Gesamtarchitektur I
und IoC

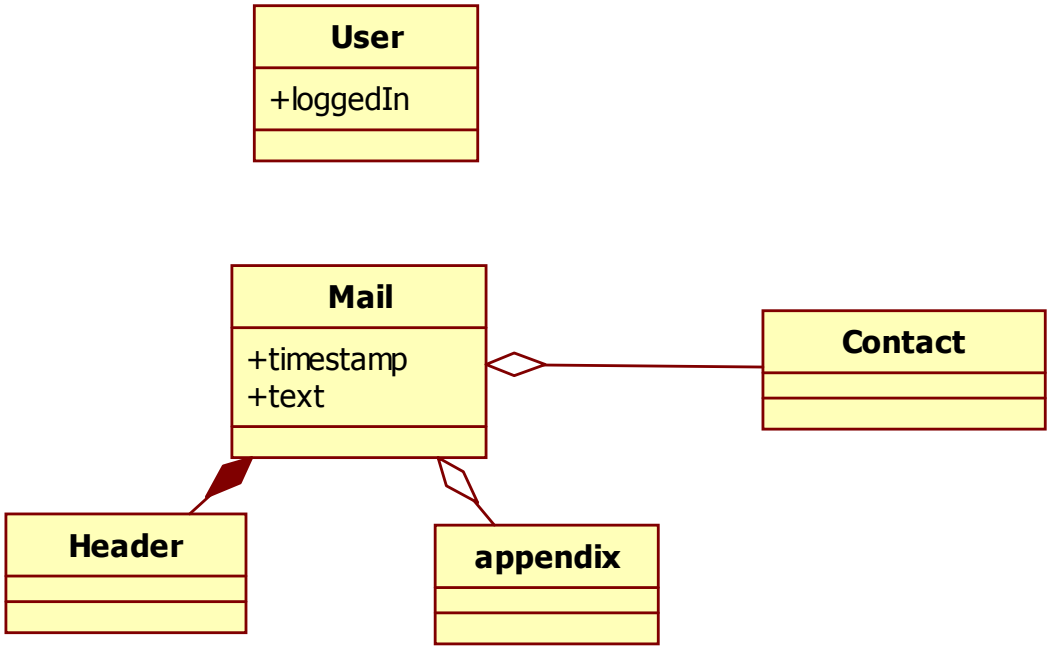
Schichten einer Web-Anwendung

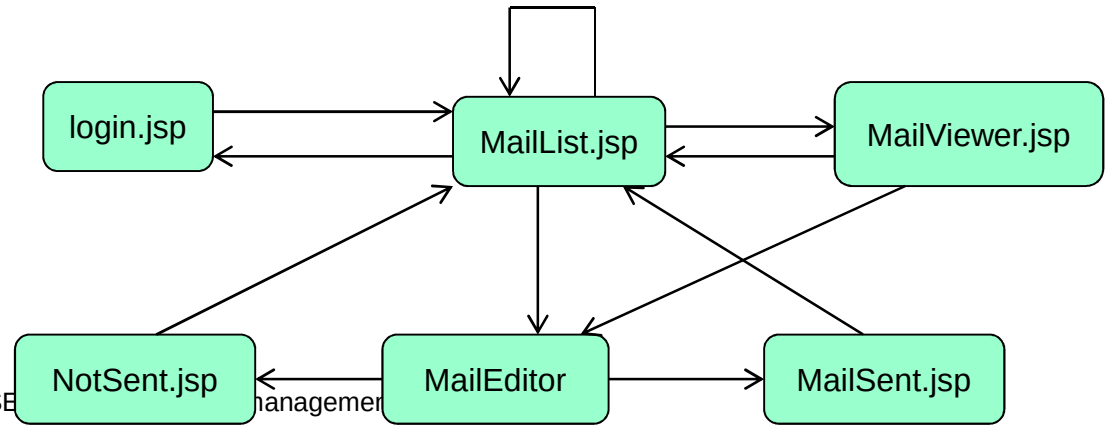
- Initiiert durch J2EE und Spring:
- Strukturierte Sicht auf UI und Fachlogik (Domäne)
- Ergibt 5 Schichten:

Man unterscheidet Präsentations- und Domänenmodell!

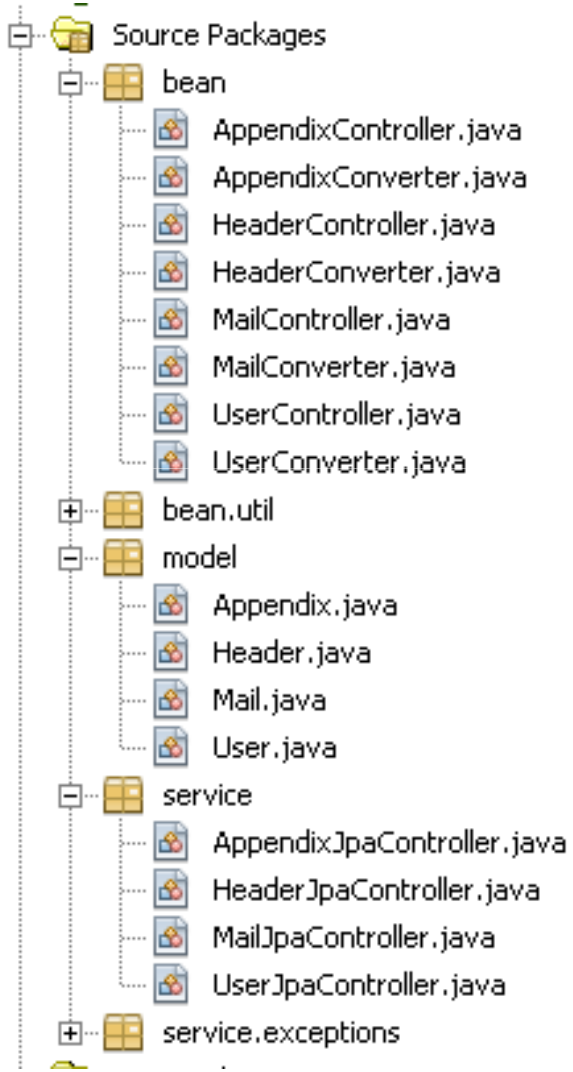
UI	Presentation V	**Sichtbare Komponenten**
	UI Application (MC)	**UI-Zusammenhang:** **Navigation, Workflow, Anwendungsobjekte, Use Cases, Session, Verbindung zu Services**
Geschäfts-modell	Services C	**Geschäftslogik und Transaktionen:** **Use-Cases, Objektübergreifende Operationen, Objektverwaltung (Suchen, Auflisten...)**
	Domain M	**Fachlogik** **Domänenmodell, persistente Einheiten**
Persistenz	Persistence	**Persistenzoperationen, Persistenzverwaltung.**

Mail-Service Analysemodell



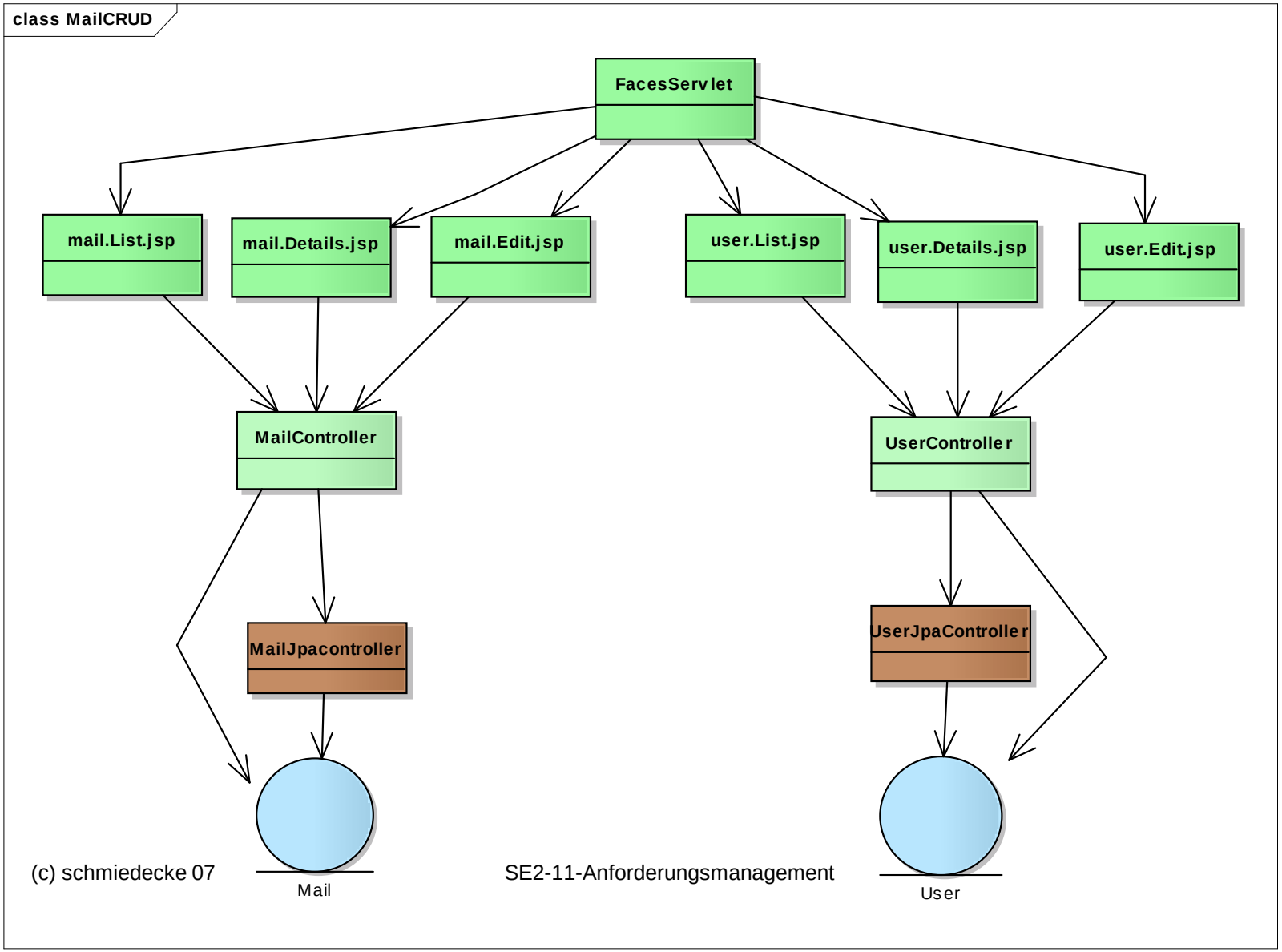


Mail-Service CRUD-Prototyp



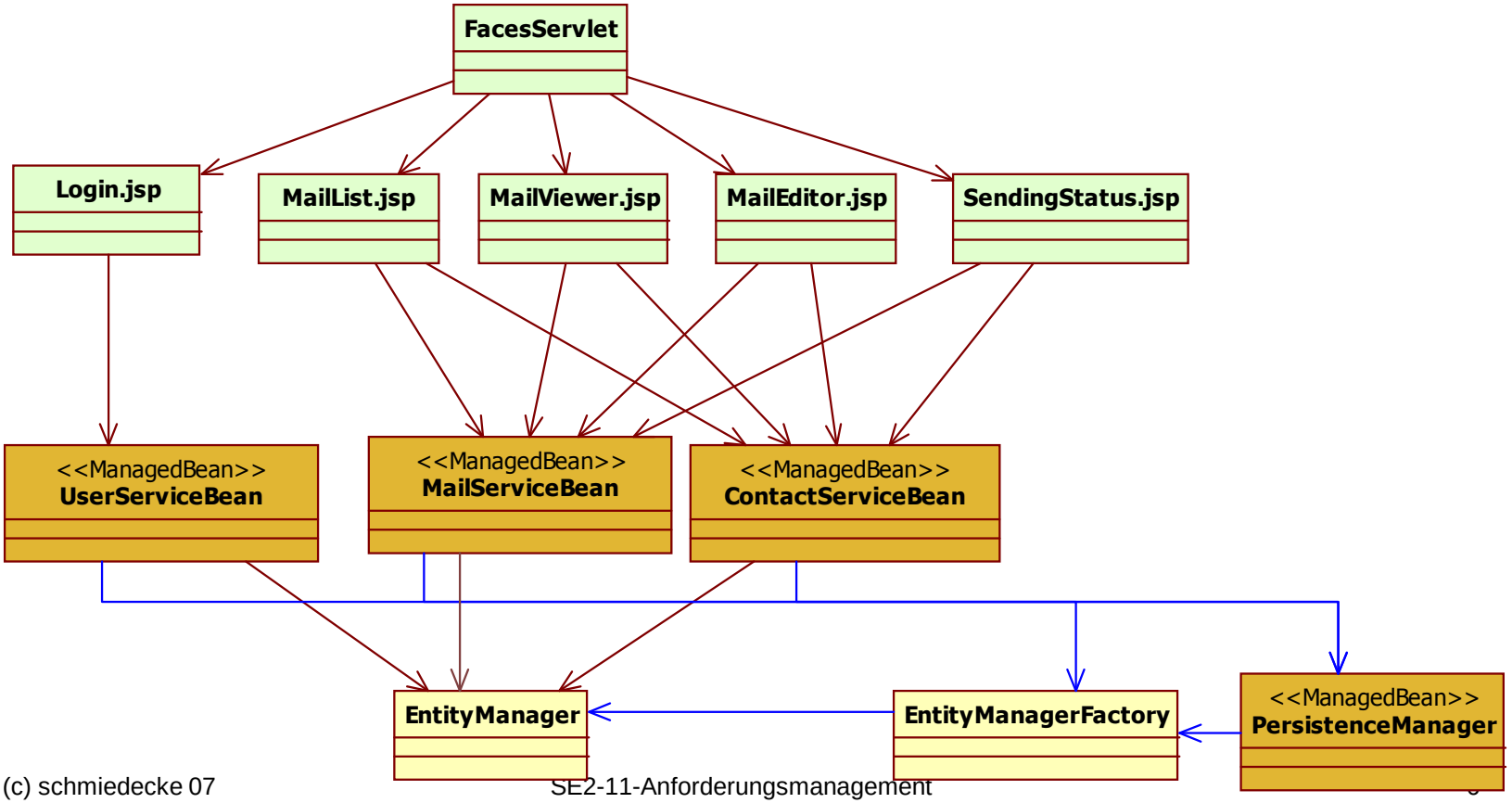
- Zu jeder Model-Klasse wird eine Controller-Klasse und eine JpaController-Klasse generiert.
- Die Controller-Klasse enthält alle Attribute und Methoden, die die JSPs benötigen.
  → Backing Bean
- Die JpaController-Klasse stellt den Zugang zu den Persistenten Objekten her.
  → Service

CRUD-Architektur

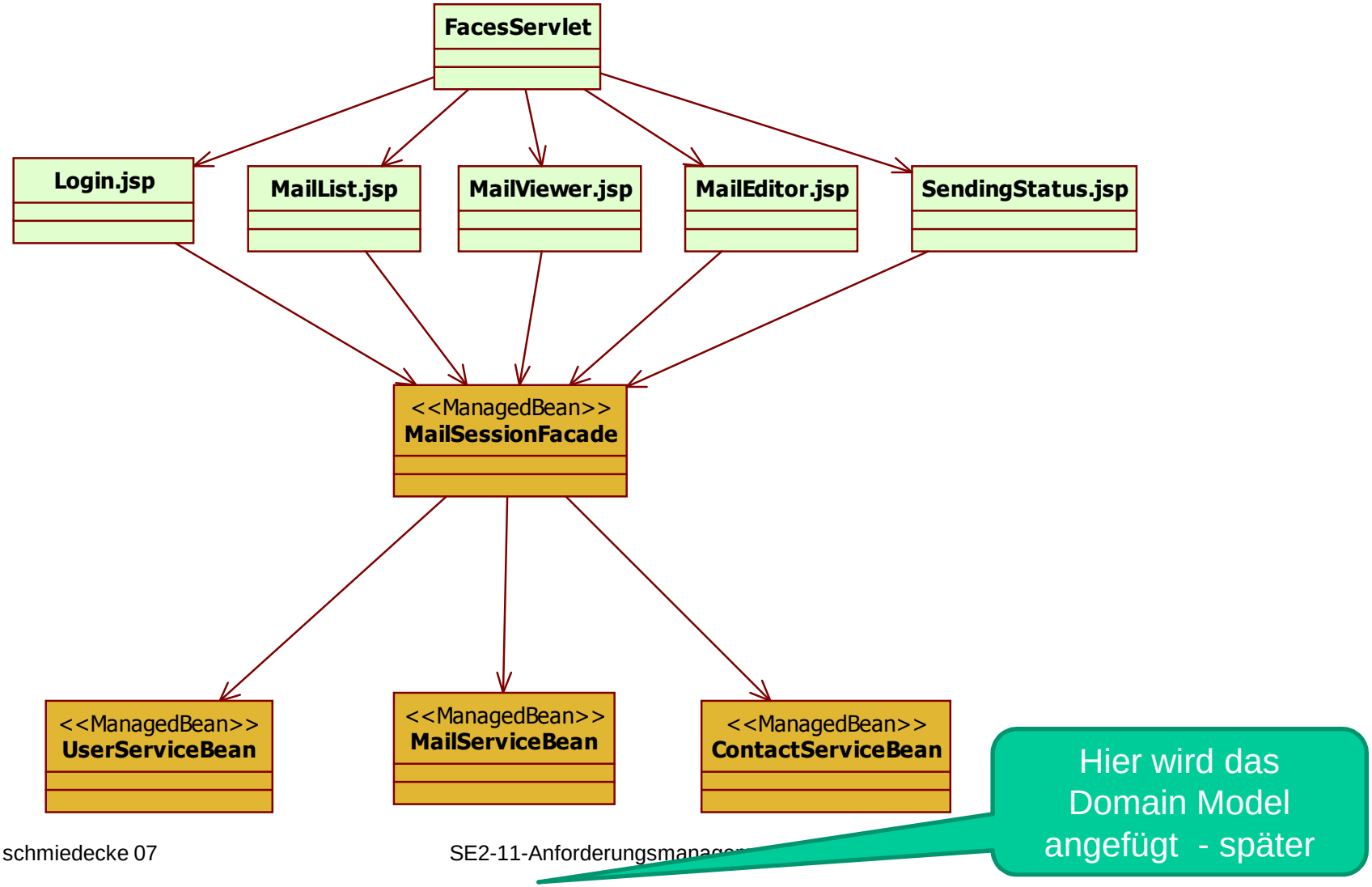


CRUD-Architektur ←→ Anwendungsarchitektur

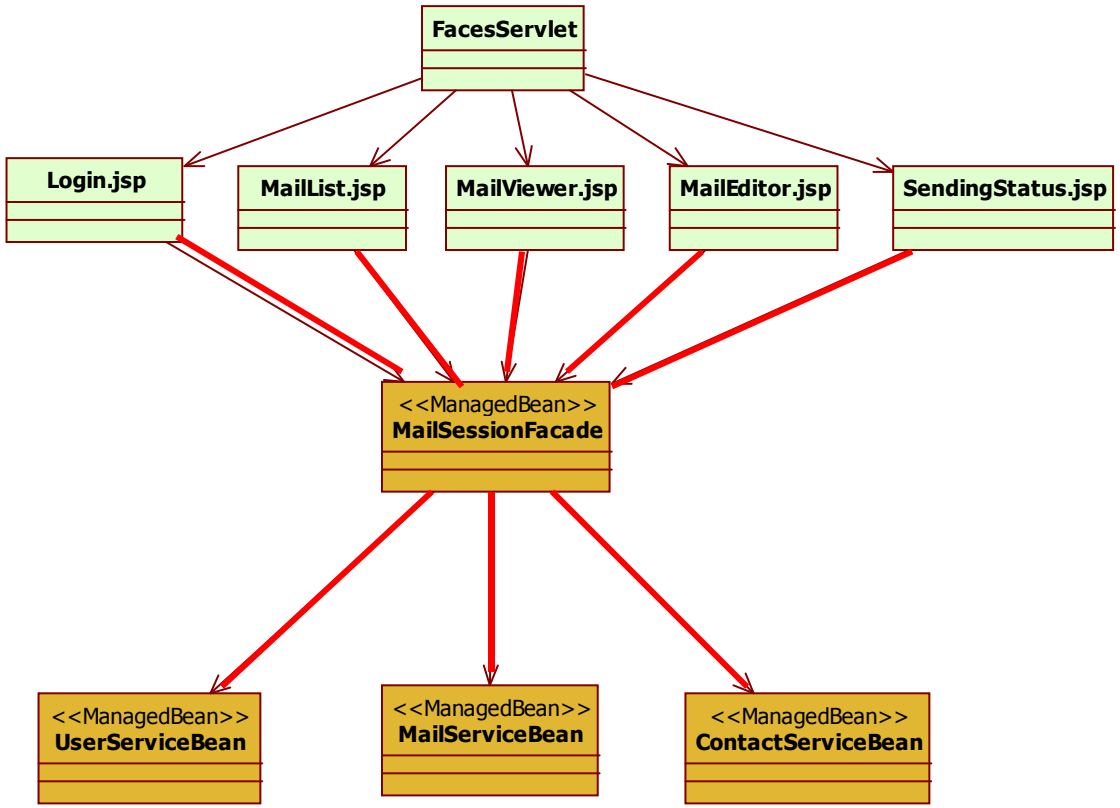
- „Echte“ GUIs sind an Use Cases ausgerichtet
- Services sind am Domain Model ausgerichtet
- → „Assoziationsgebüsch“



„Aufräumen“ mit einer Session Facade



Abhängigkeiten zwischen Schichten



- Rote Assoziationen können nicht „naiv“ gesetzt werden.
- Alternative? „Klebstoff“ benötigt ☺

IoC – Inversion of Control

- Objektmanagement durch die Umgebung
  - IoC-Framework
  - Oder als IoC-Pattern implementiert

- Service-Konzept:
  - Anwendung "bestellt" Objekt: "*brauche ContactServiceBean*"
  - Framework liefert, gemäß Spezifikation,
    - neues Objekt
    - aktuell verfügbares Objekt
    - allgemein veröffentlichtes Objekt
  - kümmert sich um Instanziierung, Verwaltung, Thread-Safety,...

- Beliefern → Dependency Injection

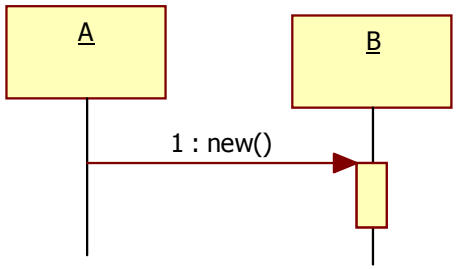
IoC - Klebstoff oder Trennmittel?

- Eine Frage der Sichtweise ☺

- Wenn man Klebstoff hat,
  muss man die Schichten nicht hart verdrahten:

→ IoC gilt als **Technik der Schichtentrennung**

Direkte Instanziierung – direkte Kontrolle

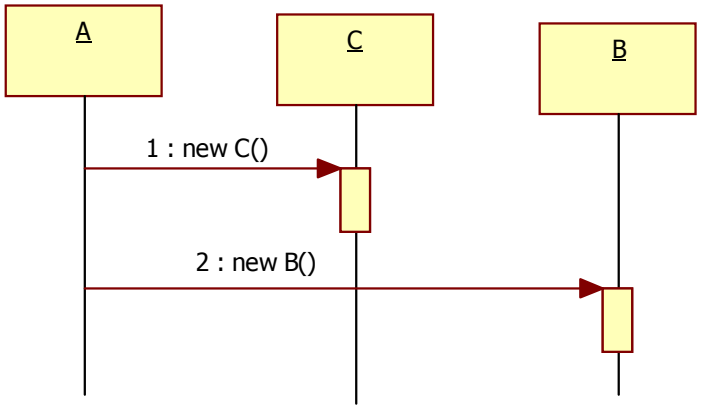



```
public class A {
    private B b;
    public A()
    { b = new B(); }
}
```

Entwurfsentscheidungen:

1. A benötigt Referenz auf B.
2. B ist eine konkrete Klasse mit Standard-Konstruktor.
3. A besitzt die Referenz auf B exklusiv.

Veränderungen an B wirken sich aus:

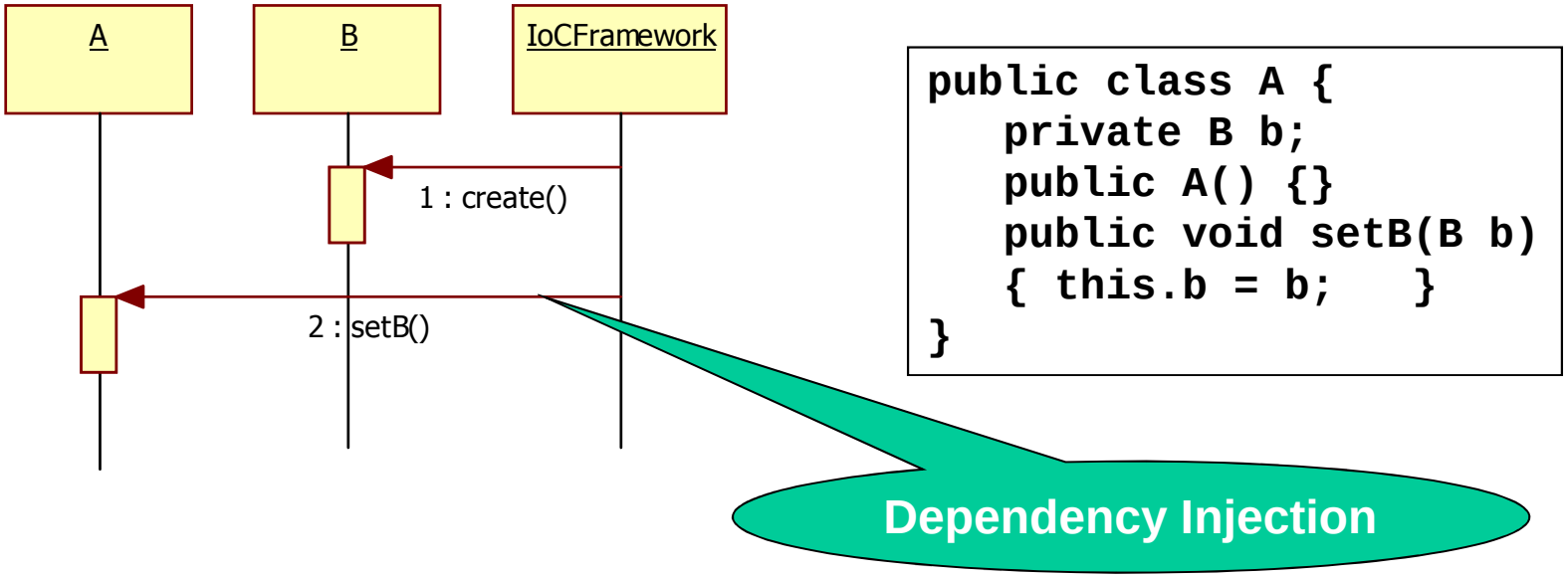



```
public class A {
   private B b;
   public A()
   {  C c = new C();
      b = new B(c);
   }
}
```

Geänderte Entwurfsentscheidung, z.B.:

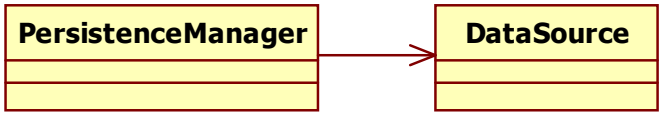
- kein Standardkonstruktor für B
- B ist abstrakte Klasse.

Umkehrung der Kontrolle: IoC-Framework erzeugt B-Instanz



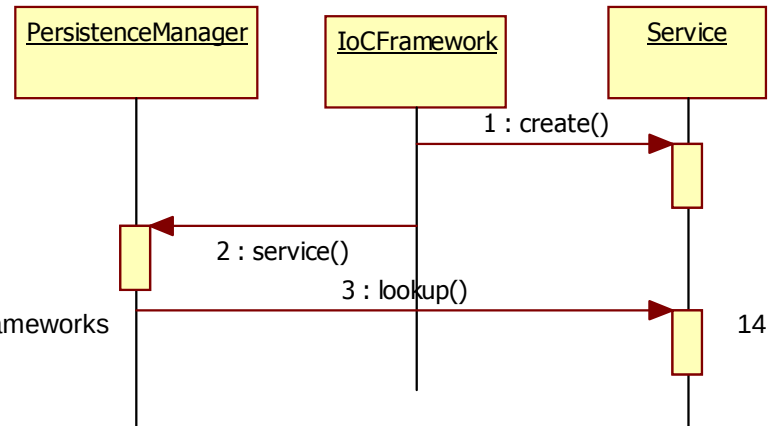
- Änderungen an B wirken sich nicht aus.
- B muss keine konkrete Klasse sein.
- A-Instanzen müssen vor Benutzung mit B-Referenz "versorgt" sein.

Dependency Injection – Typ 1
Interface Injection




```
import org.apache.avalon.framework.*;

public class PersistenceManager implements Serviceable {
    DataSource dataSource;
    public void service (ServiceManager sm)
                        throws ServiceException {
        dataSource = (DataSource)sm.lookup("dataSource");
    }
    public void getData() {
        // use dataSource
        // to do some work
    }
}
```

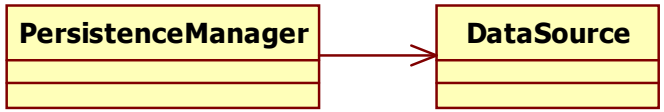


Diskussion Interface Injection

+ keine Konfiguration erforderlich.

- Java-Code ist framework-spezifisch.

Dependency Injection – Typ 2
Setter Injection




```
// Spring Framework: PersistenceManager ist normales Bean
public class PersistenceManager {
    DataSource dataSource;
    public void setDataSource(DataSource dataSource)
    { this.dataSource = dataSource;  }
    public void getData() {
        // use dataSource to do some work
    }
```

```
<bean id="dataSourceBean"  class=".....">
   <property name="driverClassName">
       <value>com.mydb.jdbc.Driver</value>
    </property>
    <property name="url">
       <value>jdbc:mydb://server:port/mydb</value
    </property>
    <property name="username">  <value>root</value>
    </property>
</bean>
```

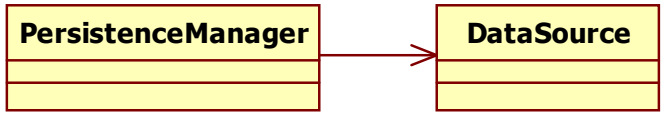
```
<bean id="persistenceManagerBean"
        class="example.PersistenceManager">
   <property name="dataSource">
       <ref bean="dataSourceBean"/>
    </property>
</bean>
```

Diskussion Setter Injection

+ Java-Code ist Framework-unabhängig
+ Testumgebung einfach in XML konfigurierbar (Dummy-Objekte)

- Die Abhängigkeit ist im Code nicht erkennbar.
- Abhängigkeiten können nicht zur Compilezeit validiert werden.
- DataSource muss public sein – würde man eigentlich gern verbergen.

Dependency Injection – Typ 3
Constructor Injection




```
// PicoContainer Framework: keine Konfiguration

public class PersistenceManager {
   DataSource dataSource;
   public PersistenceManager(DataSource dataSource)
   { this.dataSource = dataSource;  }
   public void getData() {
      // use dataSource to do some work
   }
}

// die Klasse wird nicht direkt instanziiert,
// sondern vom PicoContainer "besorgt".
// Vorher muss sie unter "key" registriert sein, s.u.

pico.getComponentInstance(key);
```

Dependency Injection – Typ 3
Constructor Injection

```
// PicoContainer Framework: programmierte Abhängigkeit

   // Datasourcebean erzeugen:
   JDBCDataSource dataSource = new JDBCDataSource();
   dataSource.setDriverClassName("com.mydb.jdbc.Driver");
   dataSource.setUrl("jdbc:mysql://localhost:3306/mydb");
   dataSource.setUsername("Jacob");

   //IoC-Container erzeugen
   MutablePicoContainer pico = new DefaultPicoContainer();

   // Komponenten registrieren
   ConstantParameter dataSourceParam =
                     new ConstantParameter(dataSource);
   String key = "JDBCPersistenceManager";
   Parameter[] params = {dataSourceParam};
   pico.registerComponentImplementation
         (key, PersistenceManager.class, params);
```

Diskussion Constructor Injection

+ Java-Bean-Code ist Framework-unabhängig
+ Good-Citizen-Pattern: Instanziierte Objekte sind "komplett"
+ Die Abhängigkeit ist im Code erkennbar.

- Vererbung bringt Komplikationen (Constructor chaining).
- Manche APIs erfordern Standard-Konstruktor.

Wichtige IoC-Frameworks

- Spring
  - Setter Injection (auch Constructor Injection möglich)
  - reichhaltige Bibliothek
  - Standard-Hibernate-Anschluss
  - Ausbaustufen für MVC, AOP
- PicoContainer
  - minimal, reiner IoC-Container
  - Constructor Injection
- Avalon
  - frühes Projekt, wird nicht mehr fortgesetzt
  - Interface Injection
- HiveMind

Welches IoC-Framework für JSF-Anwendungen?

- Mit allen kombinierbar.
- IoC-Pattern kann auch selbst implementiert werden.
- **Managed Beans sind bereits ein IoC-Framework!**

Managed Beans als IoC-Framework:

Instanziierung:

- Managed Beans werden vom Container instanziiert.

Initialisierung:

- Initialisierungen von Properties können in der faces-config.xml angegeben werden.
- Auch als Assoziationen zu anderen Managed Beans

```
<value>#{otherbean}</value
```

Gültigkeit und Lebensdauer:

- Scope-Typen
- none, request, session, application
- Lebensdauer wird in faces-config.xml spezifiziert.

Bean-Konfiguration: IoC

```
<managed-bean>
    <managed-bean-name>sessionFacade</managed-bean-name>
    <managed-bean-class>mail.ui.MailSessionFacade</managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
    <managed-property>
      <property-name>readingService</property-name>
      <value>#{readingService}</value>
    </managed-property>
     <managed-property>
      <property-name>writingService</property-name>
      <value>#{writingService}</value>
    </managed-property>
    <managed-property>
      <property-name>addressbookService</property-name>
      <value>#{addressbookService}</value>
    </managed-property>
  </managed-bean>
```

… genug für heute ☺

Nächste Woche stellen wir
die Architektur fertig
und fügen Authentifizierung hinzu.