Flexible Architekturen

mit der Eclipse Extension-Point-Technologie

Martin Lippert, akquinet agile GmbH
Gernot Neppert, Otto Group

Agenda

- Ein kleiner Einstieg
  - Was ist ein Extension-Point? Was ist eine Extension?
  - Architektonischer Blick auf die Technologie
- Reale Beispiele aus Projekten
  - Von einfachen Extension-Points
  - Bis zur „Pluggable Persistence“
- Zusammenfassung

Wir über uns

Gernot Neppert
Otto Group
gernot@neppert.com

Schwerpunkte:
- Plattformunabhängige Libraries
- Multithreading
- OR-Mapping

Martin Lippert
akquinet agile GmbH
martin.lippert@it-agile.de

Schwerpunkte:
- Eclipse-Technologie
- Architekturen
- Agile Software-Entwicklung

Flexible Architekturen

- Die Architektur eines Software-Systems muss flexibel sein

- Denn:
  - Anforderungen ändern sich
  - Wir lernen im Laufe der Zeit hinzu
  - Systeme leben eine lange Zeit

Flexible Systeme

- Systeme müssen flexibel zu entwickeln und zu erweitern sein

- Denn:
  - Wir können nicht alle Features eines Systems vorhersagen
  - Viele Personen arbeiten an einem System
  - Systeme leben eine lange Zeit und müssen immer wieder angepasst und erweitert werden

Welche Möglichkeiten haben wir?

- Na klar, das kennen wir:
  - OO (und vielleicht auch AOP ;-)
  - Saubere Modularisierung in Klassen
  - Diverse Design-Prinzipien (von SOC bis zu DRY)

- Und das machen wir auch - versteht sich:
  - Lose Kopplung
  - Dependency Injection (beispielsweise mit Spring oder ähnlichen Frameworks)

Noch mehr Möglichkeiten? – Klar!

- Komponenten
  - Modularisierung in größere Einheiten
  - Beispielsweise mit OSGi, wie z.B. die Eclipse-Plattform
- Layering
  - z. B. horizontal: Persistenz, Business-Logik, Präsentation, …
  - z. B. vertikal: Geschäftsbereiche, …
  - …

Extension-Point-Technologie

- Definiere explizit solche Stellen im System, an denen das System erweiterbar sein soll
  - „Eine Steckdose zur Verfügung stellen“
  - Einen Erweiterungspunkt (**Extension-Point**)
- Erlaube anderen Komponenten, Erweiterungen zu diesen definierten Erweiterungs-Punkten beizusteuern
  - „Einen Stecker liefern“
  - Eine Erweiterung (**Extension**)

Eine Sichtweise auf Architektur

- Es geht bei Extension-Points also nicht hauptsächlich um…
  - Abhängigkeits-Management
  - Kapselung
  - Schichten-Bildung
- Stattdessen:
  - Wie spielen Komponenten zusammen?
  - Eine Art „minimales Komponentenmodell“

Den Vertrag definieren

- Der Anbieter eines Extension-Points definiert den Vertrag
  - Wie muss eine Extension aussehen?
  - Wann und wie wird die Extension verwendet?

- Das passiert häufig mit:
  - Einem XML-Schema (wie sieht der deklarative Anteil einer Extension aus)
  - Einem API (Java-Interface), welches die Extension implementieren soll

Beispiel

Der Extension-Point, für den diese Extension definiert ist

Die Klasse, die das Interface IViewPart implementiert

Weitere Meta-Daten für die Extension

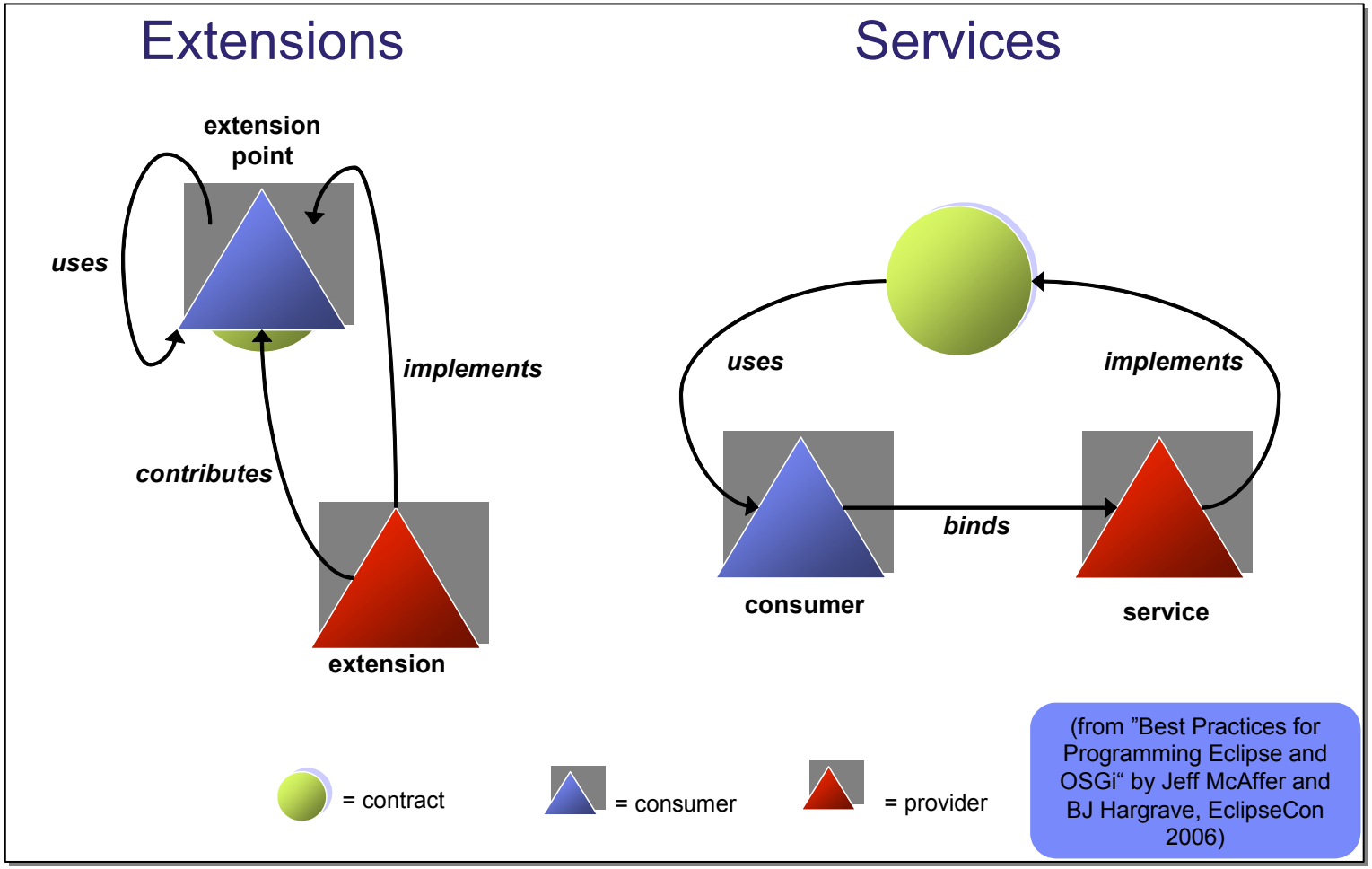
```
<extension
      point="org.eclipse.ui.views">
   <view
         name="Message"
         allowMultiple="true"
         icon="icons/sample2.gif"
         class="org.eclipse.example.rcpmail.View"
         id="org.eclipse.example.rcpmail.view">
   </view>
</extension>
```

Ausschnitt aus der plugin.xml

Auswirkungen

- Abhängigkeiten kehren sich um
  - Der Anbieter des Extension-Points kennt die Extensions nicht statisch, obwohl er sie zur Laufzeit aufruft (typische Framework-Konstruktion)
  - Die Extension kennt den Aufrufer, also den Extension-Point
- Enge Kopplung zwischen Extension und Extension-Point
  - Aufgrund des Vertrags
  - Und weil eine Extension zu genau einem Extension-Point hinzugefügt wird

Extension-Points vs. Services



Extension-Registry

- Die „Extension-Registry“ implementiert den Extension-Point-Mechanismus
  - Steht als zusätzliches OSGi-Bundle zur Verfügung
    `org.eclipse.equinox.registry`

- Anmerkung:
  - Die Extension-Registry kann auch ohne OSGi verwendet werden
  - Sehr interessant für große Anwendungen, die von diesem Mechanismus profitieren wollen, aber kein OSGi einsetzen (können)

Logging als Beispiel

Prämissen:

- Es gibt ein Log-Interface, das von der Applikation benutzt wird, um Log-Records wegzuschreiben.
- Der dahinterliegende Log-Service spricht „Appender“ über ein Interface an und verteilt an sie die Log-Records.
- Es gibt Implementierungen des Appender für Console, für Files, für TCP/IP usw.

Die Frage: Wie meldet man die Appender bei der Applikation an?

Logging als Beispiel

1. Idee:

Explizite Instanzierung.

```
public static void main(String[] argv]) {
    LogService.registerAppender(new ConsoleAppender());
    LogService.registerAppender(new RollingFileAppender(„application.log“));
    LogService.registerAppender(new TcpAppender(4711));
...
}
```

Vorteile:

- Eigentlich keine...

Nachteile:

- Die Konfiguration erfolgt hart verdrahtet ***in der Anwendung***
- Es wird im Anwendungscode festgelegt, welche Implementationsklassen es für Appender geben soll.

Logging als Beispiel

2. Idee:

Config-file (wie z.b. bei Log4J)

```
<log4j:configuration>
<appender name=„console" class="org.apache.log4j.ConsoleAppender„/>
<appender name=„file" class="org.apache.log4j.RollingFileAppender„/>
<appender name=„tcpip" class="org.apache.log4j.net.SocketAppender„/>
</log4j:configuration>
```

Vorteile:

- Die Applikation braucht die Appender nicht mehr zu verwalten, sondern nur der Code im Log-Service.
- Die Implementationsklassen sind nicht mehr hartverdrahtet.

Logging als Beispiel

Aber:

- Die Dependencies haben sich in Wirklichkeit überhaupt nicht geändert!
- Es gibt immer noch eine zentrale Stelle, die alle Appenderklassen „kennen“ muss, nämlich das Config-File.
- Auch die explizite Aufnahme der Appenderklassen in den Applikations-Classpath bleibt bestehen.

Das ist ja nur „halbgar“!

Logging als Beispiel

Deshalb die 3. Idee:

Es gibt einen Extension-Point „Appender“, der vorgibt, dass sich Klassen, die das Appender-Interface implementieren, hier anmelden können.

So eine Anmeldung sieht dann z.B. so aus:

```
<extension
        point="Jax2007.LoggingAppender">
        <appender class="com.neppert.RollingFileAppender"/>
  </extension>
```

Logging als Beispiel

Vorteile der Lösung mit Extension-Point:

- Der zentrale Code im Log-Service braucht nicht in einem bestimmten Pfad ein bestimmtes Config-File zu parsen. Er fragt einfach über das API der ExtensionRegistry: „Hat sich jemand als Appender angemeldet?“
- Es lassen sich ohne weitere Änderungen weitere Appender „hinzustecken“, einfach dadurch, dass sie von der ExtensionRegistry gefunden werden können.
- Es besteht keine Abhängigkeit mehr zur Existenz bestimmter Implementationsklassen.

Weitere Beispiele für Extension-Points

- Im Eclipse-SDK
  - Views
  - Editors
  - Actions
  - Perspectives
  - Refactorings
  - Quick-Fixes

  - viele viele mehr…

Was haben diese Beispiele gemein?

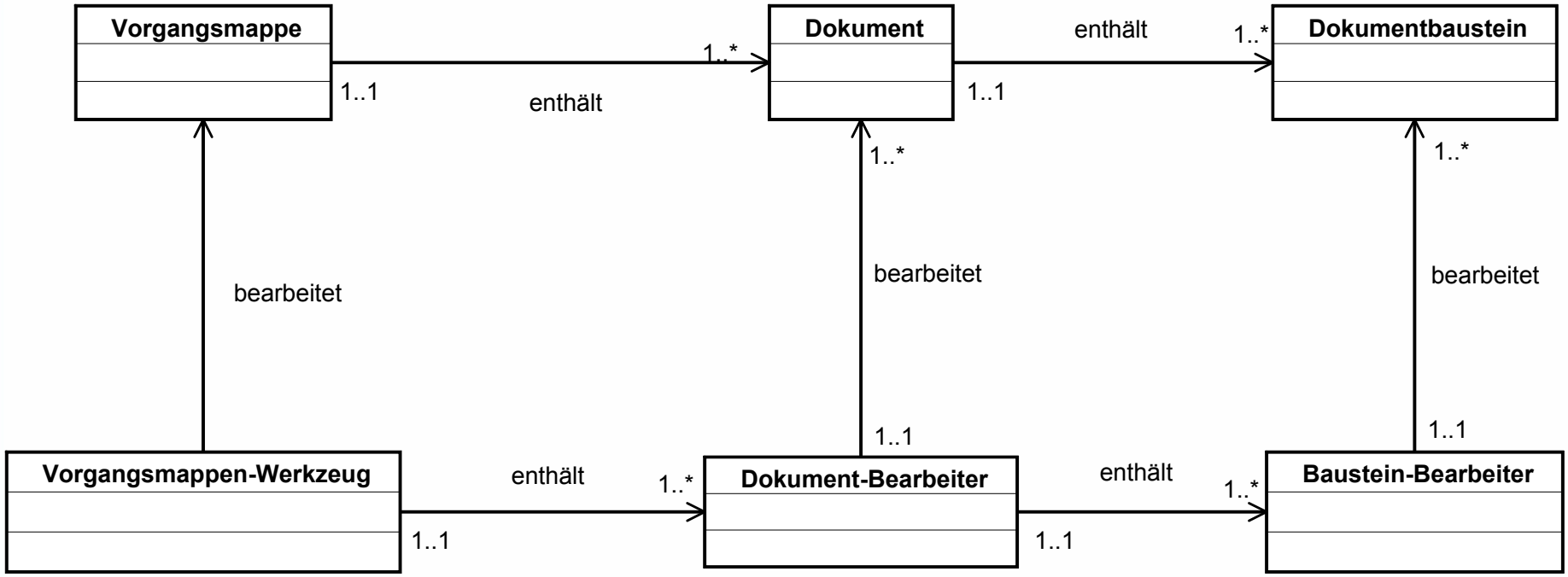
- Die Schnittstelle wird von einem oder mehreren Extension-Providern implementiert
  - Der Extension-Point-Anbieter muss mit 1 bis n dieser Extensions umgehen können
- Die Extensions melden sich selber bei der Anwendung an
  - einfach indem sie wiederum ihre Extension als XML-Konfiguration zur Verfügung stellen
- Aus Sicht des Anwendungscodes sind die angemeldeten Extensions „einfach da“, sie müssen nicht explizit gesucht werden

Mehr als UI: Weitere Beispiele

- Extension-Points spielen nicht nur für UIs eine Rolle!!!
- Sie können auf allen Ebenen definiert und nützlich eingesetzt werden.
- Beispiele aus dem Eclipse-SDK:
  - Applications
  - Builders
  - Filesystems
  - …

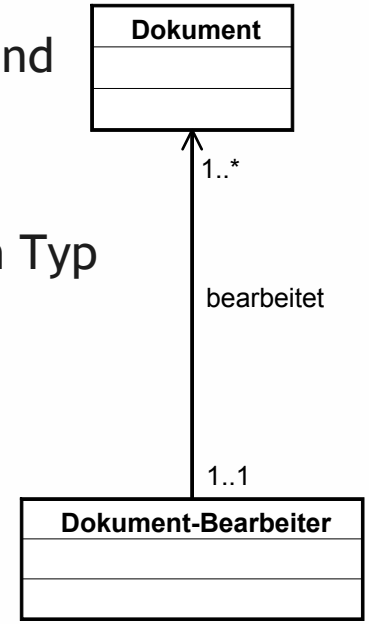
Beispiel: UI

Strukturübersicht Vorgangsmappen – Dokumente



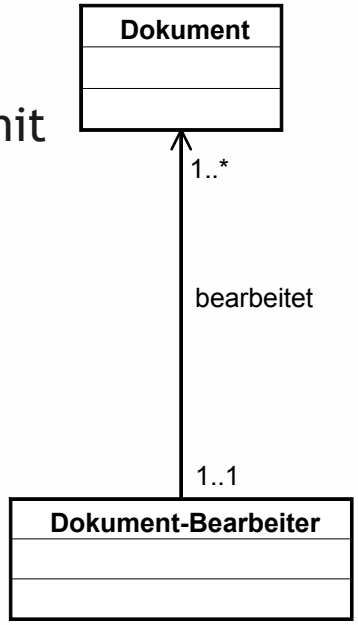
Beispiel: UI

- Dokument-Bearbeiter ist ein Java Interface und ein Extension Point
- Java-Klassen, die Dokument-Bearbeiter implementieren, enthalten den vollständigen Code, um Dokumente eines Typs darstellen und editieren zu können
- Jede Extension, die sich hier anmeldet, deklariert, für welchen Typ von Dokument sie zuständig ist
- Die Zuordnung von Dokument-Typen und ihren zuständigen Bearbeitern erfolgt ausschließlich über die Konfiguration (plugin.xml)



- Man kann flexibel neue Dokumente und ihre entsprechenden Bearbeiter dem System hinzufügen, ohne das Vorgangsmappen-Werkzeug selbst zu verändern

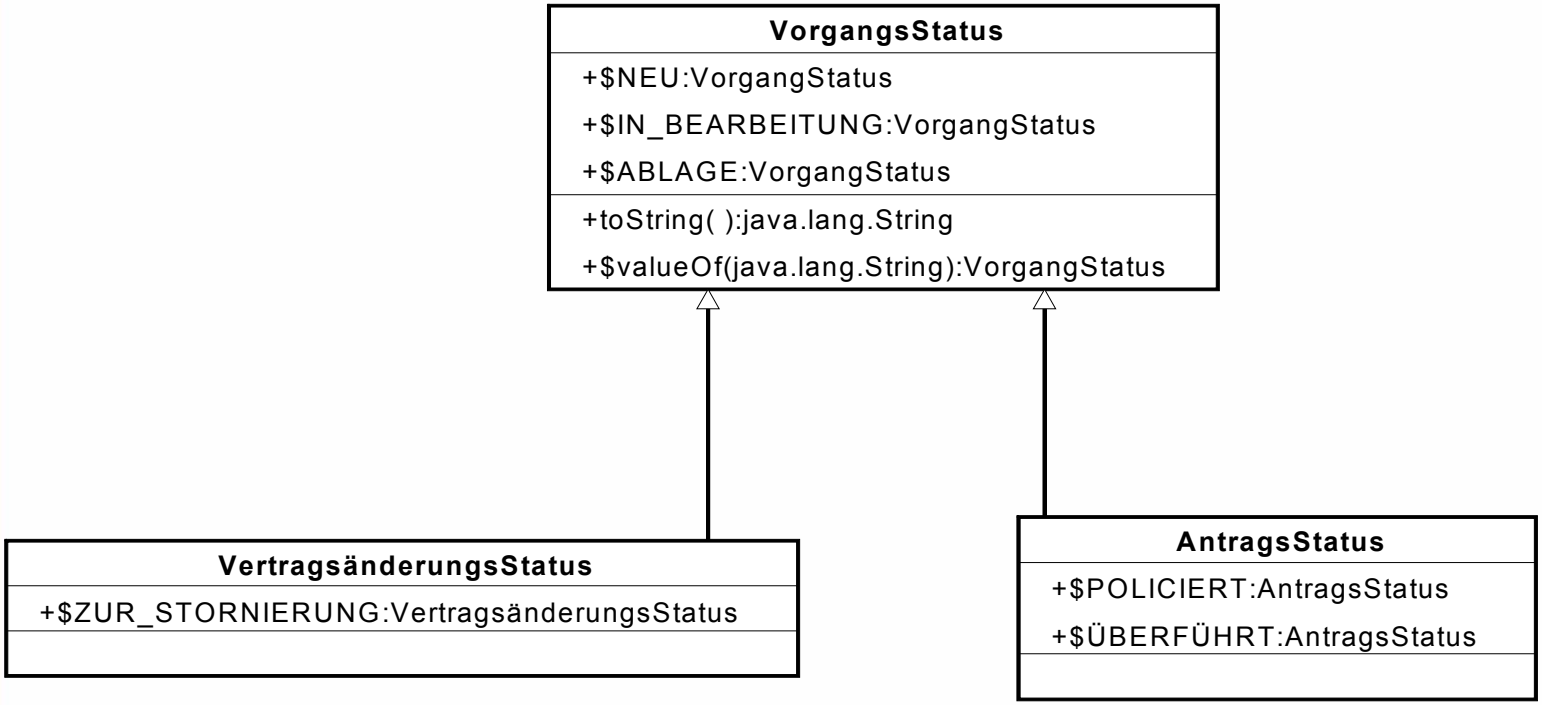
- Je nachdem, welche Plugins man ausliefert, kann man Werkzeuge mit verschiedenen „Sichten“ auf dieselben Vorgangsmappen ausliefern



Beispiel: Initialisierung erweiterbarer Wertetypen

- Erweiterbare Wertetypen sind keine Java-Enums!

  Prinzip: Sobald die abgeleiteten Klassen geladen und initialisiert werden, registrieren sich ihre statischen Member **automatisch** als Instanzen ihrer Oberklasse.



Beispiel: Initialisierung erweiterbarer Wertetypen

Speicherung in der Datenbank über String-Mapping.

Der Hinweg über `toString`:

```
VorgangStatus status = vorgangsmappe.getStatus();

VorgangsmappenBean bean = new VorgangsmappenBean();

bean.setStatus(status.toString());

persistenceManager.makePersistent(bean);
```

Beispiel: Initialisierung erweiterbarer Wertetypen

Speicherung in der Datenbank über String-Mapping.

Der Rückweg über `valueOf`:

```
VorgangsmappenBean bean =

  persistenceManager.getObjectById(vorgangsId);

Vorgangsmappe mappe = new Vorgangsmappe();

mappe.setStatus(VorgangsStatus.valueOf(bean.getStatus()));
```

Beispiel: Initialisierung erweiterbarer Wertetypen

Das Problem: Woher soll **VorgangsStatus.valueOf** etwas von den Konstanten in den abgeleiteten Klassen wissen?

Zur Erinnerung: sobald die abgeleiteten Klassen geladen und initialisiert werden, registrieren sich ihre statischen Member automatisch als Instanzen der Oberklasse VorgangsStatus.

Wir benötigen also eine verlässliche Initialisierung der Klassen erweiterbarer Wertetypen zum Programmstart!

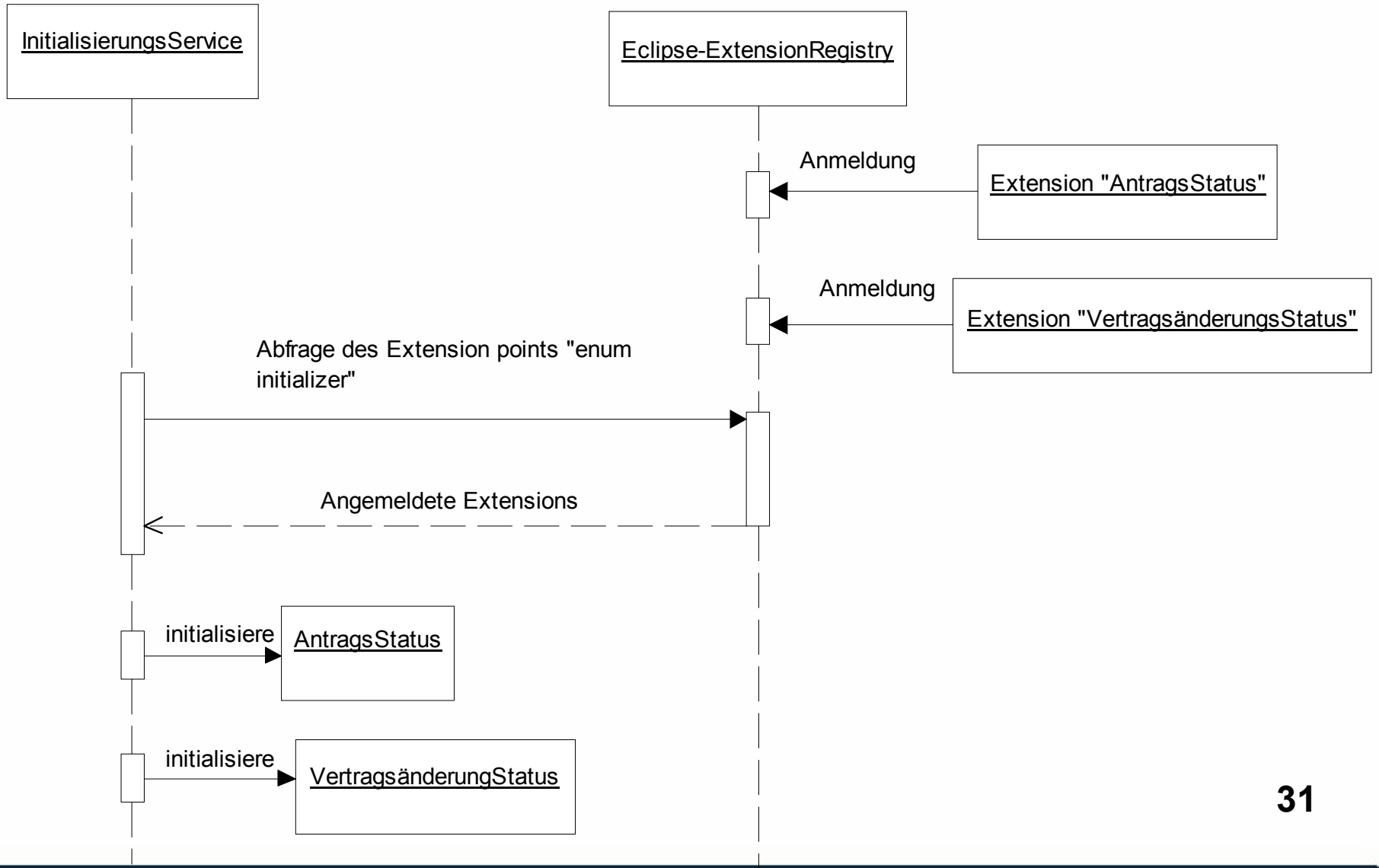
Beispiel: Initialisierung erweiterbarer Wertetypen

Die Lösung: Ein Initialisierungs-Service und ein Extension point „enum initializer“.

Die Klassen melden sich über diesen Extension Point selber zur Initialisierung an.

Die Applikation muss nur noch beim Programmstart den Service aufrufen, der dann alle angemeldeten Klassen initialisiert.

Beispiel: Initialisierung erweiterbarer Wertetypen



Highlevel-Beispiel: Konfiguration von Versicherungstarifen

- **Generelle Frage:**
  - Was flexibilisiert man per Konfiguration (aus der DB) und was per Extension-Point inkl. Code?

- **Unser Weg: Kein komplettes Meta-Modell**
  - Neue Tarife, Annahme-Grenzen und ähnliches wird per Datenbank konfiguriert
  - Neue Produkte können per Extension hinzugefügt werden

Beispiel: Konfiguration von Versicherungstarifen

- Neue Produkte per Extensions:
  - Produkt-Definition als Business Object (Dokument)
  - Produkt-Persistenz als Service (Dokument-Verwalter)
  - Produkt-spezifisches UI als UI-Komponenten (Dokument-Bearbeiter, etc.)
  - Neue Prüfungen als Services
  - Passende Marker-Fixes
  - Spezielle Actions

Beispiel: Pluggable Persistence

- Persistenz für Vorgangsmappen und Dokumente
  - Vorgangs-Service realisiert die Persistenz für die gesamte Mappe (inkl. Transaktions-Handling)
  - Persistierer für einzelne Dokumente können hinzu-gepluggt werden (sowohl auf dem Client, als auch dem Server)
- Erlaubt sehr flexible Weiterentwicklung von Vorgangsmappen
  - Neuartige Dokument-Typen können schnell und einfach hinzugefügt werden
  - Persistenz-Technologie kann schrittweise (pro Dokument-Typ) verändert oder migriert werden

Typische Arten von Extension-Points

- „Optional & Multiple“:
  - „Der klassische Extension-Point“
  - Es muss keine Erweiterung gegen, das System funktioniert auch ohne
  - Es kann aber beliebig viele Erweiterungen geben, die alle berücksichtigt und verwendet werden
- „Single & Required“:
  - Extension-Point-Anbieter erwartet *genau eine* Extension
  - Ohne diese eine Erweiterung kann er selbst nicht arbeiten
  - Ist eher ein klassischer Fall für Dependency-Injection

Skalierbarkeit mit Extension-Points

- Der Extension-Point-Mechanismus ist entworfen für eine potentiell große Anzahl von Extensions
  - Deklarativer Anteil ist schnell und gecached
  - Und hängt am Bundle-Resolved-State
    - Extension-Point-Anbieter kann diese Informationen nutzen
    - Kein Classloading oder Bundle-Activate notwendig
  - Bundle-Activate findet erst statt, wenn Extension genutzt wird

- Dadurch können auch Tausende von Extensions hineingesteckt werden, ohne dass Performance-Probleme auftreten
  - Wenn Extension-Point-Anbieter entsprechend realisiert ist

Extension-Points vs. Dependency Injection

- Ist der Extension-Point-Mechanismus eine Art von Dependency Injection?

- Ja, weil…
  - … Abhängigkeiten injiziert werden
  - … der Anbieter konkrete Implementationen nicht kennt
- Nein, weil…
  - … der Mechanismus architekturell nicht für generelle Abhängigkeiten zwischen Komponenten entworfen wurde
  - ... der Skalierbarkeits-Aspekt eine große Rolle spielt

Der Prozess

- Extension-Points nicht auf Vorrat definieren:
  - „Always have a client“
- Können gut im Laufe der Zeit heraus-faktorisiert werden
  - Passt also gut zu unserer Ausgangs-Vorstellung von inkrementeller und evolutionärer Architektur-Entwicklung

Zusammenfassung

- Extension-Points ermöglichen es in großen Projekten,
  - Aufgaben gut zu verteilen (parallele Entwicklung von Extensions beispielsweise)
  - dass das System überschaubar bleibt (weniger Code im „Core“)
  - dass die Software erweiterbar wird
  - dass die Software flexibel zusammengesetzt werden kann
- Extension-Points haben sich in großen Projekten bereits bewährt
  - Eclipse-Ecosystem
  - IBM Rational, IBM Lotus, IBM Jazz

Vielen Dank!

- … für die Aufmerksamkeit!

- Fragen und Feedback jederzeit gerne!!!
- Besuchen Sie uns auf dem it-agile-Stand

- Martin Lippert, martin.lippert@akquinet.de
- Gernot Neppert, gernot@neppert.com