oose.
Innovative Informatik

Beziehungskiste.

Abhängigkeiten im Softwareentwurf planen und überwachen

Stefan Zörner (sz@oose.de)

Berlin, den 05. April 2013
Berlin Expert Days

BED BERLIN EXPERT DAYS 2013
{join the talk}

Stefan Zörner:
„Beziehungskiste:
Abhängigkeiten im Softwareentwurf planen und überwachen

Das Planen von Beziehungen zwischen Bausteinen ("Wer mit wem?") ist zentraler Bestandteil der Strukturierung eines Softwaresystems. Hier werden grundlegende Entscheidungen getroffen, die auf Wartbarkeit, Portierbarkeit etc. signifikante Auswirkung haben können. Leider oft negative. In vielen Fällen wird das Abhängigkeitsmanagement nicht bewusst betrieben, oder das Einhalten der Architektur nicht überwacht.

In diesem Vortrag stelle ich diesen Problemraum speziell für die Java-Welt vor. Sie erleben an einem nachvollziehbaren Beispiel, wie Abhängigkeiten für ein Java-System entworfen und überwacht werden können. Im Anschluss können Sie auch den Einsatz von Frameworks und Komponentenmodellen in diesen Kontext einordnen. Als Hilfsmittel kommen Methodik aus der Softwarearchitektur zum Einsatz, und aktuelle, frei verfügbare Tools, die es Ihnen auch im Team und ohne zentrale Architektenrolle ermöglichen, entscheidende Ziele zu erreichen.

Stefan Zörner :: sz@oose.de

- seit 2006 Berater und Trainer bei oose
- Vorher IBM, Mummert + Partner, Bayer AG, …

Schwerpunkte:

- Softwarearchitektur (Entwurf, Bewertung, Dokumentation)
- Java Technologien

sz@oose.de ::@StefanZoerner :: szoerner@apache.org

Agenda

1. Warum Abhängigkeiten planen?
2. Abhängigkeiten in Java
3. Werkzeuge
4. Fazit + Ausblick

Agenda

1

1 **Warum Abhängigkeiten planen?**

2 Abhängigkeiten in Java

3 Werkzeuge

4 Fazit + Ausblick

Abhängigkeiten in Java.
oose. Innovative Informatik

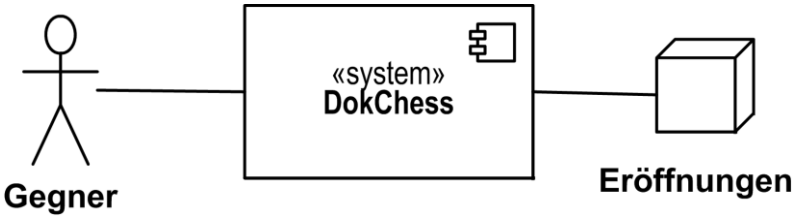
Beispiel: Ein Schach-Programm

Zentrale Features

- Text-basiertes UI (reine Schach-Engine)
- Spiel gegen menschliche und Computergegner
- Vollständige Umsetzung der offiziellen FIDE-Schachregeln
- Beherrscht taktische Elemente wie Gabel und Spieß
- Anbindung von Eröffnungsbibliotheken
- Implementierung in Java

Das Schach-Programm in Aktion

Schach-Programm, Systemkontext



Nehmen wir mal an …

… Ihr wollt, dass die Engine auch nach anderen als den Standardregeln spielen kann, z.B.

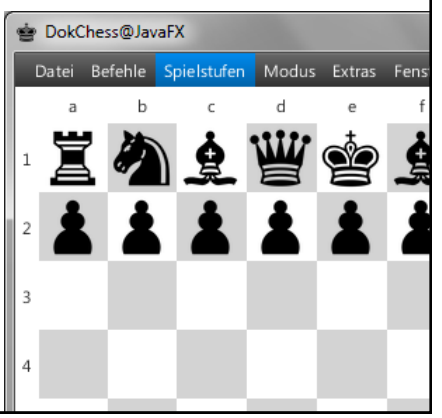
- Schach 960
- Räuber-Schach
- Atom-Schach
- …

Abhängigkeiten in Java.
oose. Innovative Informatik

Nehmen wir mal an …

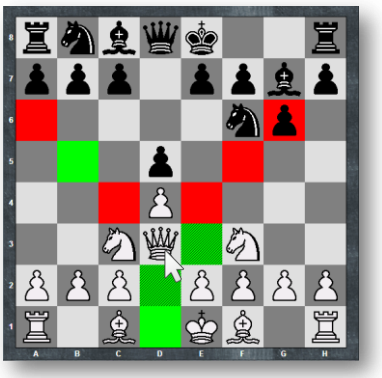
… Ihr wollt das textbasierte Frontend durch ein grafisches ersetzen, zum Beispiel realisiert mit JavaFX …

JavaFX



Nehmen wir mal an …

… Ihr wollt die Überprüfung auf gültige Züge in einer in Java realisierten graphischen Oberfläche zur Visualisierung von Hilfen wiederverwenden …



Abhängigkeiten in Java.
oose. Innovative Informatik

Nehmen wir mal an …

… Ihr wollt effizientere Datenstrukturen benutzen, um die Ermittlung des besten Zuges zu beschleunigen, oder in gleicher Zeit weiter vorausschauen zu können …

Nehmen wir mal an …

… Ihr wollt das Schach-Programm oder Teile davon auf Android portieren …

android

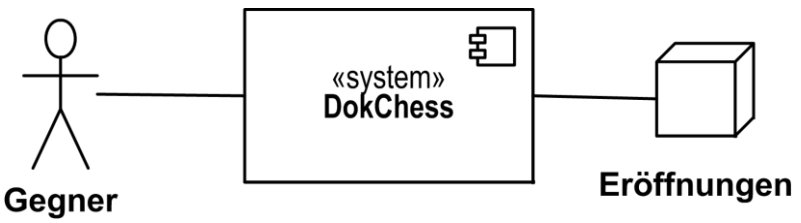
Geforderte Qualitätsmerkmale

Wartbarkeit
Austauschbarkeit
Änderbarkeit
Erweiterbarkeit
Portierbarkeit
Testbarkeit
Wiederverwendbarkeit
Anpassbarkeit

All diese konkreten Anforderungen …

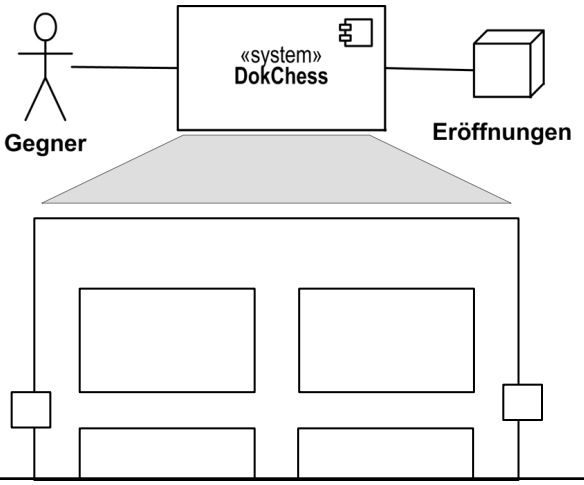
… sind die mit der aktuellen Lösung leicht möglich?

DokChess als Blackbox.



Das hängt vor allem hiervon ab:

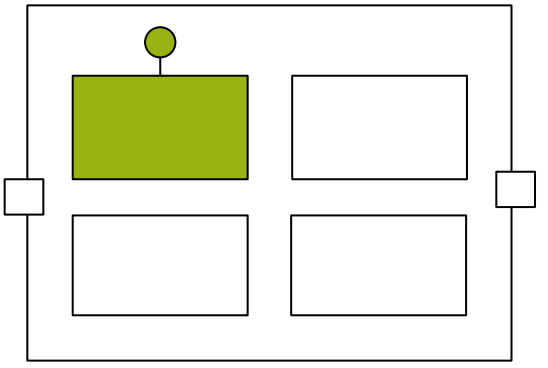
Wie ist die Lösung intern strukturiert?



Abhängigkeiten in Java.
oose. Innovative Informatik

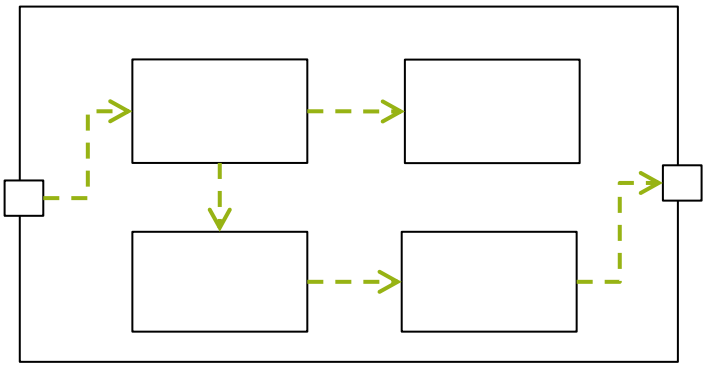
Das hängt vor allem hiervon ab:

Welche Funktionalität wird wie angeboten?



Das hängt vor allem hiervon ab:

Und wie hängen die Teile zusammen?



➔Abhängigkeiten als Schlüssel.

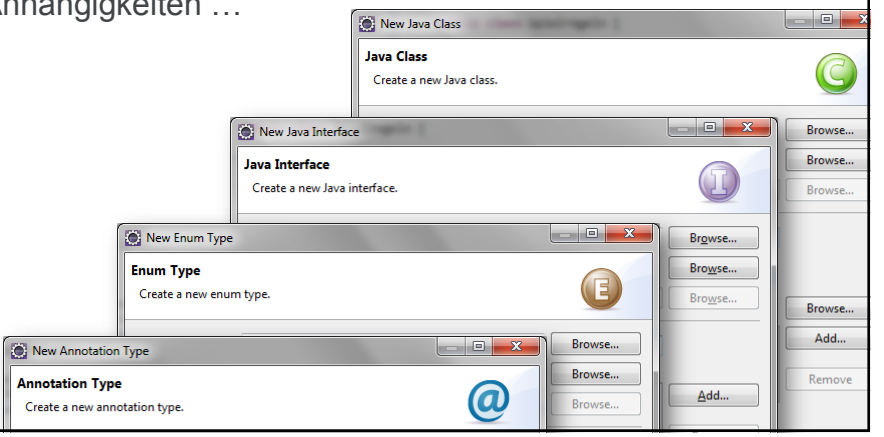
Abhängigkeiten in Java.
oose. Innovative Informatik

Agenda

Elemente des Java-Quelltextes

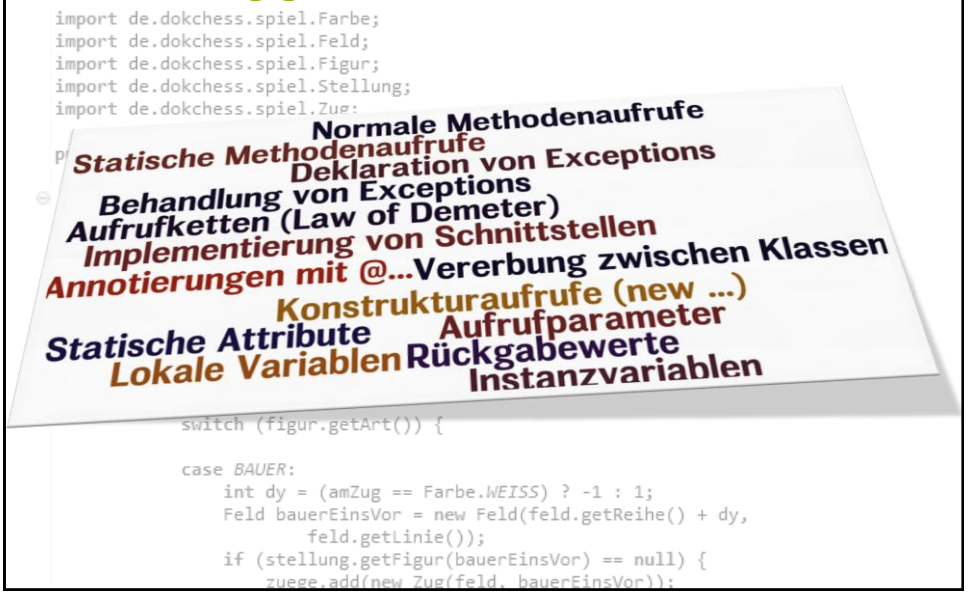
Vergleichsweise kleine Teile

- Klassen, Schnittstellen, Aufzählungen, Annotationen
- Zwischen diesen entstehen durch jeweilige Verwendung Anhängigkeiten …



Abhängigkeiten in Java.
oose. Innovative Informatik

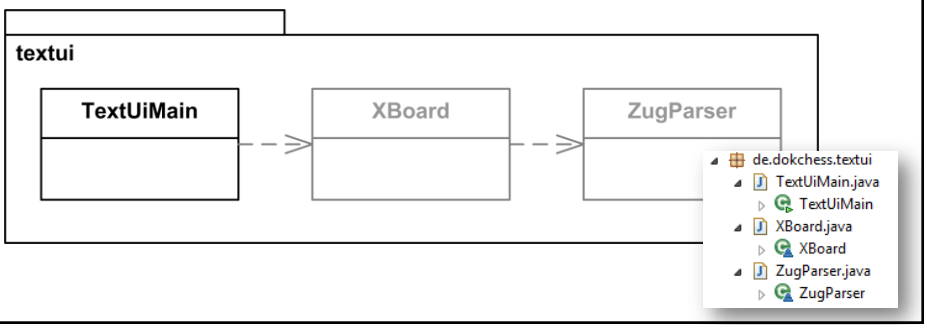
Wie Abhängigkeiten dazwischen entstehen …



Pakete in Java …

fassen Elemente zusammen

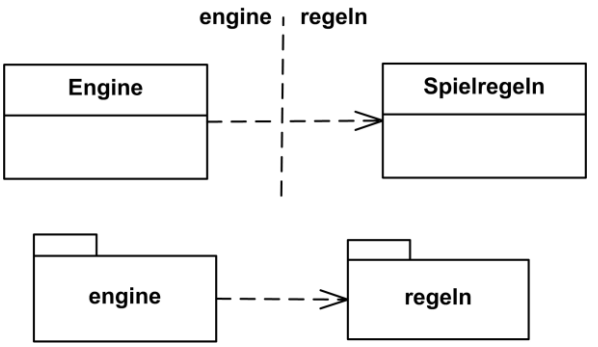
- Jede Klasse etc. gehört zu genau einem Paket (Angabe via Schlüsselwort *package*, sonst „*default*“-Paket)
- Schutz bzgl. Sichtbarkeit („package visible“)



Zwischen Paketen in Java …

… gibt es keine (direkten) Abhängigkeiten

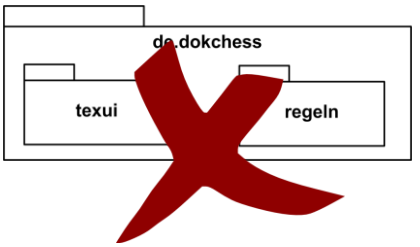
- Sie entstehen durch Verwendung von Elementen aus unterschiedlichen Paketen

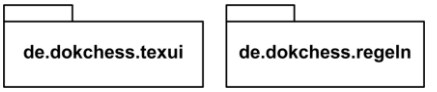


Pakete in Java

Pakete lassen sich in Java nicht schachteln

- Die Punktnotation (*org.apache*.) suggeriert das lediglich
- *de.dokchess.textui* und *de.dokchess.regeln* haben nichts miteinander zu tun, auch nichts mit *de.dokchess*
- „.“ Teil des Namens





Module

jar-Files (+ Varianten, z.B. aus Java EE)

- Um Meta-Informationen angereichertes ZIP-File
- Verwendung als Deployment-Unit / Artefakt
- Keine Grenze bzgl. Sichtbarkeiten, keine expliziten Abhängigkeiten

OSGi („Bundles“)

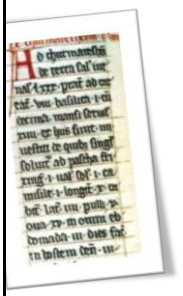
- Echtes Modulkonzept, ausgereift, kein Java-Standard
- Sichtbarkeiten, explizites Deklarieren von Abhängigkeiten

Project Jigsaw

- Modulkonzept für Java, Teil des OpenJDK-Projektes
- Sichtbarkeiten, explizites Deklarieren von Abhängigkeiten

Jigsaw frühestens in Java 9

Beispiel für Prinzipien: SOLID



S **Single Responsibility Principle (SRP)**
(Verantwortlichkeitsprinzip)

O **Open/Closed Principle (OCP)**
(Offen/Geschlossen Prinzip)

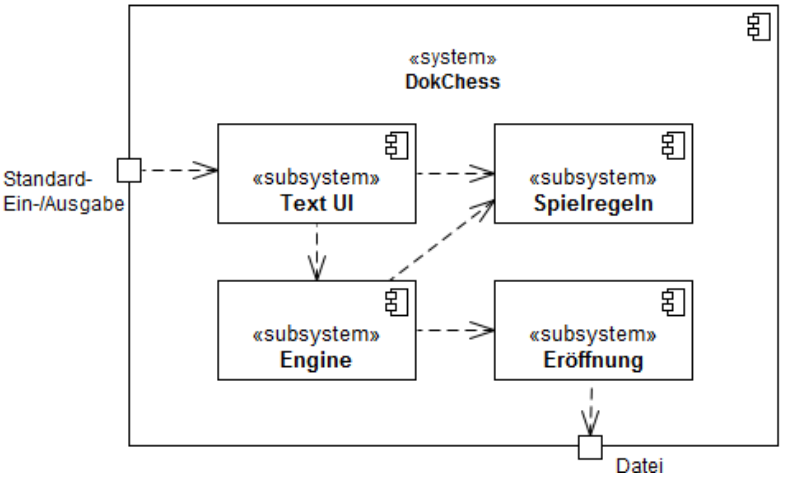
L **Liskov Substitution Principle**
(Liskov'sches Substitutionsprinzip)

I **Interface Segregation Principle (ISP)**
(Prinzip der Schnittstellentrennung)

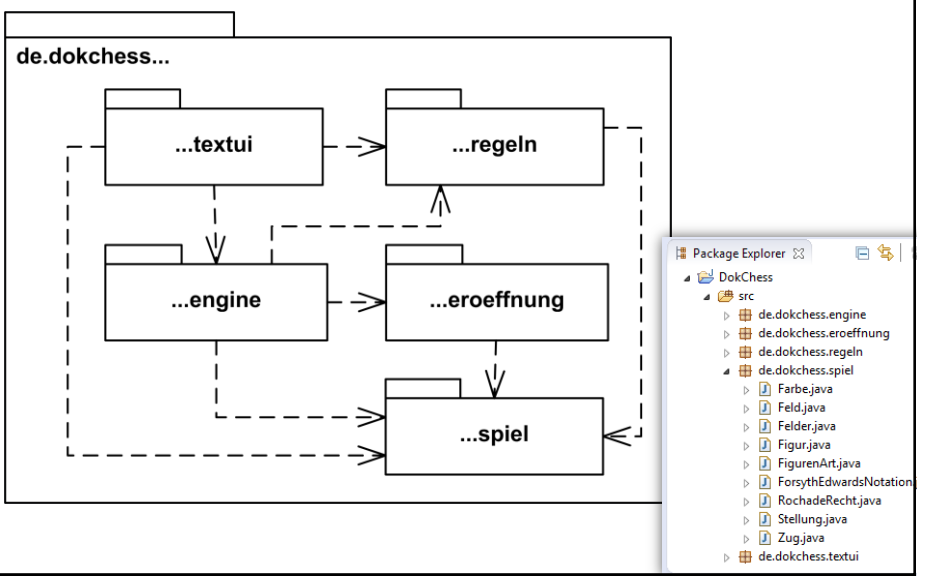
D **Dependency Inversion Principle (DIP)**
(Prinzip der Abhängigkeitsumkehrung)

Single Responsibility Principle – Beispiel

Zerlegung von DokChess nach Verantwortlichkeiten:



Paketstruktur von DokChess



Nehmen wir mal an …

… Ihr wollt, dass die Engine auch nach anderen als den Standardregeln spielen kann, z.B.

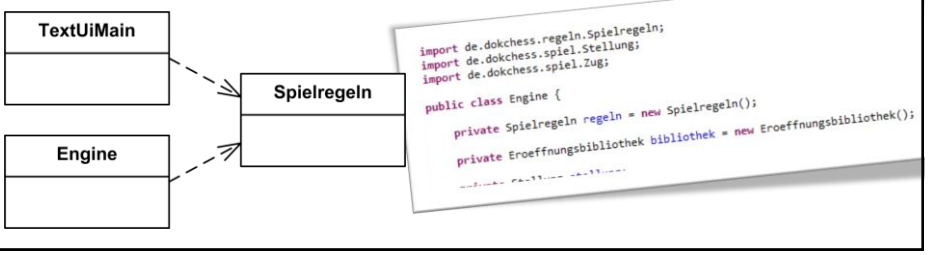
- Schach 960
- Räuber-Schach
- Atom-Schach
- …

Open Closed Principle – Beispiel

„ offen für Erweiterungen, geschlossen gegen Änderungen …“

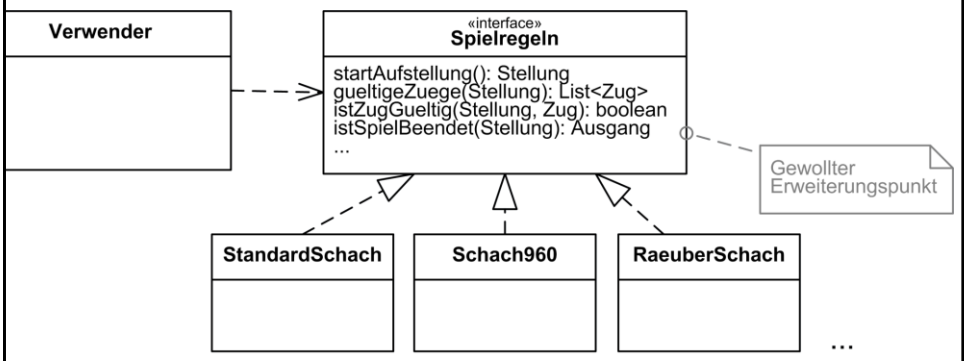
Anforderung: Leicht um andere Spielregeln ergänzen können

Ausgangssituation:



Einführen einer Abstraktion (Strategy Pattern)

- Verwender nutzen nur noch das Interface
- Konkrete Spielregeln als Implementierungen

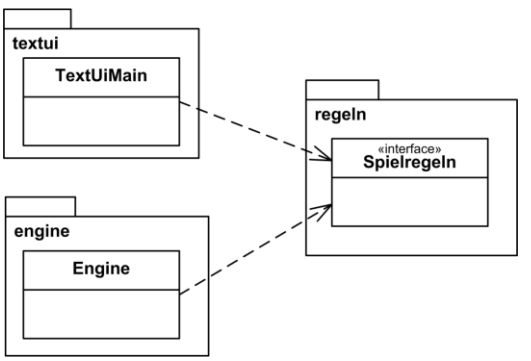


Fragestellung: Wie kommen Verwender an eine konkrete Implementierung? Optionen: Fabrik, Dependency Injection …

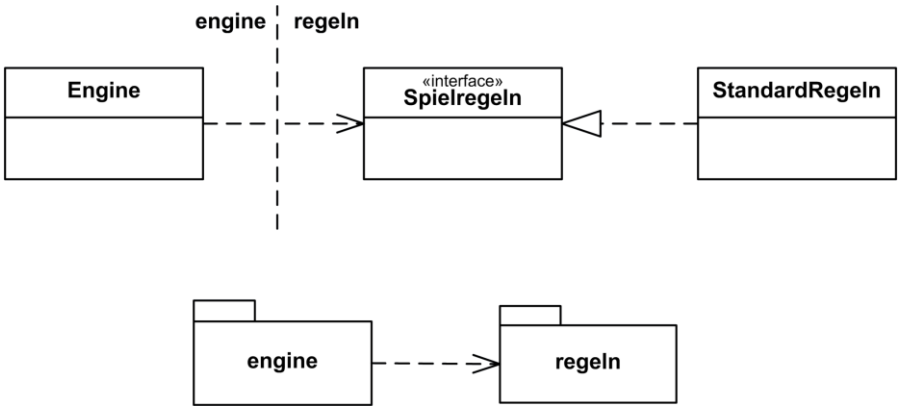
Fragestellungen

Entwurfsalternativen

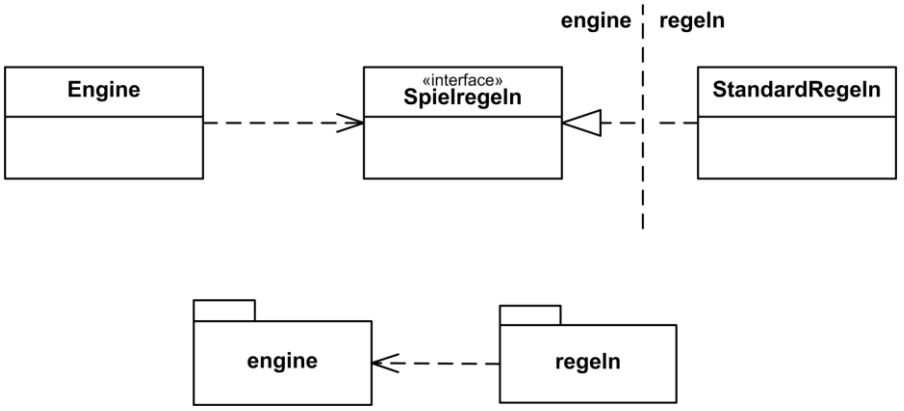
- Erhalten beide Verwender das selbe Interface? (vgl. Interface Segregation Principle)
- Wo platzieren wir das Interface? Bestehendes Paket? Neues Paket?



Dependency Inversion Principle



Dependency Inversion Principle



Nehmen wir mal an …

… Ihr wollt effizientere Datenstrukturen benutzen, um die Ermittlung des besten Zuges zu beschleunigen, oder in gleicher Zeit weiter vorausschauen zu können …

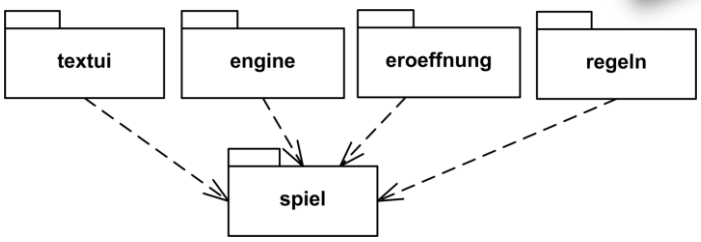


Entwurfsmuster – Beispiel

Anforderung

Datenstrukturen leicht durch effizientere austauschen können

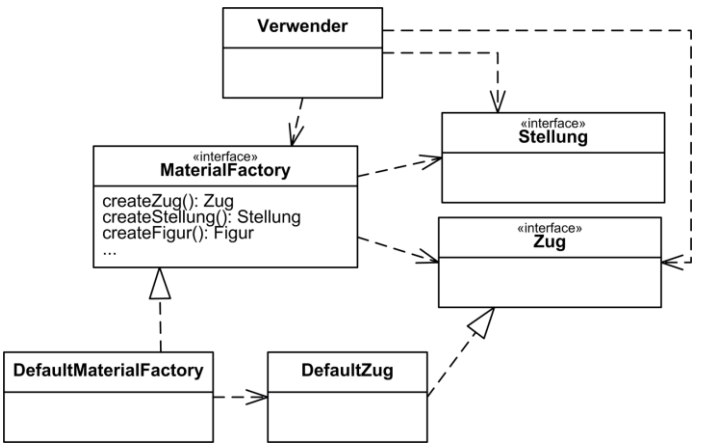
Ausgangssituation:



Alle Subsysteme sind vom Paket **`spiel`** mit den Klassen Figur, Zug, Feld, Stellung abhängig.

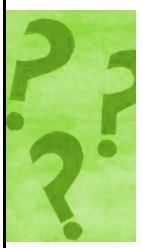
Abstract Factory (Ausschnitt)

- Einführen abstrakter Produkte und abstrakter Fabrik
- Verwender nutzen wieder nur Abstraktionen



Fragestellungen

Entwurfsalternativen



- In welche Pakete werden Abstraktionen und Standardimplementierungen der Produkte platziert?
  - Bestehende Pakete?
  - Neues Paket für Schnittstellen?
- Wie erhalten Verwender eine konkrete Fabrik?
  - Fabrikfabrik?
  - Dependency Injection?
  - …

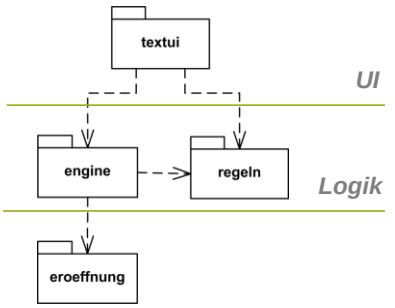
- Auch hier: Antworten haben Einfluss auf die Abhängigkeiten auf Paketebene

Beispiel Architekturmuster: Schichten

Anforderungen:

- Text UI leicht durch grafisches UI ersetzen können.
- Portierung auf Android

Ausgangssituation (als Schichten):



Sieht grundsätzlich gut aus …
Was ist mit dem spiel-Paket?
Andere Pakete?

Zusammenfassend: Aktivitäten

- Verantwortlichkeiten identifizieren (SRP)
  - Klassen, Pakete (, Module?)
- Verwendung festlegen
  - Abhängigkeiten
- Abstraktionen herausarbeiten (OCP)
  - Schnittstellen definieren (ISP)
  - Schnittstellen platzieren (DIP)
- Auffinden konkreter Implementierungen
- Schichten planen
- Richtung von Abhängigkeiten steuern (DIP)

Agenda

3

1 Warum Abhängigkeiten planen?
2 Abhängigkeiten in Java
3 **Werkzeuge**
4 Fazit + Ausblick

Planung

Tools unterstützen beim …

- Treffen der Entscheidungen (Entwurf)
- Festhalten der Entscheidungen (Struktur, Beziehungen)
- Visualisieren und Kommunizieren im Team

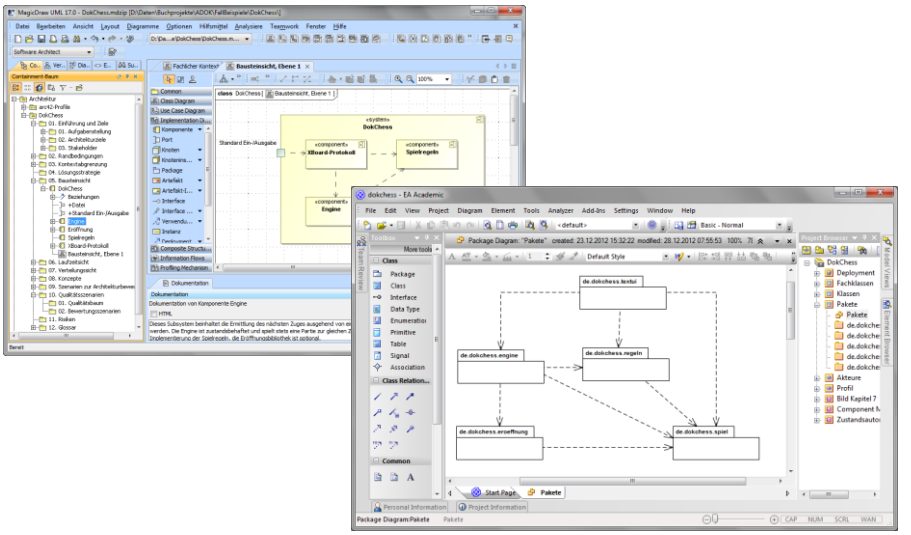
Analoge Tools

- Einsatz ohne Hürden, z.B. in Workshops
- Festhalten von Entscheidungen in Bildern

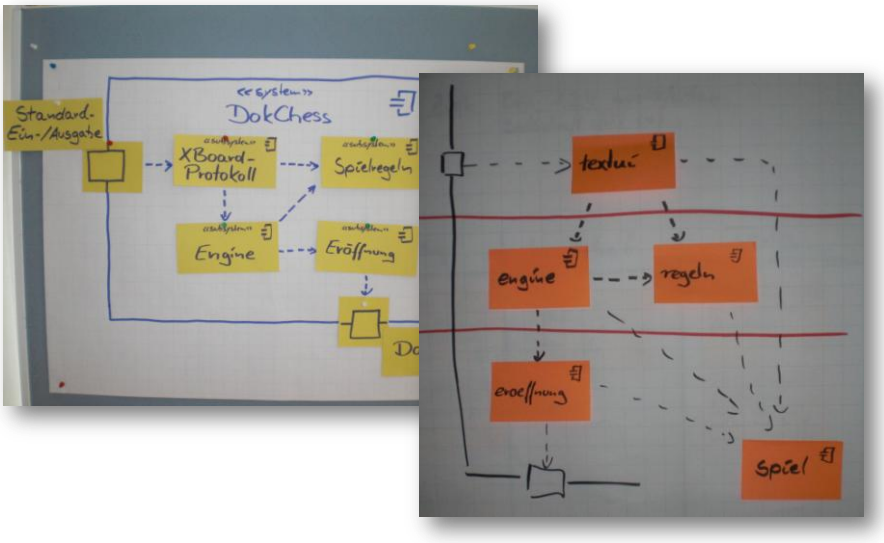
Digitale Tools

- Große Bandbreite von Zeichenprogramm bis UML-Tool
- Möglich: Modelle, Diagrammen als Sichten aufs Modell

Modellierungswerkzeuge



Analoge Werkzeuge



Umsetzung

Tools unterstützen beim …

- Überprüfen, ob die Struktur eingehalten wird
- Verhindern von unerwünschten Verwendungen
- Verhindern von zyklischen Abhängigkeiten

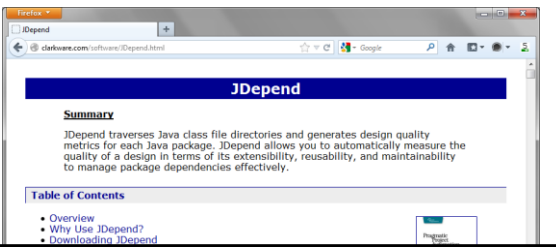
Konkrete Vertreter für Java (Auswahl, freie Tools)

- CheckStyle
  - ➔ http://checkstyle.sourceforge.net/
- JDepend
  - ➔ http://clarkware.com/software/JDepend.html

JDepend

„Out of the box“

- Kommandozeilenwerkzeug, um Berichte zu Metriken und Abhängigkeiten zu generieren (Text, XML)
- Integration in Ant, Maven, … zur automatischen Generierung
- API, um zum Beispiel JUnit-Tests zu schreiben, die Abhängigkeiten überwachen können



Beispiel: Berichte im Text-Format

jdepend_out.txt

```
-------------------------------------------------
- Package: de.dokchess.engine
-------------------------------------------------

Depends Upon:
    de.dokchess.eroeffnung
    de.dokchess.regeln
    de.dokchess.spiel
    java.lang
    java.util

Used By:
    de.dokchess.textui

-------------------------------------------------
- Package: de.dokchess.eroeffnung
-------------------------------------------------

Depends Upon:
    de.dokchess.spiel
    java.io
    java.lang
    java.util

Used By:
    de.dokchess.engine

-------------------------------------------------
```

jdepend_out.xml

```
<?xml version="1.0"?>
<JDepend>
    <Packages>
        <Package name="de.dokchess.engine">
            <DependsUpon>
                <Package>de.dokchess.eroeffnung</Package>
                <Package>de.dokchess.regeln</Package>
                <Package>de.dokchess.spiel</Package>
                <Package>java.lang</Package>
                <Package>java.util</Package>
            </DependsUpon>
            <UsedBy>
                <Package>de.dokchess.textui</Package>
            </UsedBy>
        </Package>

        <Package name="de.dokchess.eroeffnung">
            <DependsUpon>
                <Package>de.dokchess.spiel</Package>
                <Package>java.io</Package>
                <Package>java.lang</Package>
                <Package>java.util</Package>
            </DependsUpon>
            <UsedBy>
                <Package>de.dokchess.engine</Package>
```


Beispiel: Verwendung der JDepend-API

JDependApiBeispiel.java

```
    private JDepend jdepend;

    @Before
    public void setUp() throws IOException {
        jdepend = new JDepend();
        jdepend.addDirectory("bin");
        jdepend.analyze();
    }

    @Test
    public void zykelfrei() {
        Assert.assertFalse(jdepend.containsCycles());
    }

    @Test
    public void abhaengigkeiten() {

        DependencyConstraint constraint = new DependencyConstraint();

        JavaPackage regeln = constraint.addPackage("de.dokchess.regeln");
        JavaPackage textui = constraint.addPackage("de.dokchess.textui");
        JavaPackage spiel = constraint.addPackage("de.dokchess.spiel");

        regeln.dependsUpon(spiel);
        textui.dependsUpon(spiel);
        textui.dependsUpon(regeln);

        Assert.assertTrue(jdepend.dependencyMatch(constraint));
    }
```

Show-Case zur Verwendung der API

Definition erlaubter Abhängigkeiten in einem Textfile

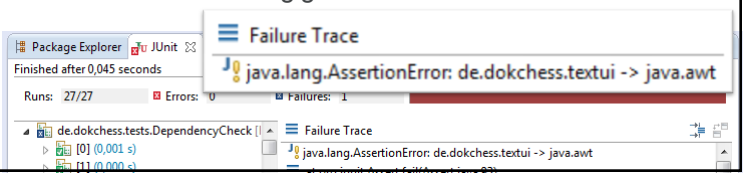
dependencies.txt

```
de.dokchess.*            -> java.lang, java.util

de.dokchess.textui       -> de.dokchess.spiel, de.dokchess.regeln, de.dokchess.engine, java.io
de.dokchess.engine       -> de.dokchess.spiel, de.dokchess.regeln, de.dokchess.eroeffnung
de.dokchess.regeln       -> de.dokchess.spiel
de.dokchess.eroeffnung   -> de.dokchess.spiel, java.io
```

JUnit-Test

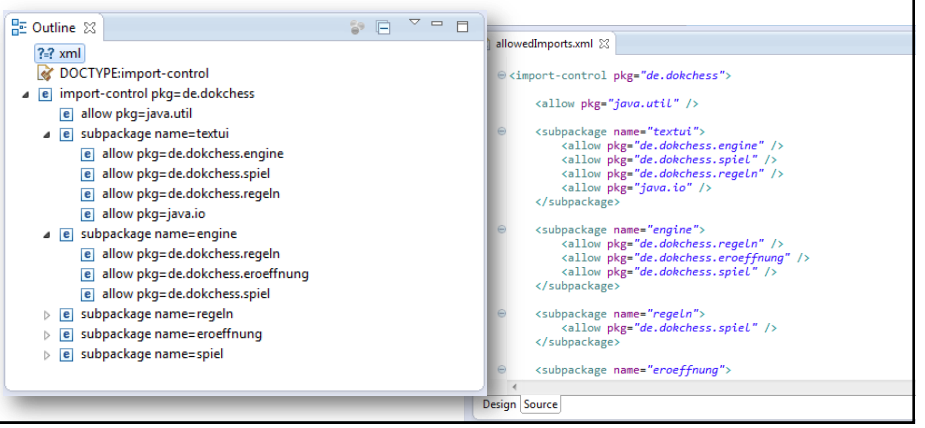
- analysiert mit JDepend
- hält gefundene Abhängigkeiten gegen erlaubte aus Datei
- schlägt Alarm bei unerlaubter Abhängigkeit …



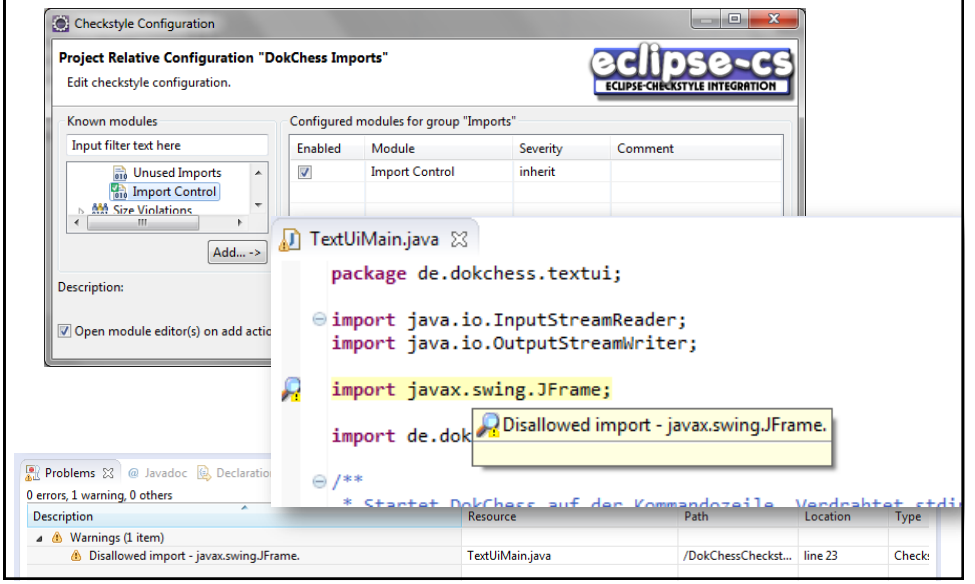
Beispiel CheckStyle

Checkstyle-Regel für Imports

- Checkstyle kann auf Verwendung von Imports prüfen
- zugehöriges Modul wird XML konfiguriert:



Integration in Eclipse



Abhängigkeiten in Java.

Analyse

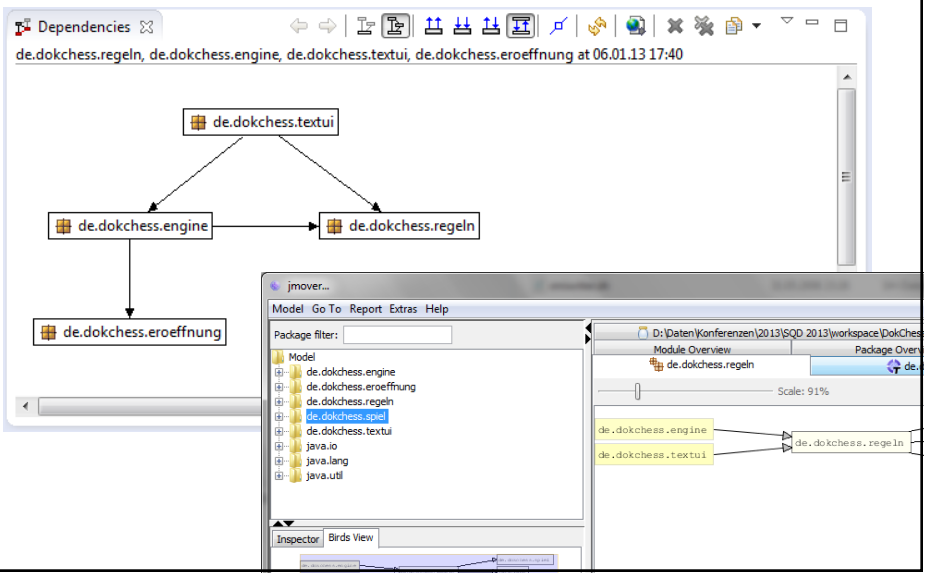
Tools unterstützen beim …

- Rekonstruieren von Strukturen
- Verifizieren von Annahmen bzgl. Abhängigkeiten
- Planen von Refactorings

Konkrete Vertreter für Java (Auswahl, freie Tools)

- CodePro Analytics (Google)
  - ➔ https://developers.google.com/java-dev-tools/
- JMove
  - ➔ http://www.jmove.org/

Beispiele CodePro Analytics, JMove



Überwachung

Tools unterstützen beim …

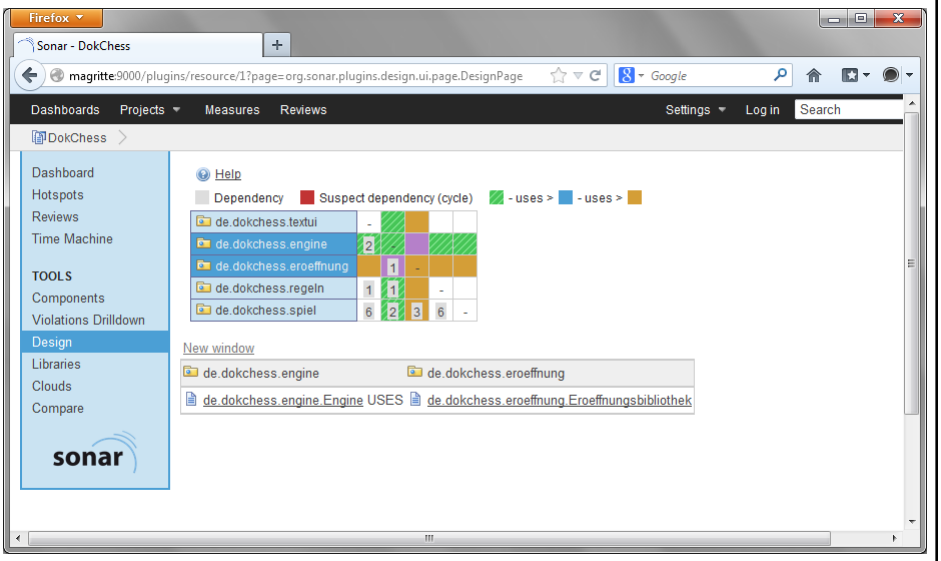
- Automatischen, kontinuierlichen Ausführen der Tests
- Kommunikation der Ergebnisse im Team

Konkrete Werkzeuge in Java (Auswahl, freie Tools)

- Build Tools (Maven, Gradle, …)
- Continous Integration Tools (Jenkins, …)
- Sonar

➔ http://www.sonarsource.org/

Beispiel Sonar



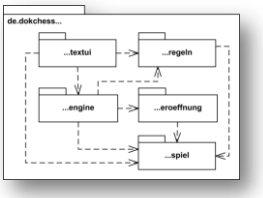
Agenda

4

1. Warum Abhängigkeiten planen?
2. Abhängigkeiten in Java
3. Werkzeuge
4. **Fazit + Ausblick**

Fazit

Prinzipien und Muster helfen Euch dabei, Euer System geeignet zu strukturieren und **Abhängigkeiten** zwischen Softwareteilen zu **planen**.



Tools unterstützen Euch dabei, die geplanten **Abhängigkeiten** im Java Quelltext **umzusetzen** und einzuhalten.

oose. Innovative Informatik

Beziehungskiste.

Abhängigkeiten im Softwareentwurf planen und überwachen

Stefan Zörner (sz@oose.de)

Voll unagil?

Berlin, den 05. April 2013
Berlin Expert Days

BED BERLIN EXPERT DAYS 2013
{join the talk}

Abhängigkeiten visualisieren.

	Modul 1	Modul 2	Modul 3	Modul 4	Modul 5	Modul 6
Modul 1	●	☒	☒			☒
Modul 2	☑	●	☒	☑	☒	
Modul 3	☑	☑	●		☒	
Modul 4		☒		●		
Modul 5		☑	☑		●	☒
Modul 6	☑				☑	●

Abhängigkeiten in Java.
oose. Innovative Informatik

Eine (!) Definition für Softwarearchitektur

"Softwarearchitektur ist die Menge der Entwurfsentscheidungen, die, wenn falsch getroffen, Dein Projekt zum Scheitern bringen kann."*

(Eoin Woods)

* Wörtlich: "Software architecture is the set of design decisions which, if made incorrectly, may cause your project to be cancelled."

Beispiel Schach-Engine

- Konsequente Anwendung von Entwurfsprinzipien führt zu einer flexiblen, erweiterbaren Lösung.

Wartbarkeit
Portabilität ⟺ Effizienz

- Gleichzeitig: Ausbalancieren mit Anforderungen bzgl. Effizienz erforderlich.
- Sonst ist die Spielstärke der Engine gefährdet, und damit ggf. der Gesamterfolg.

Entscheidend:

Ein gutes Verständnis Eurer Qualitätsanforderungen

Nur so könnt Ihr entscheiden, ob eine Abstraktion oder ein Muster

- vom Aufwand oder der Komplexität her **angemessen**
- überhaupt **zielführend**

ist.

Mehr zum Festhalten von Entscheidungen …

Softwarearchitekturen dokumentieren und kommunzieren.
Entwürfe, Entscheidungen und Lösungen nachvollziehbar und wirkungsvoll festhalten

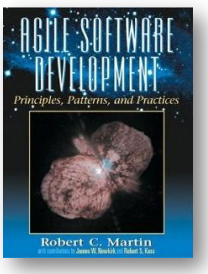
von Stefan Zörner
Verlag: Hanser, Mai 2012
Sprache: Deutsch (ca. 280 Seiten)
ISBN-13: 978-3446429246

➔ **http://www.swadok.de**

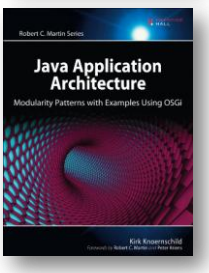
- Erfahren Sie, wie die Dokumentation der Architektur von der lästigen Pflicht zu einem integralen Kommunikations- und Arbeitsmittel wird.
- Lernen Sie architekturrelevante Einflussfaktoren und zentrale Entscheidungen festzuhalten.
- Erleben Sie am Beispiel einer Schach-Engine, wie eine nachvollziehbare Architektur entsteht.

Abhängigkeiten in Java.
oose. Innovative Informatik

Weitere Literatur zum Thema



Agile Software Development. Principles, Patterns, and Practices
von Robert C. Martin
Gebundene Ausgabe, 529 Seiten
Verlag Prentice Hall Computer (2002)
ISBN-13: 978-0135974445



Java Application Architecture. Modularity Patterns with Examples Using OSGi
von Kirk Knoernschild
Taschenbuch, 384 Seiten
Verlag: Addison Wesley (2012)
ISBN-13: 978-0321247131

oose. Innovative Informatik

Veranstaltungshinweis

Oliver Gierke :

Huch, wo ist meine Architektur hin?

bedcon 2013 | 05.04.2013 | 16:00 - 17:00 Uhr

BED BERLIN EXPERT DAYS 2013
{join the talk}

Abstract: Wenn Applikationen über eine bestimme Größe oder einen bestimmten Zeitraum hinaus wachsen wird Modularität ein Kernaspekt für Wartbarkeit.
Designentscheidungen die getroffen wurden sind kaum noch im Code wiederzufinden, Abhängigkeiten zwischen einzelnen Modulen der Applikation wachsen oft wild. Der Vortrag Patterns und Best Practices rund um generelle Code-Organisation und Package-Strukturen vor um eine solide Grundlage für langlebige Java-Applikationen zu legen, sowie eine Möglichkeit mit Spring lose gekoppelte Komponenten und dedizierte Erweiterungspunkte in Applikationen zu definieren und zu verwenden.

oose. Innovative Informatik

Vielen Dank!

Ich freue mich auf Eure Fragen!

Stefan.Zoerner@oose.de