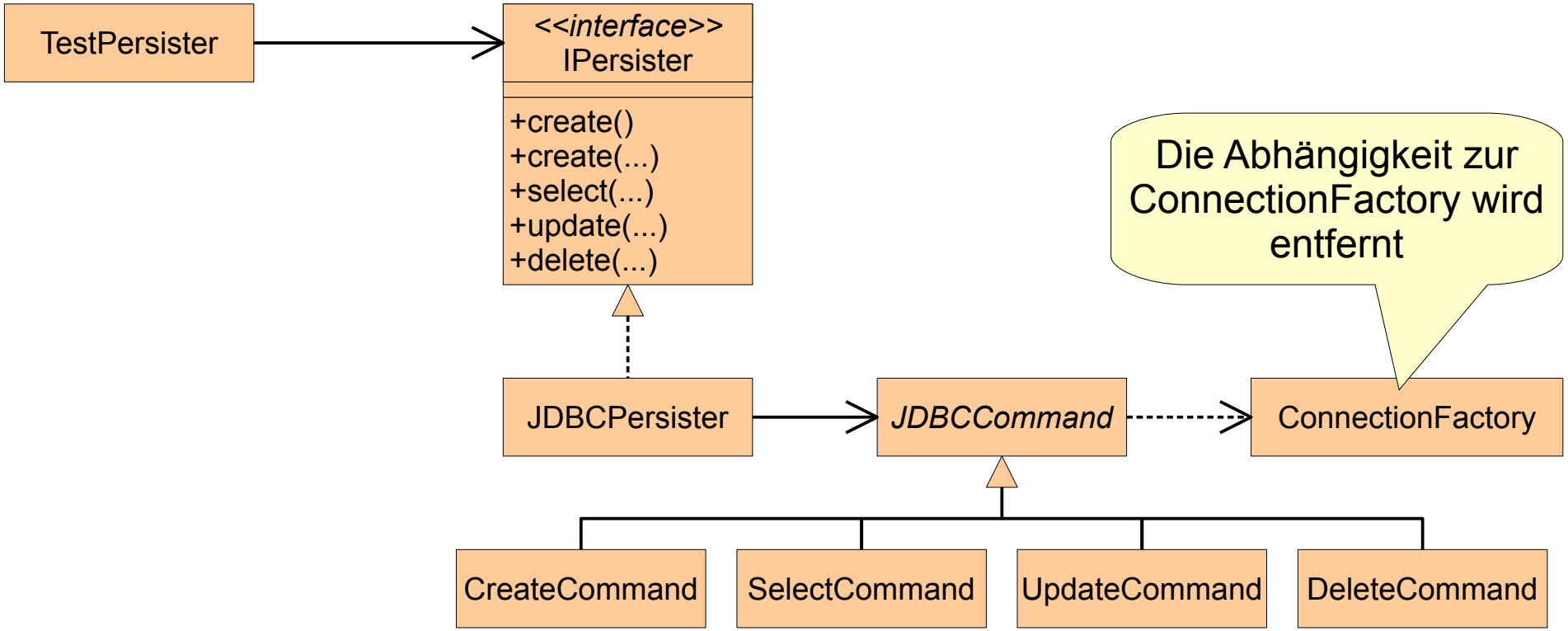
- Definiert fünf Annotation:
  - `@Inject`: Markiert aufzulösende Abhängigkeiten
  - `@Named`, `@Qualifier`: Einschränkung der injizierten Objekte
  - `@Singleton`, `@Scope`: Definition der Lebenszeit injizierter Instanzen
- Zusätzlich definiert der JSR das Interface `Provider<T>`
- Der **Konfigurationsmechanismus** wird im JSR 330 **nicht spezifiziert**.

- Referenzimplementierung: Google Guice
  - **Link:** http://code.google.com/p/google-guice/

Dependency Injection am Beispiel



Dependency Injection am Beispiel



Instanziierungen aus JDBCCommand entfernen

```
public abstract class JDBCCommand {
   //...


   private Connection con;


   public final void execute() {
      //...
      try {
         con = ConnectionFactory.
               getInstance().
               getConnection();
         //...
```

Instanziierungen aus JDBCCommand entfernen

```
public abstract class JDBCCommand {
   //...

   private Connection con;

   public final void execute() {
      //...
      try {
         con = ConnectionFactory.
               getInstance().
               getConnection();
         //...
```

```
public abstract class JDBCCommand {
   //...

   @Inject
   private Provider<Connection>
         connectionProvider;

   public final void execute() {
      //...
      try {
         con = connectionProvider.get();
         //...
```

Provider für java.sql.Connection erstellen

```
public class JDBCConnectionProvider implements Provider<Connection> {

   @Inject
   private Logger log;

   private final String dbUrl;

   @Inject
   public JDBCConnectionProvider(
         @DatabaseDriver String dbDriver, @DatabaseURL String dbUrl) {
      this.dbUrl = dbUrl;
      //... Load the driver for the DriverManager
   }

   @Override
   public Connection get() {
      Connection con = null;
      try {
         con = DriverManager.getConnection(dbUrl);
      } catch (SQLException e) {
         throw new ProvisionException(e.getMessage(), e);
      }
      return con;
   }
}
```

Provider für java.sql.Connection erstellen

```
public class JDBCConnectionProvider implements Provider<Connection> {

   @Inject
   private Logger log;

   private final String dbUrl;

   @Inject
   public JDBCConnectionProvider(
         @DatabaseDriver String dbDriver, @DatabaseURL String dbUrl) {
      this.dbUrl = dbUrl;
      //... Load the driver for the DriverManager
   }

   @Override
   public Connection get() {
      Connection con = null;
      try {
         con = DriverManager.getConnection(dbUrl);
      } catch (SQLException e) {
         throw new ProvisionException(e.getMessage(), e);
      }
      return con;
   }
}
```

Provider für java.sql.Connection erstellen

```
public class JDBCConnectionProvider implements Provider<Connection> {

   @Inject
   private Logger log;

   private final String dbUrl;

   @Inject
   public JDBCConnectionProvider(
         @DatabaseDriver String dbDriver, @DatabaseURL String dbUrl) {
      this.dbUrl = dbUrl;
      //... Load the driver for the DriverManager
   }

   @Override
   public Connection get() {
      Connection con = null;
      try {
         con = DriverManager.getConnection(dbUrl);
      } catch (SQLException e) {
         throw new ProvisionException(e.getMessage(), e);
      }
      return con;
   }
}
```

Provider für java.sql.Connection erstellen

```
public class JDBCConnectionProvider implements Provider<Connection> {

   @Inject
   private Logger log;

   private final String dbUrl;

   @Inject
   public JDBCConnectionProvider(
         @DatabaseDriver String dbDriver, @DatabaseURL String dbUrl) {
      this.dbUrl = dbUrl;
      //... Load the driver for the DriverManager
   }

   @Override
   public Connection get() {
      Connection con = null;
      try {
         con = DriverManager.getConnection(dbUrl);
      } catch (SQLException e) {
         throw new ProvisionException(e.getMessage(), e);
      }
      return con;
   }
}
```

Abhängigkeiten konfigurieren

```
public class TestPersisterModule extends AbstractModule {

   @Override
   protected void configure() {
      bind(Connection.class).toProvider(JDBCConnectionProvider.class);

      bind(IPersister.class).to(JDBCPersister.class);

      bind(String.class).annotatedWith(DatabaseDriver.class).
            toInstance("org.apache.derby.jdbc.EmbeddedDriver");
      bind(String.class).annotatedWith(DatabaseURL.class).
            toInstance("jdbc:derby:tododb;create=true");
   }
}
```

Abhängigkeiten konfigurieren

```
public class TestPersisterModule extends AbstractModule {

   @Override
   protected void configure() {
      bind(Connection.class).toProvider(JDBCConnectionProvider.class);

      bind(IPersister.class).to(JDBCPersister.class);

      bind(String.class).annotatedWith(DatabaseDriver.class).
            toInstance("org.apache.derby.jdbc.EmbeddedDriver");
      bind(String.class).annotatedWith(DatabaseURL.class).
            toInstance("jdbc:derby:tododb;create=true");
   }
}
```

Abhängigkeiten konfigurieren

```
public class TestPersisterModule extends AbstractModule {

   @Override
   protected void configure() {
      bind(Connection.class).toProvider(JDBCConnectionProvider.class);

      bind(IPersister.class).to(JDBCPersister.class);

      bind(String.class).annotatedWith(DatabaseDriver.class).
            toInstance("org.apache.derby.jdbc.EmbeddedDriver");
      bind(String.class).annotatedWith(DatabaseURL.class).
            toInstance("jdbc:derby:tododb;create=true");
   }
}
```

Abhängigkeiten konfigurieren

```
public class TestPersisterModule extends AbstractModule {

   @Override
   protected void configure() {
      bind(Connection.class).toProvider(JDBCConnectionProvider.class);

      bind(IPersister.class).to(JDBCPersister.class);

      bind(String.class).annotatedWith(DatabaseDriver.class).
            toInstance("org.apache.derby.jdbc.EmbeddedDriver");
      bind(String.class).annotatedWith(DatabaseURL.class).
            toInstance("jdbc:derby:tododb;create=true");
   }
}
```

Testtreiber anpassen

```
public class TestPersister {

   private static final Injector injector =
         Guice.createInjector(new TestPersisterModule());

   private final IPersister persister;

   public TestPersister() {
      persister = injector.getInstance(IPersister.class);
   }

   @BeforeClass
   public static void setupTable() {
      JDBCCommand createTable = new JDBCCommand() {
         @Override
         public String getStatement() {
            return "CREATE TABLE todos (id BIGINT NOT NULL, " +
                  "description VARCHAR(1024), duedate DATE, " +
                  "PRIMARY KEY (id))";
         }
      };
      injector.injectMembers(createTable);
      createTable.execute();
   }
   //...
```

Testtreiber anpassen

```
public class TestPersister {

   private static final Injector injector =
         Guice.createInjector(new TestPersisterModule());

   private final IPersister persister;

   public TestPersister() {
      persister = injector.getInstance(IPersister.class);
   }

   @BeforeClass
   public static void setupTable() {
      JDBCCommand createTable = new JDBCCommand() {
         @Override
         public String getStatement() {
            return "CREATE TABLE todos (id BIGINT NOT NULL, " +
                  "description VARCHAR(1024), duedate DATE, " +
                  "PRIMARY KEY (id))";
         }
      };
      injector.injectMembers(createTable);
      createTable.execute();
   }
   //...
```

Testtreiber anpassen

```
public class TestPersister {

   private static final Injector injector =
         Guice.createInjector(new TestPersisterModule());

   private final IPersister persister;

   public TestPersister() {
      persister = injector.getInstance(IPersister.class);
   }

   @BeforeClass
   public static void setupTable() {
      JDBCCommand createTable = new JDBCCommand() {
         @Override
         public String getStatement() {
            return "CREATE TABLE todos (id BIGINT NOT NULL, " +
                  "description VARCHAR(1024), duedate DATE, " +
                  "PRIMARY KEY (id))";
         }
      };
      injector.injectMembers(createTable);
      createTable.execute();
   }
   //...
```

Testtreiber anpassen

```
public class TestPersister {

   private static final Injector injector =
         Guice.createInjector(new TestPersisterModule());

   private final IPersister persister;

   public TestPersister() {
      persister = injector.getInstance(IPersister.class);
   }

   @BeforeClass
   public static void setupTable() {
      JDBCCommand createTable = new JDBCCommand() {
         @Override
         public String getStatement() {
            return "CREATE TABLE todos (id BIGINT NOT NULL, " +
                  "description VARCHAR(1024), duedate DATE, " +
                  "PRIMARY KEY (id))";
         }
      };
      injector.injectMembers(createTable);
      createTable.execute();
   }
   //...
```

Zusammenfassung des Beispiels

- Die direkte Abhängigkeit zur Implementierung des `IPersister`-Interface wurde entfernt.

Zusammenfassung des Beispiels

- Die direkte Abhängigkeit zur Implementierung des `IPersister`-Interface wurde entfernt.
- Die direkte Abhängigkeit zur `ConnectionFactory` – die `ConnectionFactory` selbst – wurde entfernt.

Zusammenfassung des Beispiels

- Die direkte Abhängigkeit zur Implementierung des `IPersister`-Interface wurde entfernt.
- Die direkte Abhängigkeit zur `ConnectionFactory` – die `ConnectionFactory` selbst – wurde entfernt.
- Alle Abhängigkeiten sind zentral im `TestPersisterModule` definiert.

Zusammenfassung des Beispiels

- Die direkte Abhängigkeit zur Implementierung des `IPersister`-Interface wurde entfernt.
- Die direkte Abhängigkeit zur `ConnectionFactory` – die `ConnectionFactory` selbst – wurde entfernt.
- Alle Abhängigkeiten sind zentral im `TestPersisterModule` definiert.
- Dependency Injection trennt die Instanziierung eines Objekts von seiner Nutzung.

Inhalt

- Einleitung und Motivation
- Abhängigkeiten und Dependency Injection
- Facts
  - JSR 330: Dependency Injection for Java
  - **JSR 299: Contexts and Dependency Injection for the Java EE Platform**
- Fictions and Dangers

JSR 299: Contexts and Dependency Injection for the Java EE Platform

- Dependency Injection für beliebige Java-Beans
- Verwendet die im JSR 330 definierten Injection-Techniken
- Bindung der Java-Beans an die Kontexte eines Containers (Request, Session, Application, …)
- Zusätzlich: Decorators, Interceptors und Event-basierte Kommunikation

JSR 299: Contexts and Dependency Injection for the Java EE Platform

- Dependency Injection für beliebige Java-Beans
- Verwendet die im JSR 330 definierten Injection-Techniken
- Bindung der Java-Beans an die Kontexte eines Containers (Request, Session, Application, …)
- Zusätzlich: Decorators, Interceptors und Event-basierte Kommunikation

- Referenzimplementierung des JSR 299: JBoss Weld
  - **Link:** http://seamframework.org/Weld

Dependency Injection und Expression Language

- Der Container injiziert Instanzen von beliebigen Objekten, von kontextabhängigen Objekten und von Ressourcen.

```
@Named
@RequestScoped
public class TodoProcessor {

   @Inject
   private ITodoDAO todoDAO;

   @Inject
   private Instance<ITodo> todoProvider;
```

```
<!-- ... -->
<h:inputText value=
   "#{todoProcessor.description}" />

<!-- ... -->
<h:commandButton action=
   "#{todoProcessor.addTodo}" value=
   "Add todo entry" />

<!-- ... -->
```

Dependency Injection und Expression Language

- Der Container injiziert Instanzen von beliebigen Objekten, von kontextabhängigen Objekten und von Ressourcen.
- Via **@Named** sind Bean-Instanzen im jeweiligen Kontext verfügbar und können, zum Beispiel in JSFs via Expression Language, genutzt werden.

```
@Named
@RequestScoped
public class TodoProcessor {

   @Inject
   private ITodoDAO todoDAO;

   @Inject
   private Instance<ITodo> todoProvider;
```

```
<!-- ... -->
<h:inputText value=
   "#{todoProcessor.description}" />

<!-- ... -->
<h:commandButton action=
   "#{todoProcessor.addTodo}" value=
   "Add todo entry" />

<!-- ... -->
```

Dependency Injection und Expression Language

- Der Container injiziert Instanzen von beliebigen Objekten, von kontextabhängigen Objekten und von Ressourcen.
- Via `@Named` sind Bean-Instanzen im jeweiligen Kontext verfügbar und können, zum Beispiel in JSFs via Expression Language, genutzt werden.
- Der Scope einer injizierten Instanz bestimmt, wann sie erzeugt wird, wem sie injiziert wird und wann sie zerstört wird.

```
@Named
@RequestScoped
public class TodoProcessor {

   @Inject
   private ITodoDAO todoDAO;

   @Inject
   private Instance<ITodo> todoProvider;
```

```
<!-- ... -->
<h:inputText value=
   "#{todoProcessor.description}" />

<!-- ... -->
<h:commandButton action=
   "#{todoProcessor.addTodo}" value=
   "Add todo entry" />

<!-- ... -->
```

Konfiguration

- JEE 6 DI ist **im Wesentlichen** konfigurationsfrei.
- `beans.xml`-Datei signalisiert dem Server, dass DI verwendet wird.
- Konfiguration von `@Alternative`, `@Decorator` und `@Interceptor`

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ... >
   <alternatives> ... </alternatives>

   <decorators> ... </decorators>

   <interceptors> ... </interceptors>
</beans>
```

Inhalt

- Einleitung und Motivation
- Abhängigkeiten und Dependency Injection
- Facts
  - JSR 330: Dependency Injection for Java
  - JSR 299: Contexts and Dependency Injection for the Java EE Platform
- **Fictions and Dangers**

Fictions

- Abhängigkeiten werden automatisch aufgelöst.
  → nicht immer: Mehrdeutigkeiten

Fictions

- Abhängigkeiten werden automatisch aufgelöst.
  → nicht immer: Mehrdeutigkeiten
- Benötigte Objekte werden automatisch injiziert.
  → nicht immer: Explizite Injektion, Laufzeit-Eigenschaften

Fictions

- Abhängigkeiten werden automatisch aufgelöst.
  → nicht immer: Mehrdeutigkeiten
- Benötigte Objekte werden automatisch injiziert.
  → nicht immer: Explizite Injektion, Laufzeit-Eigenschaften
- Abhängigkeiten werden beim Deployment geprüft.
  → nicht immer: Producer-Klassen

Fictions

- Abhängigkeiten werden automatisch aufgelöst.
  → nicht immer: Mehrdeutigkeiten
- Benötigte Objekte werden automatisch injiziert.
  → nicht immer: Explizite Injektion, Laufzeit-Eigenschaften
- Abhängigkeiten werden beim Deployment geprüft.
  → nicht immer: Producer-Klassen
- Dependency Injection ist typsicher.
  → Ja!

Fictions

- Abhängigkeiten werden automatisch aufgelöst.
  → nicht immer: Mehrdeutigkeiten
- Benötigte Objekte werden automatisch injiziert.
  → nicht immer: Explizite Injektion, Laufzeit-Eigenschaften
- Abhängigkeiten werden beim Deployment geprüft.
  → nicht immer: Producer-Klassen
- Dependency Injection ist typsicher.
  → Ja!
- Zusätzlicher Aufwand ist kaum nötig.
  → Doch: Auflösung der Abhängigkeiten, Objekt-Instanziierung, Initialisierung durch Injektion, Lernaufwand, Programmverstehen

Dangers

- Nutzung eines Objekts ohne offensichtliche **Initialisierung** widerspricht dem normalen Programmieren.

Dangers

- Nutzung eines Objekts ohne offensichtliche **Initialisierung** widerspricht dem normalen Programmieren.
- Die **Indirektion** erschwert es, die Implementierung einer Funktion nachzuvollziehen.

Dangers

- Nutzung eines Objekts ohne offensichtliche **Initialisierung** widerspricht dem normalen Programmieren.
- Die **Indirektion** erschwert es, die Implementierung einer Funktion nachzuvollziehen.
- Der **Konfigurationsmechanismus** muss bekannt sein. Wann wird injiziert?.

Fazit

- Die Konfiguration von Anwendungen – vor allem von JEE-6-Anwendungen – wird erleichtert und ist elegant.

Fazit

- Die Konfiguration von Anwendungen – vor allem von JEE-6-Anwendungen – wird erleichtert und ist elegant.
- Die Einarbeitung in die Konzepte und das Programmverstehen sind aufwändig, geübten Entwicklern sollte es aber möglich sein, den Überblick zu behalten.

Fazit

- Die Konfiguration von Anwendungen – vor allem von JEE-6-Anwendungen – wird erleichtert und ist elegant.
- Die Einarbeitung in die Konzepte und das Programmverstehen sind aufwändig, geübten Entwicklern sollte es aber möglich sein, den Überblick zu behalten.
- Die Typsicherheit und die frühe Abhängigkeitsprüfung sind vorteilhaft.

Fazit

- Die Konfiguration von Anwendungen – vor allem von JEE-6-Anwendungen – wird erleichtert und ist elegant.
- Die Einarbeitung in die Konzepte und das Programmverstehen sind aufwändig, geübten Entwicklern sollte es aber möglich sein, den Überblick zu behalten.
- Die Typsicherheit und die frühe Abhängigkeitsprüfung sind vorteilhaft.
- In der JSE (Java 7) ist der JSR 330 nicht enthalten; In der JEE 6 sind der JSR 299 und 330 enthalten, allerdings sind bisher nur zwei Server zertifiziert.

Inhalt

- Einleitung und Motivation
- Abhängigkeiten und Dependency Injection
- Facts
  - JSR 330: Dependency Injection for Java
  - JSR 299: Contexts and Dependency Injection for the Java EE Platform
- Fictions and Dangers

Markus Knauß

knauss@informatik.uni-stuttgart.de

Abteilung Software Engineering
Institut für Softwaretechnologie

Universität Stuttgart

http://www.iste.uni-stuttgart.de/se