Dependency Injection in der Praxis: Spring, PicoContainer und Eclipse im Vergleich

Dipl.-Informatiker Martin Lippert
Senior IT-Berater
martin.lippert@it-agile.de
http://www.it-agile.de/

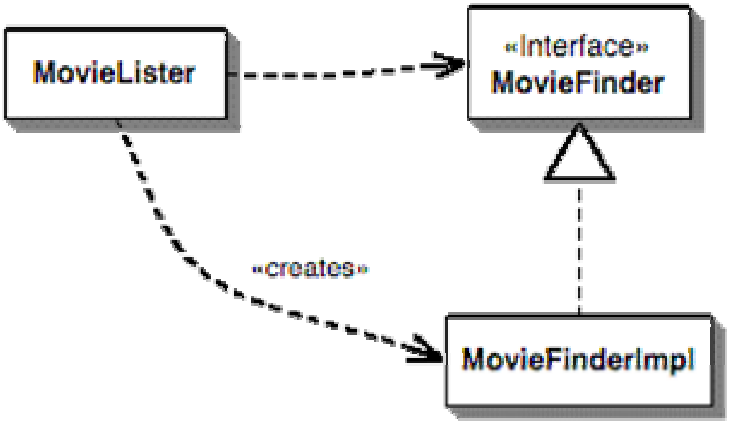
Überblick

- Motivation
- Dependency Injection
  - Was ist das?
  - Wie funktioniert das?
- Dependency-Injection-Frameworks im Vergleich:
  - PicoContainer
  - Spring
  - Eclipse
- Fazit

Motivation

- Unsere Software wollen wir ...
  - ... gut modularisieren
  - ... leicht verändern können
  - ... einfach testen können

- Was finden wir aber in der Praxis?
  - Viele Abhängigkeiten
  - Eng verzahnte Systemteile
  - Wenig definierte Schnittstellen
  - „Evil Singletons"



Klassiker: Direkte Erzeugung

- Es werden Interfaces benutzt, aber die Implementation wird trotzdem direkt erzeugt

```
public class MovieLister {
  private MovieFinder finder;

  public MovieLister() {
    finder = new ColonDelimitedMovieFinder("movies1.txt");
  }

  ...
}
```

Klassiker: Singleton

- Gerne und häufig genutztes, aber in Verruf geratenes Pattern
- Typische Probleme:
  - Singletons sind schwer zu testen
  - Singletons auf dem Server sind durch die J2EE-Spezifikation verboten
  - Singletons auf dem Server müssen Thread-Safe programmiert sein
  - Dynamischer Austausch der Implementierung schwierig, z.B. für Tests mit Mock-Types (EasyMock)

Dependency Injection

- **Abhängigkeiten explizit machen!!!**
- **Separate Configuration from Use!!!**

- Mit dem Dependency-Injection-Muster werden die Abhängigkeiten zwischen den Objekten von außen gesteuert

- Die häufigsten zwei Varianten:
  - Constructor Injection
  - Setter Injection
  - Interface Injection

Constructor Injection

- An der Schnittstelle (z.B. im Konstruktor) werden die benötigten Objekte mitgegeben

```
public class MovieLister {
    private MovieFinder finder;

    public MovieLister(MovieFinder finder) {
        this.finder = finder;
    }

    ...
}
```

Setter Injection

- Die Klasse besitzt Setter, um anhängige Komponenten bekannt zu machen

```
public class MovieLister {
    private MovieFinder finder;

    public void setFinder(MovieFinder finder) {
        this.finder = finder;
    }

    ...
}
```

Interface Injection

- Es wird ein Interface definiert, um die Injection zu kennzeichnen

```
public interface InjectFinder {
    void injectFinder(MovieFinder finder);
}
```

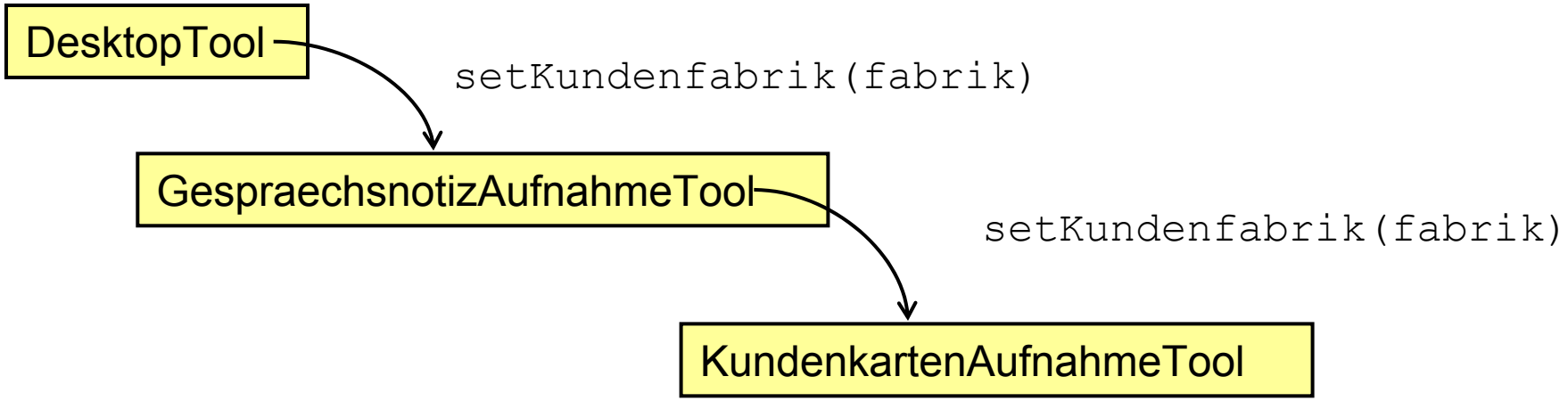
- Die entsprechenden abhängigen Klasssen implementieren das Interface

```
public class MovieLister implements InjectFinder...
    public void injectFinder(MovieFinder finder) {
        this.finder = finder;
    }
```

Folgen

Ein Objekt bekommt alles, was es braucht, gesetzt (z.B. im Konstruktor).

- Probleme:
  1. Das Objekt, das das betroffene Objekt erzeugt, muss alle Objekte, die dieses braucht, kennen.
  2. Das führt zum Durchschleifen von Parametern und langen Parameterlisten.



Abhilfe: Dependency-Injection-Frameworks

- Bei Dependency-Injection-Frameworks wird die Erzeugung fast aller Objekte von einer anderen Instanz erledigt, die alle benötigten Objekte kennt oder weiß, wie diese zu erzeugen sind.

- Verwalter von Singletons:
  - Diese erzeugen ein neues Exemplar falls noch keines vom geforderten Typ vorhanden ist und geben dann immer dieses eine Exemplar zurück. Diese können dazu verwendet werden die Objekte zu verwalten, die bisher als Singletons verfügbar waren.
- Verwalter gewöhnlicher Objekte:
  - Diese geben immer neue Objekte heraus und können dort verwendet werden, wo bisher über einen Konstruktoraufruf neue Objekte erzeugt wurden.

Open-Source-Frameworks

- PicoContainer
  - http://picocontainer.codehaus.org/
- Spring
  - http://www.springframework.org/
- Eclipse
  - http://www.eclipse.org/

- Weitere?
  - Hivemind
  - NanoContainer
  - ...

PicoContainer: Konstruktion der Klassen

Die Klassen, von denen der PicoContainer Objekte erzeugen soll, müssen einen Konstruktor haben, der als Parameter alles aufnimmt, was von dem Objekt später gebraucht wird.

```
public class KundenFabrik
{
  public KundenFabrik(PersistenceConfiguration config)
  {
    assert config != null :
        „Precondition: config != null“;
    _persistenceConfiguration = config;
  }
}
```

Der PicoContainer wird diesen Konstruktor aufrufen. Wenn er die benötigten Objekte noch nicht hat, wird er sie erzeugen.
Wenn es mehr als einen Konstruktor gibt, nimmt der PicoContainer den größten den er befüllen kann.
Empfehlung: Nur einen Konstruktor.

PicoContainer: Registrierung

An einem PicoContainer können Klassen und Objekte registriert werden.

```
picoContainer.registerComponentImplementation(
  KundenFabrik.class);
```

Registriert eine Klasse, von der bei Bedarf ein Objekt erzeugt wird

```
picoContainer.registerComponentImplementation("altDaten"
  AltDatenService.class);
```

Registriert eine Klasse unter einem bestimmten Key.

```
AndereFabrik fabrik = new AndereFabrik("test");
picoContainer.registerComponentInstance(fabrik);
```

Registriert ein vorkonfiguriertes Exemplar mit seiner Klasse als Key.

PicoContainer: Abfrage von Objekten

Von einem PicoContainer können Objekte abgeholt werden.

```
KundenFabrik kundenFabrik = (KundenFabrik)
  picoContainer.getComponentInstanceOfType(
  KundenFabrik.class);
```

Holt eine Exemplar anhand des Typs.
Berücksichtigt Vererbung und Interfaces.

```
AltDatenService service = (AltDatenService)
  picoContainer.getComponentInstance("altDaten");
```

Holt eine Exemplar anhand eines Key.

PicoContainer: Erzeugung von PicoContainern

Ein PicoContainer wird bei der Erzeugung konfiguriert.

```
MutablePicoContainer pc1 = new DefaultPicoContainer();
```

Erzeugt einen PicoContainer, der Singletons verwaltet

```
MutablePicoContainer pc2 = new DefaultPicoContainer(new
  ConstructorInjectionComponentAdapterFactory());
```

Erzeugt einen PicoContainer, der jedes Mal ein neues Exemplar erzeugt

PicoContainer: Eltern-Beziehung

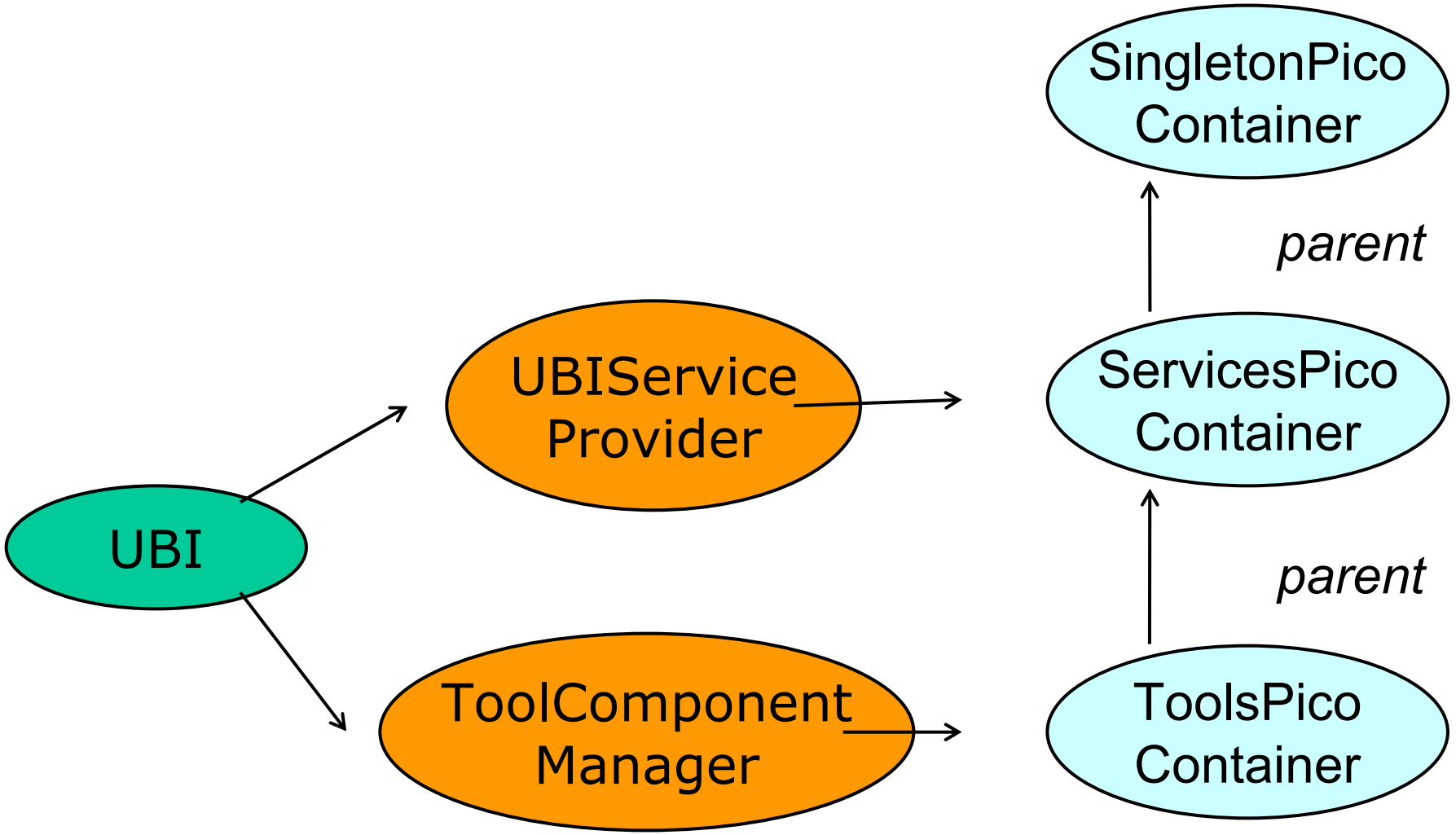
```
MutablePicoContainer pc1 = new DefaultPicoContainer();

MutablePicoContainer pc2 = new DefaultPicoContainer(new
 ConstructorInjectionComponentAdapterFactory(), pc1);
```

pc1 ist Parent von pc2. Wenn ein Objekt, das von pc2 erzeugt wird, ein Singleton braucht, das in pc1 registriert ist, wird pc2 pc1 nach diesem Objekt fragen und es bekommen.

Dadurch lassen sich Hierarchien von PicoContainern mit verschiedenen Aufgabenbereichen erzeugen.

Der Parent eines PicoContainers kann nach der Konstruktion nicht mehr geändert werden und ein PicoContainer kann nur einen Parent haben.

PicoContainer: Aus einem Projekt



Spring: Überblick

- Leichtgewichtiges Opensource-J2EE Framework
- Ziel: Entwicklung von J2EE-Anwendungen vereinfachen (z.B. im Vergl. zu EJB)
- Integriert existierenden Technologien statt eigener Lösungen
- Unabhängig vom eingesetzten Application Server
- Anwendungscode ist (meist) nicht von Spring APIs abhängig
- Erste Version im Oktober 2003 erschienen, aktuell Version 1.2.1
- Modularer Aufbau aus unabhängig einsetzbaren Paketen

Spring: Pakete

- Spring Core: Basisinfrastruktur, Unterstützung für Beans, DI
- Spring AOP: Unterstützung für deklarative Dienste
- Spring DAO: DB-Zugriff mit JDBC oder DAOs
- Spring ORM: Integration von Hibernate, JDO
- Spring Transaction: Unterstützung für Transaktions Management
- Spring Web: Anbindung an bestehende Frontend-Technologien (Struts, JSF, …)
- Spring MVC: Eigenes Web-Framework
- …

Spring: Pakete

- Spring Core: Basisinfrastruktur, Unterstützung für Beans, **DI**
- Spring AOP: Unterstützung für deklarative Dienste
- Spring DAO: DB-Zugriff mit JDBC oder DAOs
- Spring ORM: Integration von Hibernate, JDO
- Spring Transaction: Unterstützung für Transaktions Management
- Spring Web: Anbindung an bestehende Frontend-Technologien (Struts, JSF, …)
- Spring MVC: Eigenes Web-Framework
- …

Spring: Dependency Injection

- Spring Framework benutzt DI als Kernkonzept
  - Constructor Injection
  - Setter Injection

- Die Abhängigkeiten zwischen Komponenten werden über Konfigurationsdateien festgelegt und durch das Framework aufgelöst

- Kernkomponente dazu ist im Spring Framework der Bean Container
- Er verwaltet und erzeugt Komponenten, die zusammen die Anwendung sind
- Alle Komponenten sind normale Java Beans
- Container verwaltet Abhängigkeiten zwischen Beans und konfiguriert sie

Spring: Dependency Injection

- 2 Arten von Bean Containern:
  - Bean Factory: Grundlegende Unterstützung für DI, Konfiguration und Verwaltung von Beans
  - Application Context: Erweiterung der BF um Ressource Management, Eventing, i18n
- Bean Container unterstützen 2 Arbeitsweisen
  - Singleton: ein gemeinsam genutztes Exemplar wird zurückgeliefert (default)
  - Prototype: bei jeder Anfrage wird ein neues Exemplar erzeugt
- Am häufigsten eingesetzte Implementierung der BeanFactory: XmlBeanFactory
  - Liest Bean-Konfiguration aus XML-Datei ein
  - Format ist in „spring-beans.dtd" festgelegt
  - Alternative: DefaultListableBeanFactory benutzt Property-Dateien

Spring: Code Beispiel

- Ohne DI müsste das UserDao direkt im UserManager erzeugt werden

```
public class UserManager {
    private UserDAO _dao;
    public UserManager() {
        _dao = new UserDAOHibernate();
    }
}
```

- UserDao wird per Setter Injection bei der Erzeugung des Objektes gesetzt:

```
public class UserManager {
    private UserDAO _dao;
    public void setUserDAO(UserDAO dao) {
        _dao = dao;
    }
}
```

Spring: Code Beispiel

- Die dazugehörige XML-Konfigurationsdatei:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
   "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <bean id="userManager" class="UserManager" singleton="true">
       <property name="userDAO"><refbean="userDAO"/></property>
    </bean>
    <bean id="userDAO" class="dao.hibernate.UserDAOHibernate">
   […]
   </bean>
</beans>
```

- Die Anpassung der Konfigurationsdatei reicht aus, um die Implementierung des Daos auszutauschen

Spring: Erfahrungen

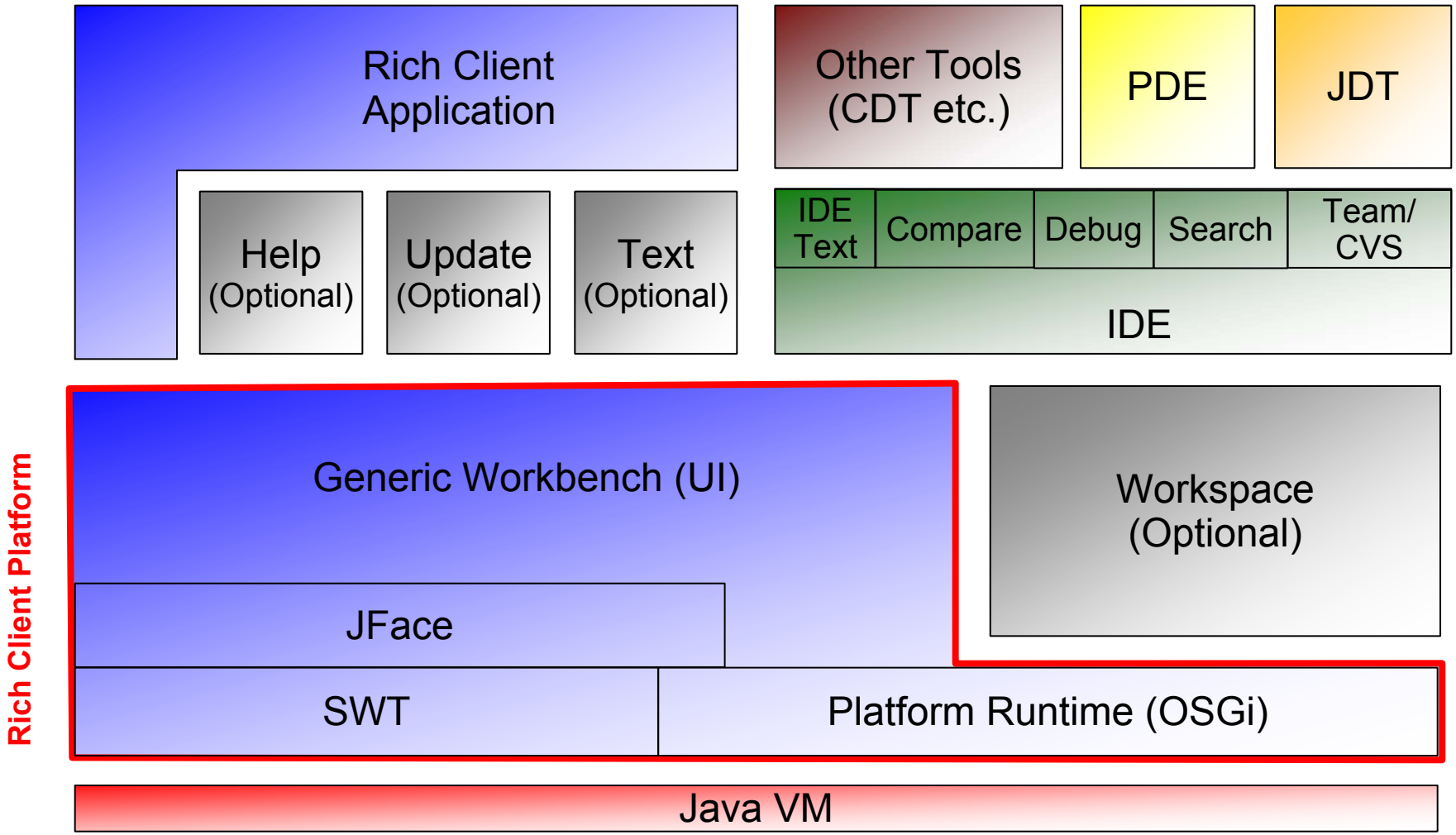
Negative:

- Verzicht auf Single-Source-Prinzip macht ständiges springen zwischen Java und XML notwendig
- Ohne Werkzeugunterstützung (z.B. Eclipse-Plugin) bleiben Tippfehler häufig unerkannt und führen zu Fehlern während der Laufzeit
- Einstieg in Spring ist insgesamt nicht ganz einfach

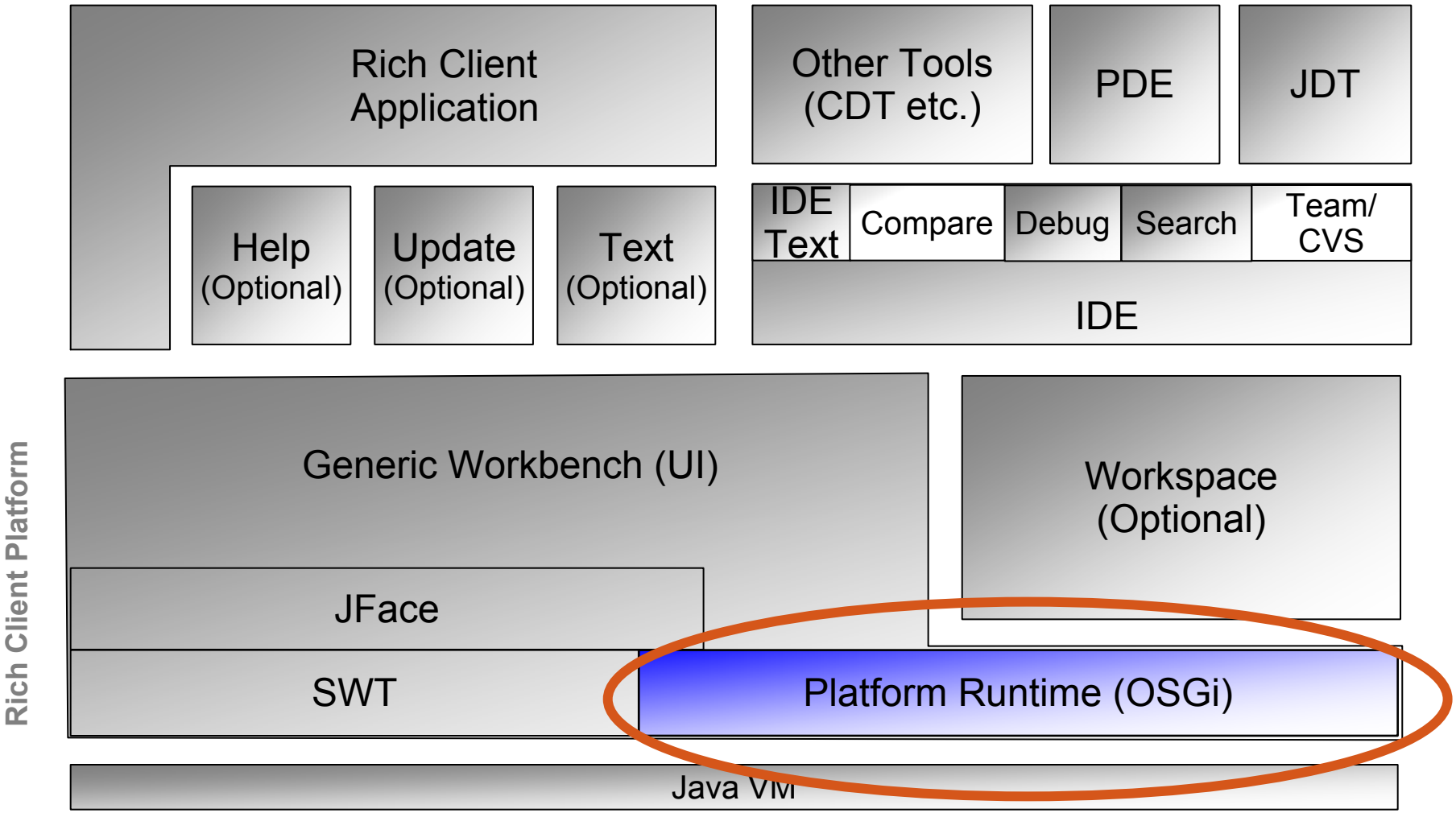
Positive:

- Änderungen an der Konfiguration sehr leicht möglich
- Bisher keine Probleme, die sich nicht per DI lösen ließen
- Praktisch kein erhöhter Entwicklungsaufwand durch den Einsatz von DI

Eclipse: Überblick



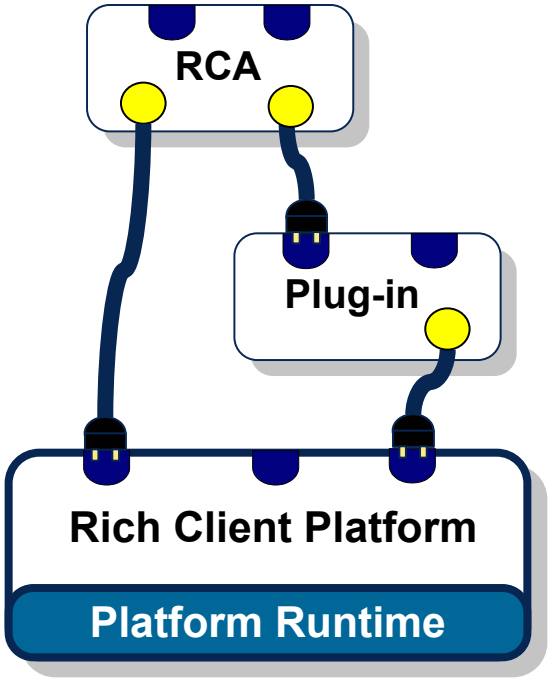
Eclipse: Überblick



Eclipse: Platform Runtime

- Erlaubt die Definition von Plugins (Modulen)
  - Inklusive Abhängigkeiten
- Definiert zusätzlich einen Erweiterungsmechanismus:

  Extension-Points und Extensions !!!



Eclipse: Extensions vs. Dependencies

it-agile

Sind Extensions/Extension-Points so etwas wie Dependency Injection?

- Ja, weil:
  - Das Plugin, welches den Extension-Point definiert, kennt nur ein Interface, nicht die konkrete Implementation
  - Die konkrete Implementation kann ausgetauscht werden, indem ein anderes Extension-Plugin verwendet wird
- Nein, weil:
  - Der Extension-Mechanismus nicht dafür gedacht ist, nur Abhängigkeiten zu entkoppeln
  - Ein Extension-Point definiert einen **Erweiterungs**punkt im System, keine **benötigte** Abhängigkeit
  - Ein Extension-Point sollte auch mit mehreren Extensions umgehen können

Fazit

- Dependency Injection ≠ PicoContainer, Spring, Eclipse …
  - Es ist ein generelles Entwurfsmuster
  - Dient der Entkopplung von Systemteilen
  - Kann auch ohne jedes Framework sinnvoll genutzt werden
- Aber: DI-Frameworks erleichtern aber die Arbeit

- Positive Effekte:
  - Testgetriebene Entwickung und Unit-Testing wird vereinfacht
  - Abhängigkeiten werden reduziert
  - Konkrete Implementationen können leicht ausgetauscht werden

- Negative Effekte:
  - Es ist aus dem Source-Code nicht mehr sofort zu erkennen, wo welches Objekt erzeugt wird (Lesbarkeit)
  - XML-Konfigurationen führen zu getrennten Quellen

Referenzen

- Martin Fowler:
  Inversion of Control Containers and the Dependency Injection pattern
  - http://www.martinfowler.com/articles/injection.html
- Robert C. Martin:
  Just Create One - The case against Singleton
  - http://www.butunclebob.com/ArticleS.UncleBob.SingletonVsJustCreateOne
- Folien-Unterlagen auf den it-agile Blogs:
  - http://entwicklerblog.it-agile.de
  - http://managerblog.it-agile.de

Danke für die Aufmerksamkeit

Fragen, Anmerkungen, Feedback?

Jederzeit!

Dipl.-Informatiker Martin Lippert
Senior IT-Berater
martin.lippert@it-agile.de