Universal Insurance Workplace: Eclipse-Technologie in Action

Ivo Eitner, Aspecta Lebensversicherung AG
Markus Herzog, Aspecta Lebensversicherung AG
Gernot Neppert, Aspecta Lebensversicherung AG
Martin Lippert, akquinet agile GmbH

Vortrag im Arbeitskreis Objekttechnologie
2. Oktober 2006

ASPECTA

Agenda

Überblick über den Vortrag

- Die Ausgangslage
- Das Architektur des GPO-Systems
- Plattform-basierte Entwicklung
  - flexible Erweiterbarkeit auf Basis von Eclipse
- Lessons Learned
  - Was würden wir wieder so machen
  - Was würden wir anders machen
- Fazit

Die Ausgangslage

Die wichtigsten Verwaltungsprozesse waren zu teuer und zu langsam.

Ursachen:

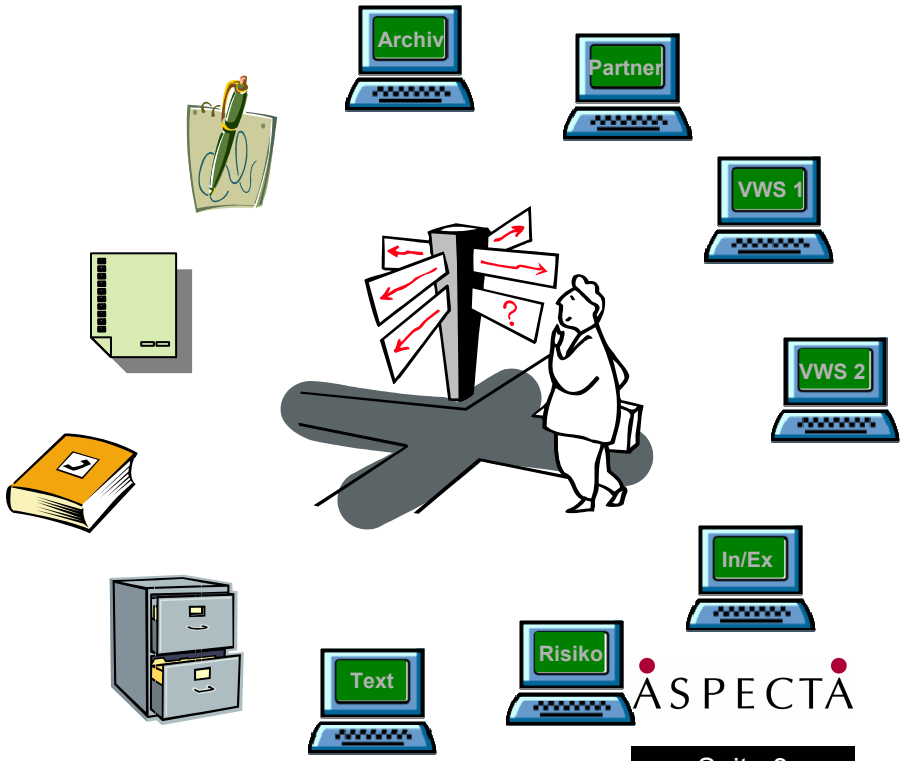
- Ineffektive Prozesse
- Schlechte IT-Unterstützung der Anwender durch
  - Gewachsene Systemlandschaft
  - Verschiedenste Technologien
  - Medienbrüche

Folge:

- Der Prozess orientiert sich an den Systemen statt umgekehrt.

Resultat:

+ Große Flexibilität
– Hohe Personalkosten
– Hohe Laufzeiten
– Geringe Transparenz



ASPECTA

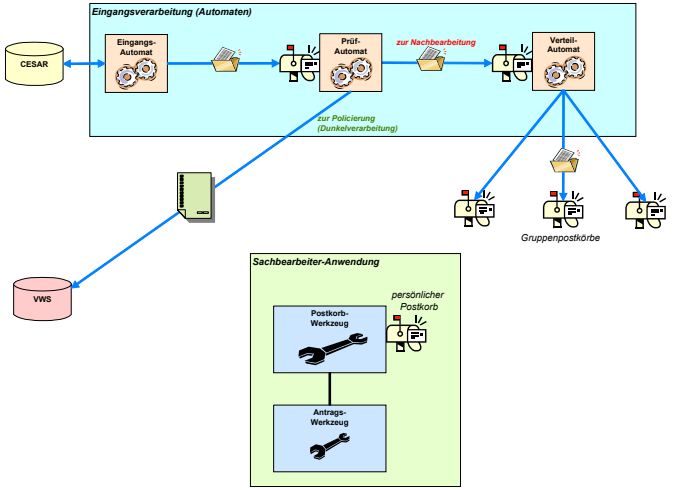
Geschäftsprozess-Optimierung

Projekt zur Geschäftsprozessoptimierung (**GPO**)

Qualität verbessern – Kundenzufriedenheit erhöhen – Kosten senken

Analyse :

- Bestandsaufnahme
  - Prozesse, Stückzahlen, Durchlaufzeiten
  - Identifikation der Optimierungskandidaten
- Definition der fachlichen Sollprozesse



Umsetzung :

- Organisatorische Veränderungen
- Bereitstellung einer technischen Prozessunterstützung :
  - Der Prozess bestimmt die Technik
  - Vollständige Nutzung der vorhandenen Möglichkeiten (frühes Scannen)
  - Beseitigung von Medienbrüchen

ASPECTA

Der Universal Insurance Workplace

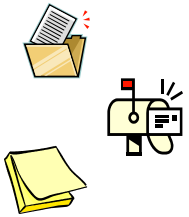
Die Idee

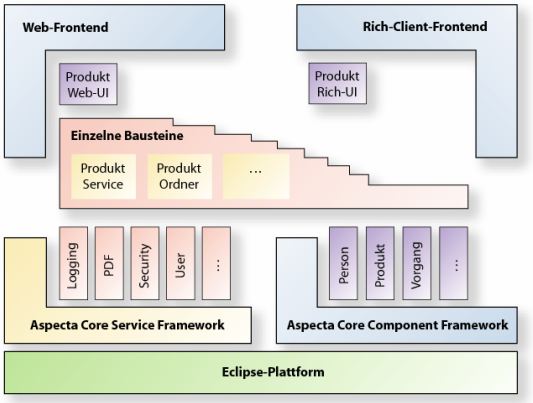
Entwicklung einer integrierten Anwendung

- Verbergen der Systemvielfalt
- Verdichten der relevanten Informationen
- Automatisieren einfacher Bearbeitungsschritte

Nutzung vertrauter Metaphern:

- Vorgangsmappe
- Postkorb
- Notizblock
- Werkzeug
- …





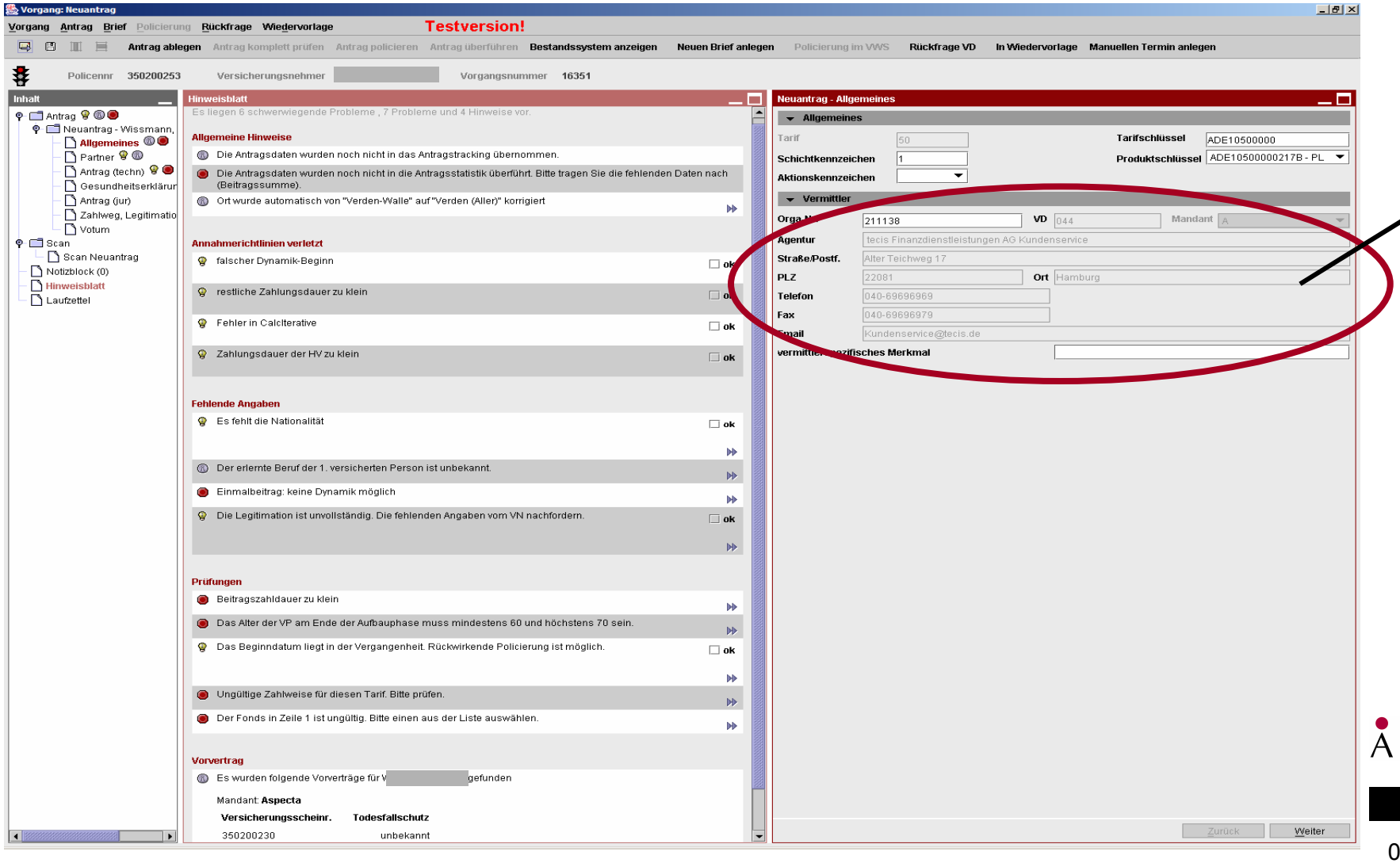
Schaffung einer zukunftsfähigen Basisarchitektur

- Erweiterbarkeit (weitere Geschäftsprozesse)
- Skalierbarkeit (weitere Anwender)

ASPECTA

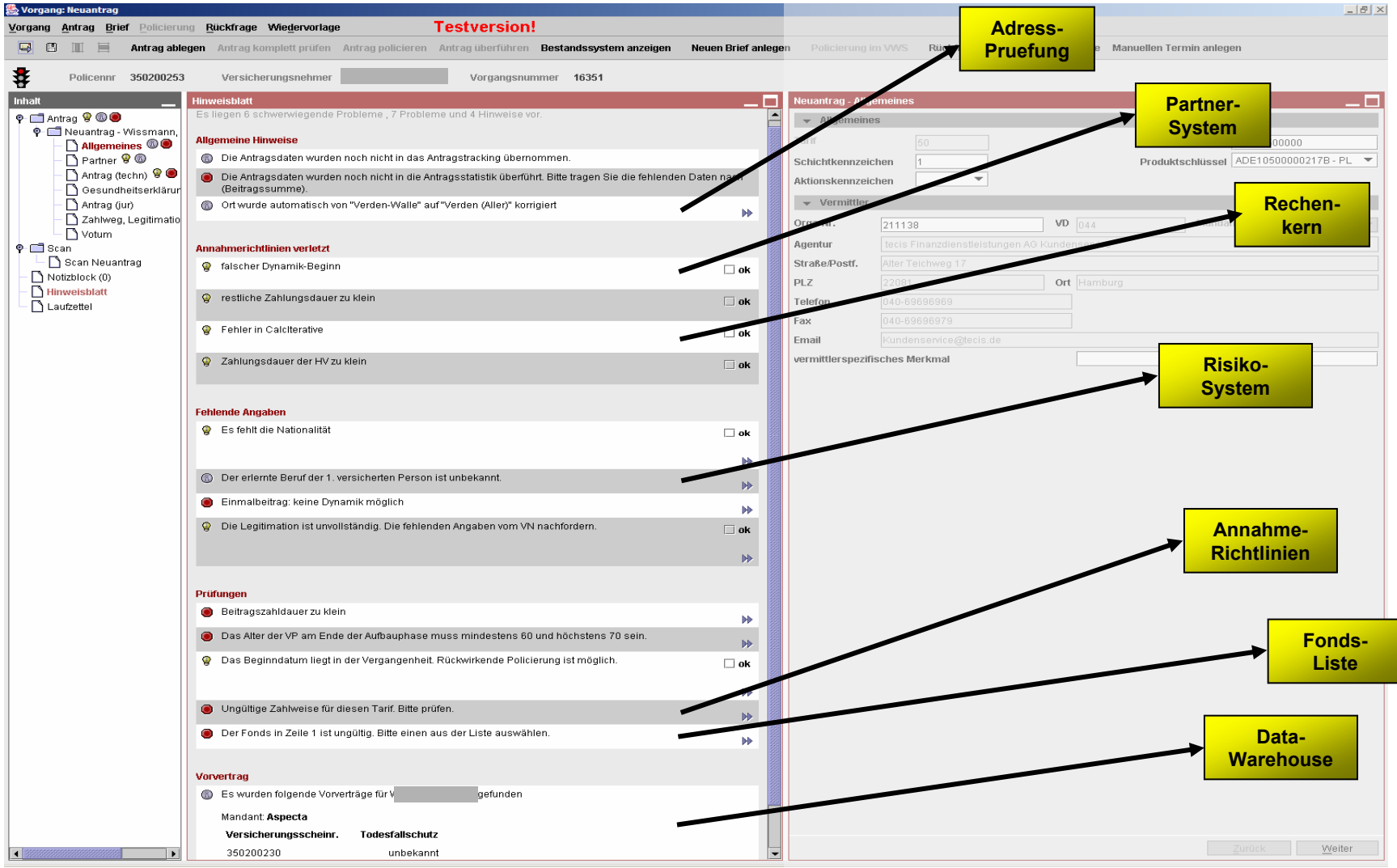
Hochintegrierte Sachbearbeitung

Das Ergebnis (am Beispiel des Antragssystems)



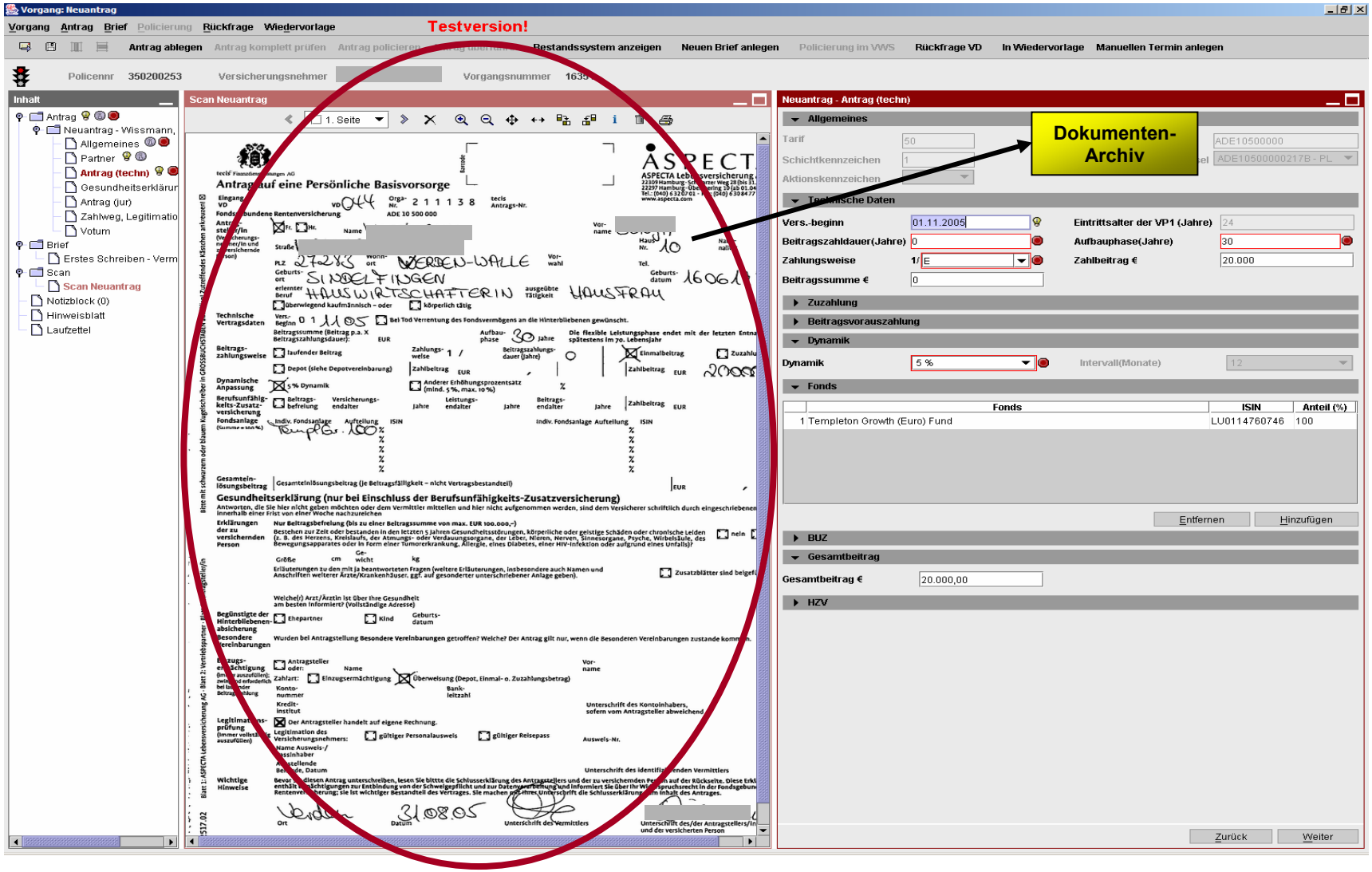
Hochintegrierte Sachbearbeitung

Das Ergebnis : Verbergen der Backendsysteme



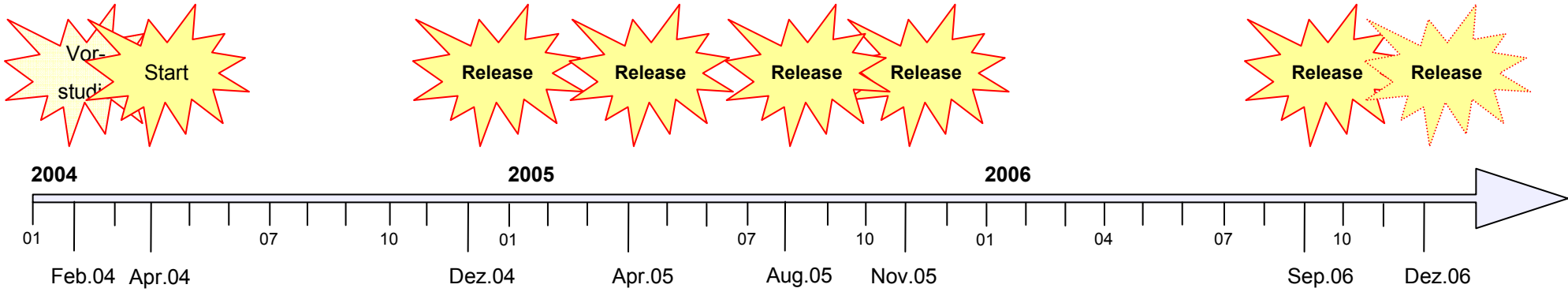
Hochintegrierte Sachbearbeitung

Das Ergebnis : schnellen Überblick verschaffen, fallabschließende Bearbeitung



Das Projekt im Überblick

Ein kurzer Blick zurück



Ergebnis : *„Wir steuern in die richtige Richtung...“*

- Erhebliche Reduktion der Prozesskosten
- Verringerung der Durchlaufzeiten um > 50 %
- Deutliche Erhöhung der Prozessqualität
- Hohe Anwenderakzeptanz

ASPECTA

Das GPO-System

Überblick

- Fachliche Architektur
- Technische Architektur
- Modularisierung
- Plattform
- Erweiterbarkeit

Die grundlegende Architektur - fachlich

Abbildung des Sachbearbeiter-Arbeitsplatzes auf die technische Welt

- Vorgangsmappen enthalten Dokumente
  - Alles, was zu einem Geschäftsvorfall gehört
  - Bildet eine Bearbeitungs-Einheit
- Dokumente können beispielsweise sein:
  - Scans, Anträge, Vertragsänderungen, Briefe, Termine, Notizen, etc.
- Vorgangsmappen können geöffnet, bearbeitet und gespeichert werden
  - Immer nur von einem Sachbearbeiter zur Zeit
- Postkörbe sind Aufbewahrungsorte für Vorgangsmappen
  - Vorgangsmappen können in Postkörbe weitergeleitet werden
  - Unterscheidung in persönliche, Gruppen- und technische Postkörbe

Die grundlegende Architektur – technisch (1/2)

Konzepte

- Services
  - Stellen Funktionalität zur Bearbeitung der Business Objects zur Verfügung (z.B. Vorgang-Service, Bankprüf-Service)
  - Bestehen aus Interface, Factory und (ggf. unterschiedlichen) Service-Implementierung(en)
  - Werden über zentralen Service-Manager zur Verfügung gestellt
- Business Objects (BO)
- (Fach-)Werte

- Werkzeuge (UIs)
  - Rahmen ist Postkorbwerkzeug (Anzeige der Postkörbe mit den enthaltenen Vorgangsmappen)
  - Einzelne Vorgangsmappe wird in einem Vorgangsmappen-Werkzeug dargestellt
  - Unterschiedliche Dokumente werden mit unterschiedlichen Dokument-Bearbeitern dargestellt
- Automaten
  - Dienen zur Automatisierung von Abläufen (Folgepost zuordnen, Prüfen, etc.)
  - Dienen zur Entkopplung von Abläufen (Briefschreibung, Rückfrage zuordnen)
  - Anwendungen ohne UI, beruhend auf der selben technologischen Basis

Die grundlegende Architektur – technisch (2/2)

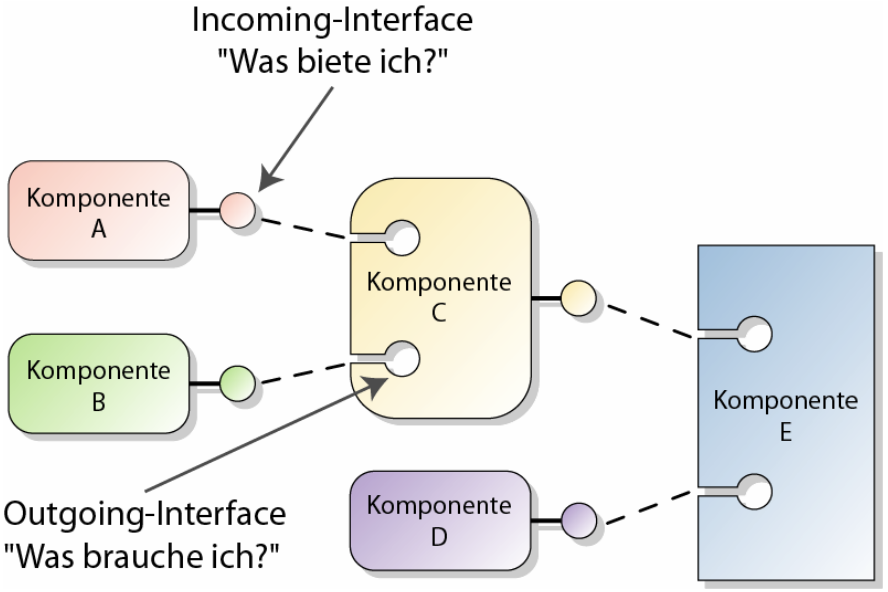
- Die Basis:
  - **Java 5**
  - **Equinox OSGi** Runtime (Eclipse-Plugin-Technologie)
  - **Extension-Registry** von Eclipse

- Weitere eingesetzte Technologien:
  - **Swing** (fürs UI)
  - **JDO** und **JDBC** (für die Persistenz)
  - **Spring** (fürs Remoting)
  - **JDBC**, **JNI**, **Axis**, etc. (für die Anbindung externer Systeme)

- Testing
  - Unit-Tests mit **JUnit**
  - Integrations-Tests mit **FIT** (noch am Anfang)
  - Hilfsmittel: **Mockrunner** (hauptsächlich JDBC-Mocks), **Easymock** (für alle anderen Mocks)

Was bedeutet Modularisierung mit Eclipse Plugins und OSGi?

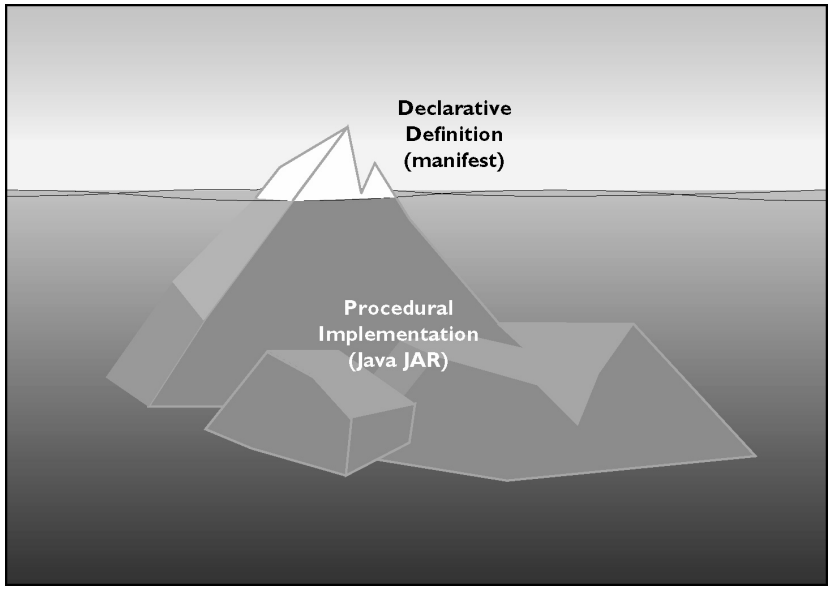
- Separate Einheiten
  - Ein eigener Classpath pro Modul
- Definierte Abhängigkeiten
  - Keine Zyklen
- Expliziter Im- und Export von Packages und Klassen



Deklaration vs. Code

Mehr als Abhängigkeits-Management…

- Trennung von Deklaration und Code mittels Manifest-Dateien
  - Dadurch gute Skalierbarkeit auch für große Systeme (mehrere tausend Plugins)
  - Schnelles Classloading



Aber: Nur Module/Plugins selbst reichen nicht

Was passiert, wenn mein System aus mehreren hundert oder tausend Plugins besteht?

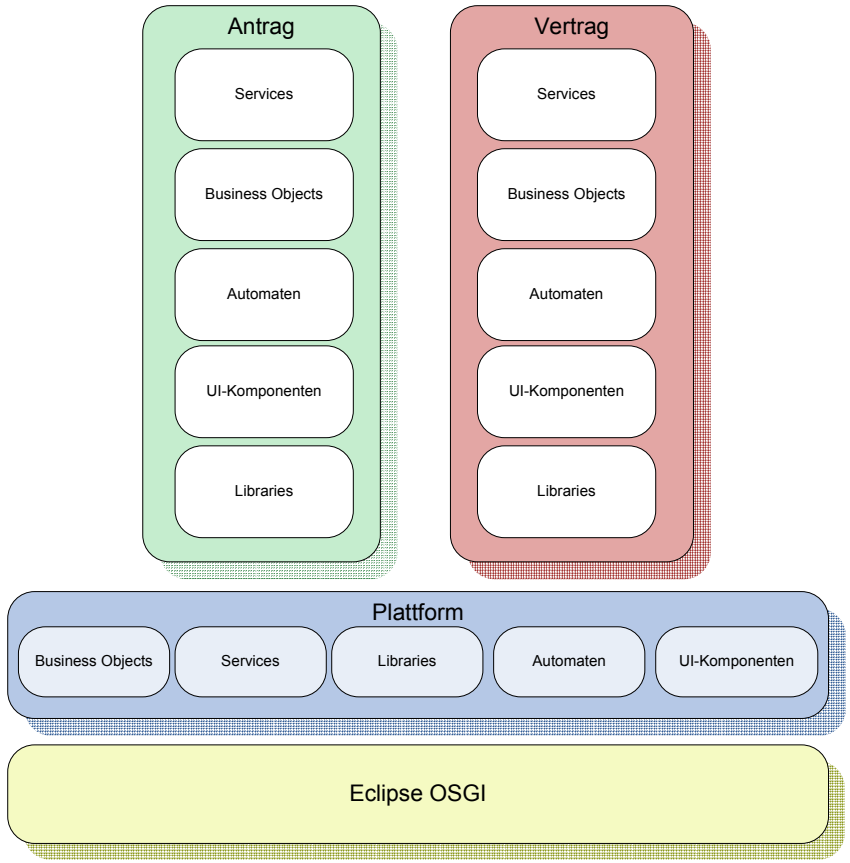
- Wir brauchen ein weiteres Strukturierungsmittel:
  - Erster Ansatz: Features in Eclipse
    - Fassen Plugins zu Features zusammen, als deploybare Einheit
    - Abhängigkeiten zwischen einzelnen Plugins trotzdem möglich
  - Zweiter Ansatz: Eine Plattform
    - Gemeinsamkeiten zwischen verschiedenen abgebildeten Geschäftsprozessen werden in einer allgemeinen Plattform für Lebensversicherungs-Anwendungen implementiert
    - Spezifische Geschäftsprozesse werden unabhängig voneinander auf Basis der Plattform implementiert

Plattform-basierte Entwicklung

Eine gemeinsame Plattform

Geschäftsprozesse als Säulen

- ✚ Integrierte Benutzungsoberfläche mit homogenem Verhalten
- ✚ Leichte Wiederverwendung gemeinsamer Konzepte und Komponenten
- ✚ Entwicklung lässt sich gut skalieren, indem Teams parallel an unterschiedlichen Säulen implementieren können



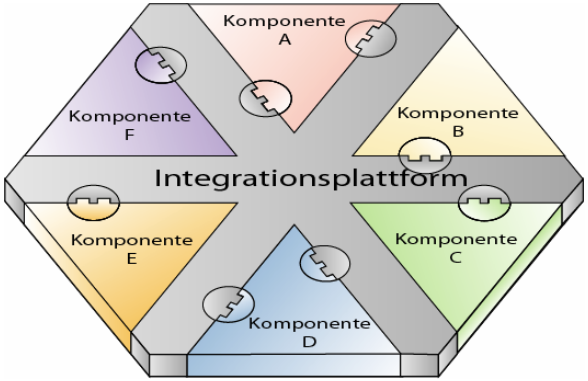
Nicht neu erfunden, sondern abgeschaut bei Eclipse:

- Bei Eclipse gibt es die IDE-Plattform, auf deren Basis unterschiedliche IDEs implementiert werden können

ASPECTA

Was bedeutet das?

Plattform-basierte Entwicklung



- Kernkonzepte werden in der Plattform implementiert
  - Gemeinsamkeiten heraus-faktorisieren
  - Kernkonzepte identifizieren und implementieren
- Spezialitäten werden außerhalb implementiert
  - Können auf den Kernkonzepten aufbauen, das fördert die fachliche Integration
  - Es stellt sich die Frage, wie die Plattform spezialisiert werden kann
- Kann nicht Up-Front passieren
  - Plattform-Entwicklung ist ein evolutionärer Prozess

Erweiterbarkeit

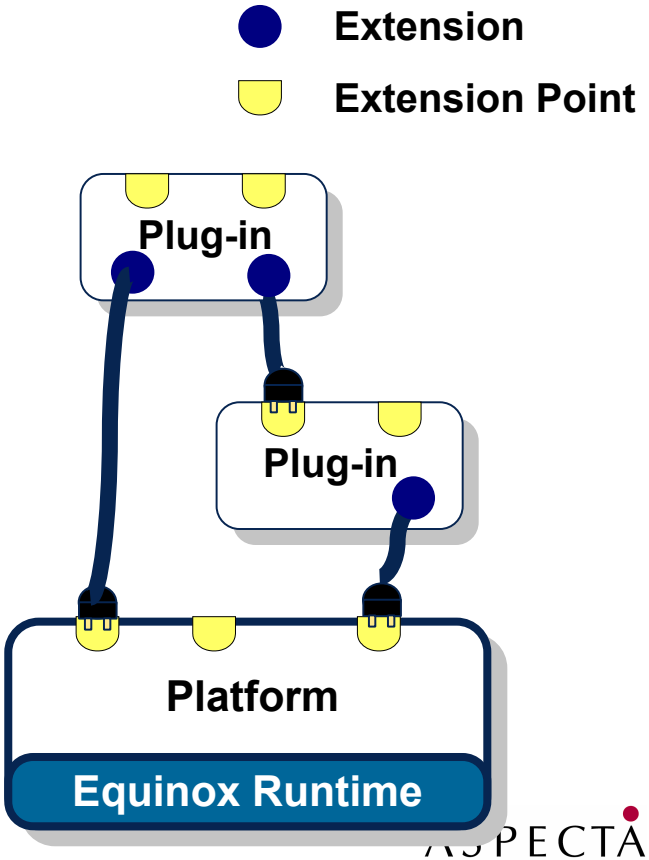
Erweiterbarkeit ist entscheidender Faktor

- Erweiterbarkeit ergibt sich nicht von selbst
  - Erweiterbare Teile der Plattform müssen sorgfältig entworfen und implementiert werden
  - *„Der erste Wurf ist fast immer falsch oder unpassend“*
- Möglichkeiten für Erweiterungs-Mechanismen
  - Benutzung und Vererbung
  - Dependency Injection
  - **Extension-Points**
- Wichtig:
  - **Klar definieren und dokumentieren, was erweiterbar ist und was nicht**

Extension-Points und Extensions

Der Extension-Point-Mechanismus von Eclipse eignet sich hervorragend, um eine eigene erweiterbare Plattform zu realisieren

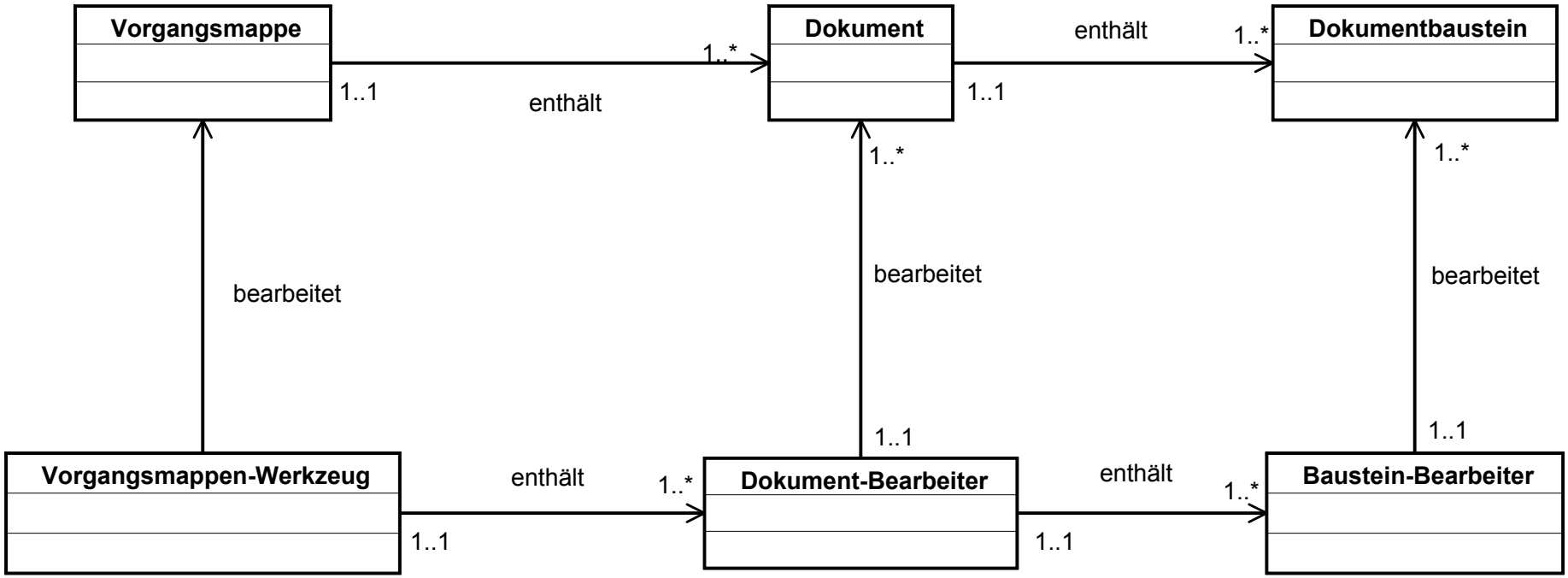
- Der Anbieter eines Extension-Points definiert, was er von Extensions erwartet
  - Schema
  - Interface
  - Ggf. weitere Dokumentation
- Der Anbieter ermittelt zur Laufzeit, welche Extensions existieren und nutzt diese entsprechend
- Der Lieferant einer Extension erfüllt den Vertrag des Extension-Points und fügt seine Implementation auf diesem Wege dem Anbieter hinzu



ASPECTA

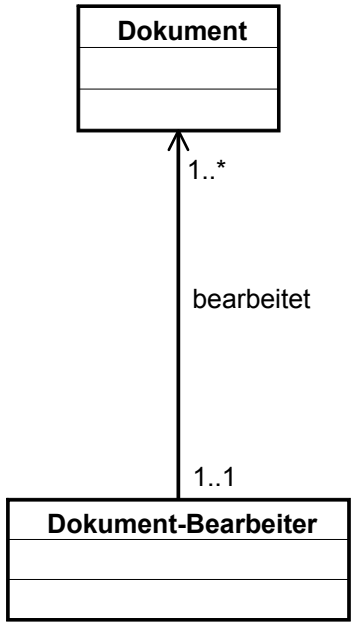
Beispiel: Vorgangsbearbeitung

Strukturübersicht



Beispiel: Vorgangsbearbeitung -> Dokument-Bearbeiter

- 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)

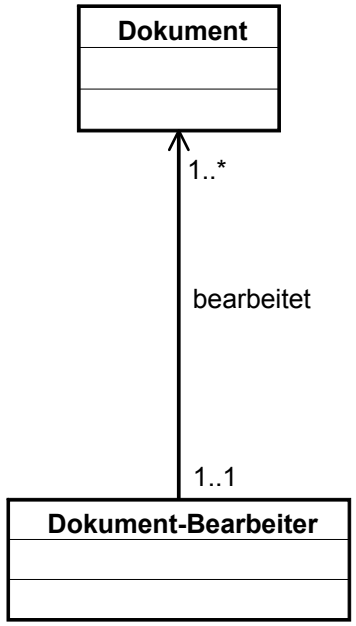


Beispiel: Vorgangsbearbeitung -> Dokument-Bearbeiter

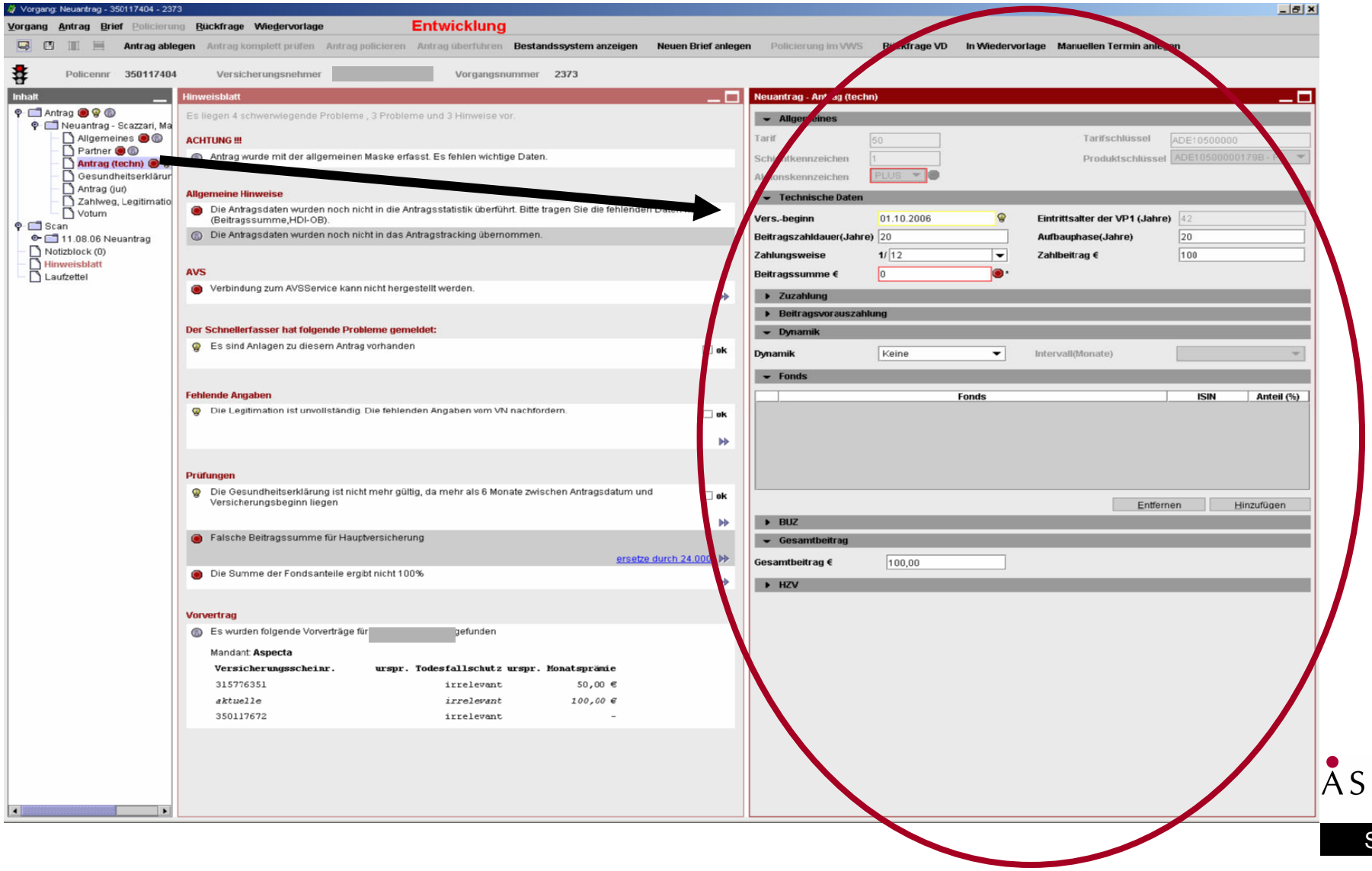
+ Man kann flexibel neue Dokumente und ihre entsprechenden Bearbeiter dem System hinzufügen, ohne das Vorgangsmappen-Werkzeug selbst zu verändern

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

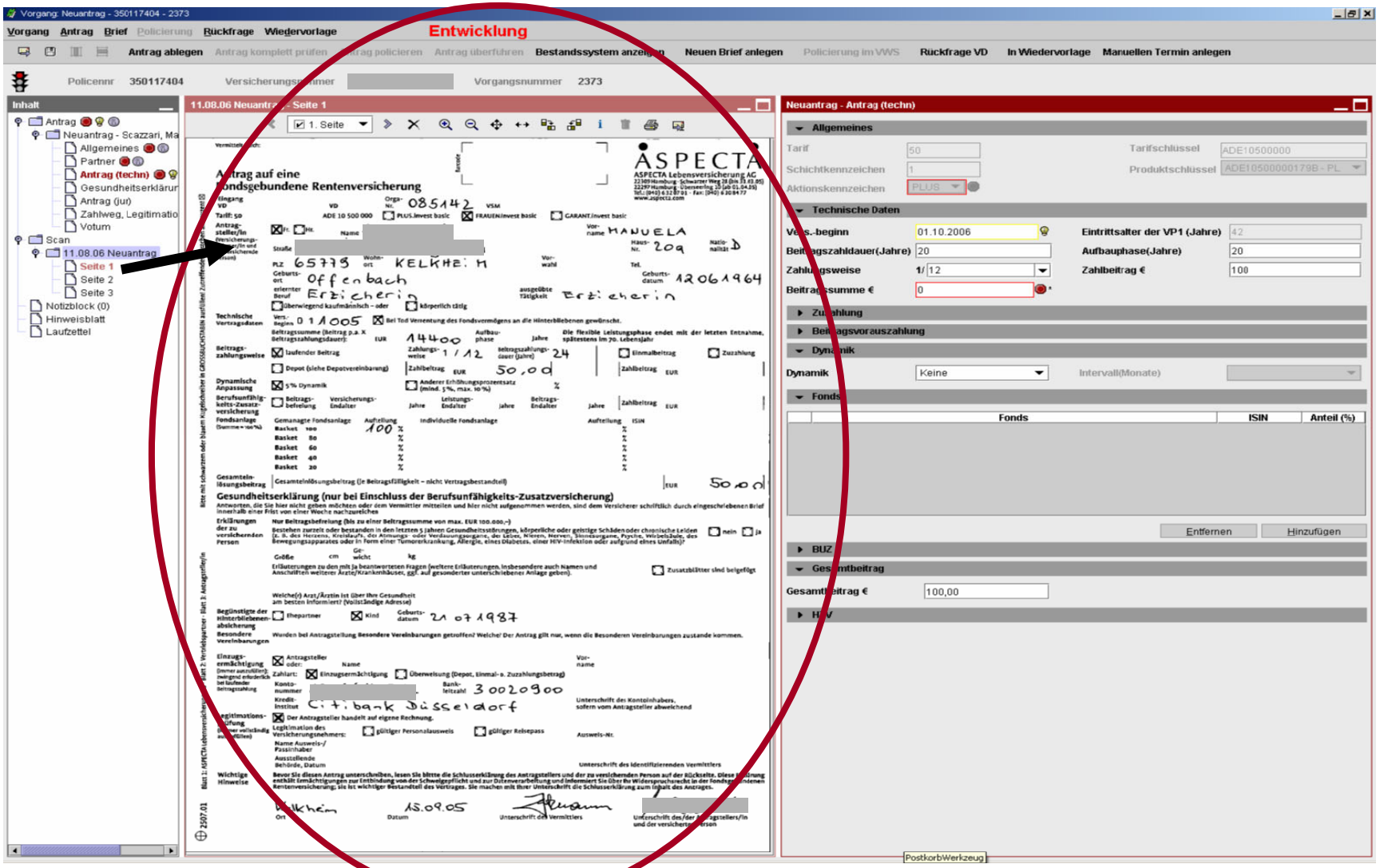
  - Es gibt bei uns beispielsweise einen vollständigen Antrags-Bearbeiter für die Hauptverwaltung und einen eingeschränkten Antrags-Viewer für die Geschäftsstellen



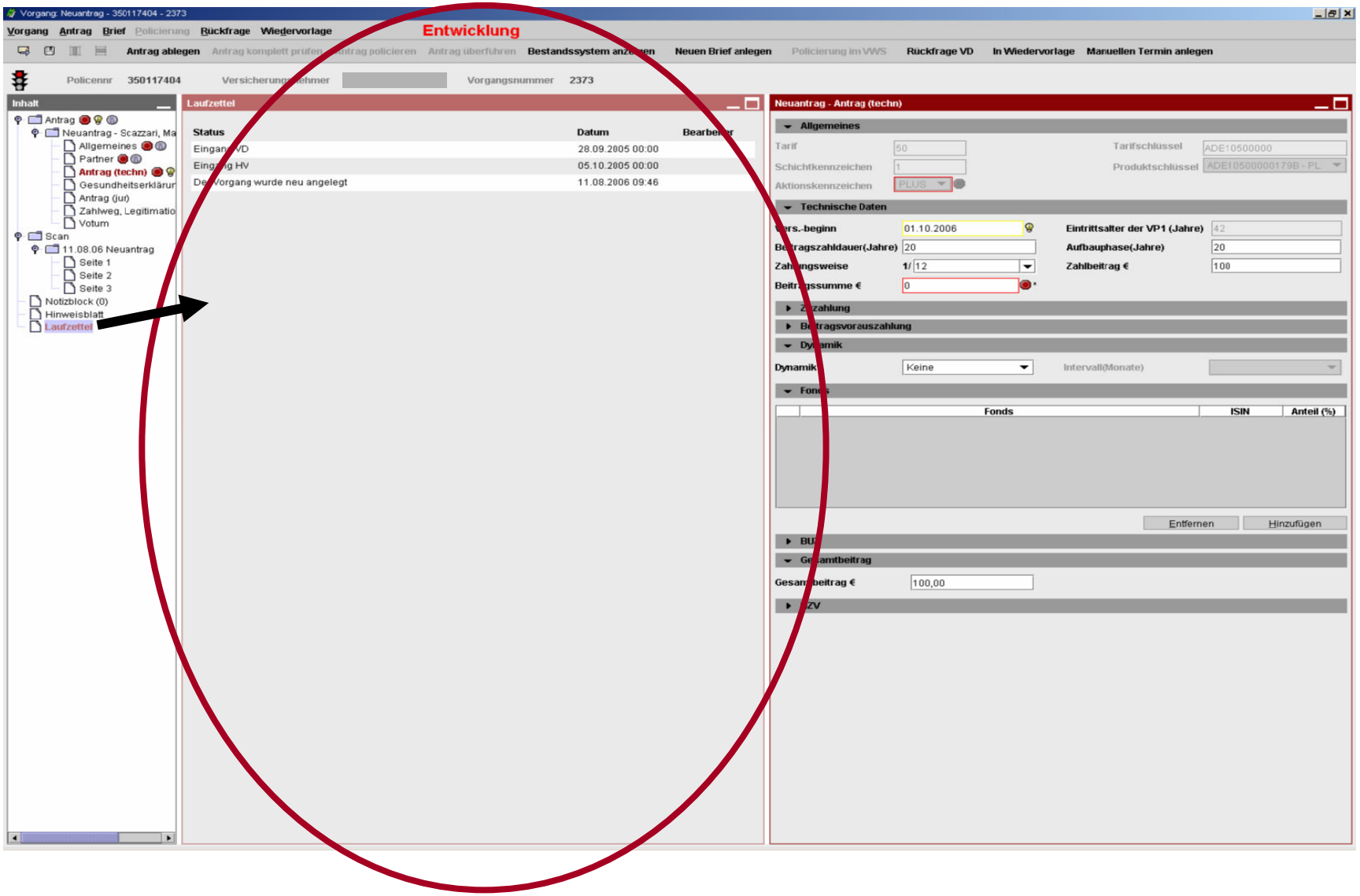
Dokument-Bearbeiter: Antrag



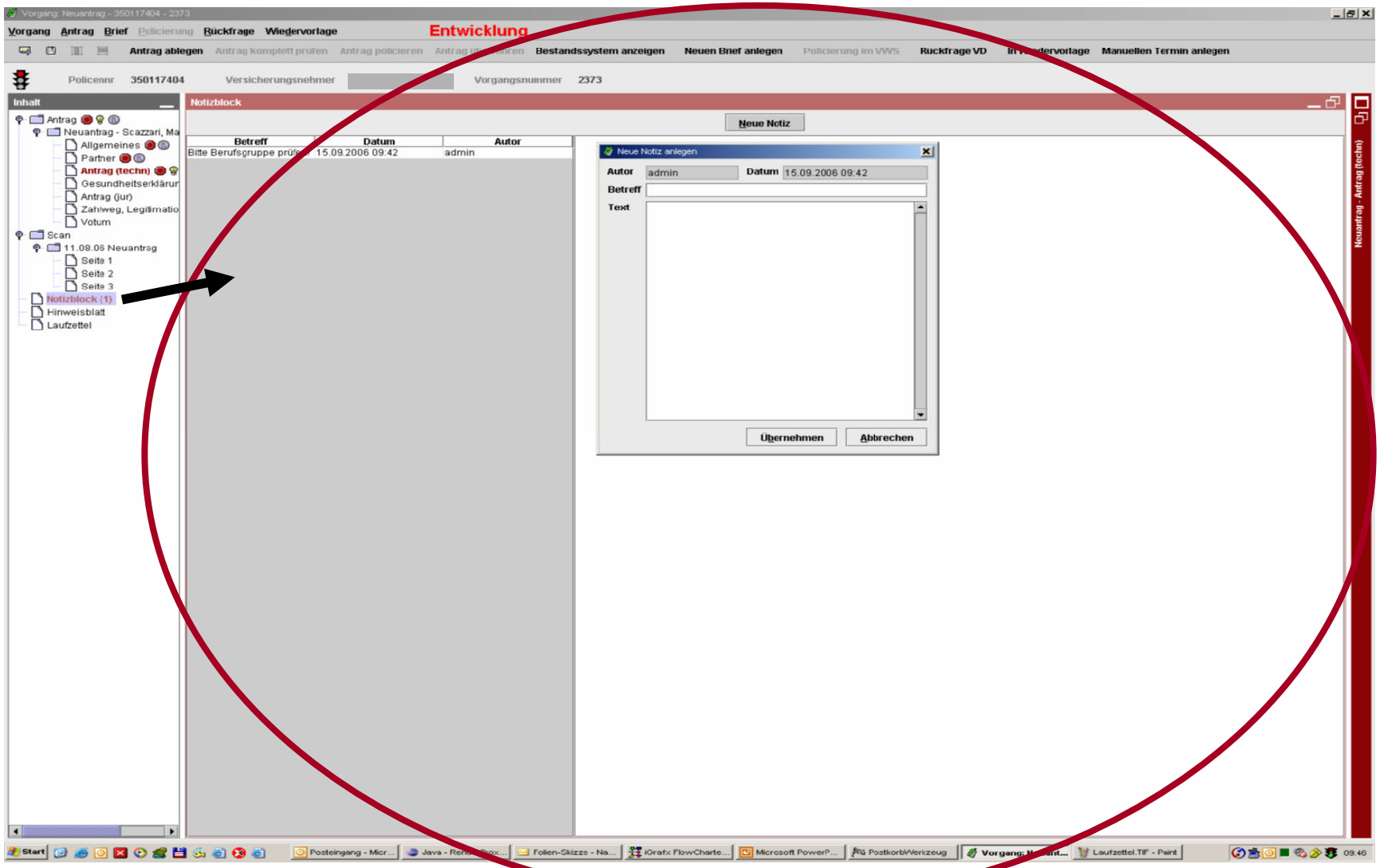
Dokument-Bearbeiter: Scan



Dokument-Bearbeiter: Laufzettel



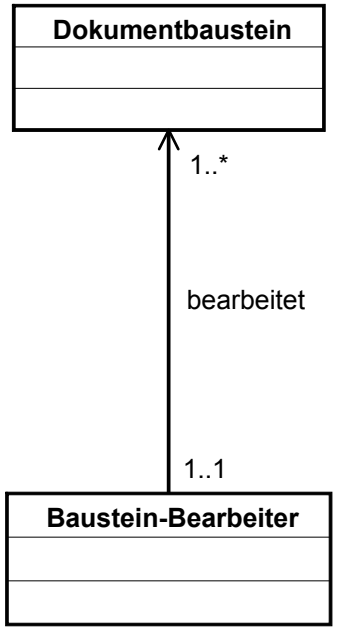
Dokument-Bearbeiter: Notizblock



Beispiel: Vorgangsbearbeitung -> Baustein-Bearbeiter

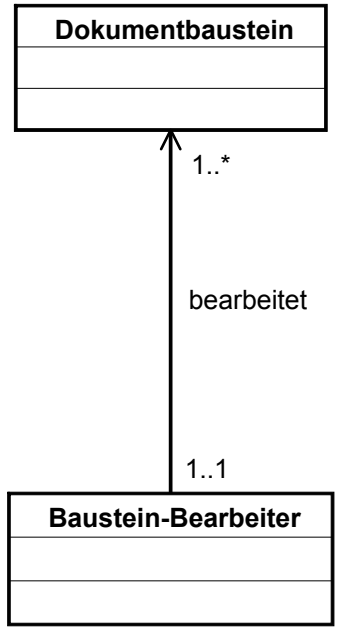
Gleiches Prinzip wie bei Dokument-Bearbeiter, eine Ebene tiefer:

- Baustein-Bearbeiter ist ein Java-Interface und ein Extension-Point
- Java-Klassen, die Baustein-Bearbeiter implementieren, enthalten den vollständigen Code, um Bausteine eines Typs darstellen und editieren zu können
- Insbesondere gibt es Methoden zum Übernehmen der UI-Felder in die Baustein-Properties und umgekehrt
- Jede Extension, die sich hier anmeldet, deklariert, für welchen Typ von Baustein sie zuständig ist
- Die Zuordnung von Baustein-Typen und ihren zuständigen Bearbeitern erfolgt ausschließlich über die Konfiguration (plugin.xml)



Beispiel: Vorgangsbearbeitung -> Baustein-Bearbeiter

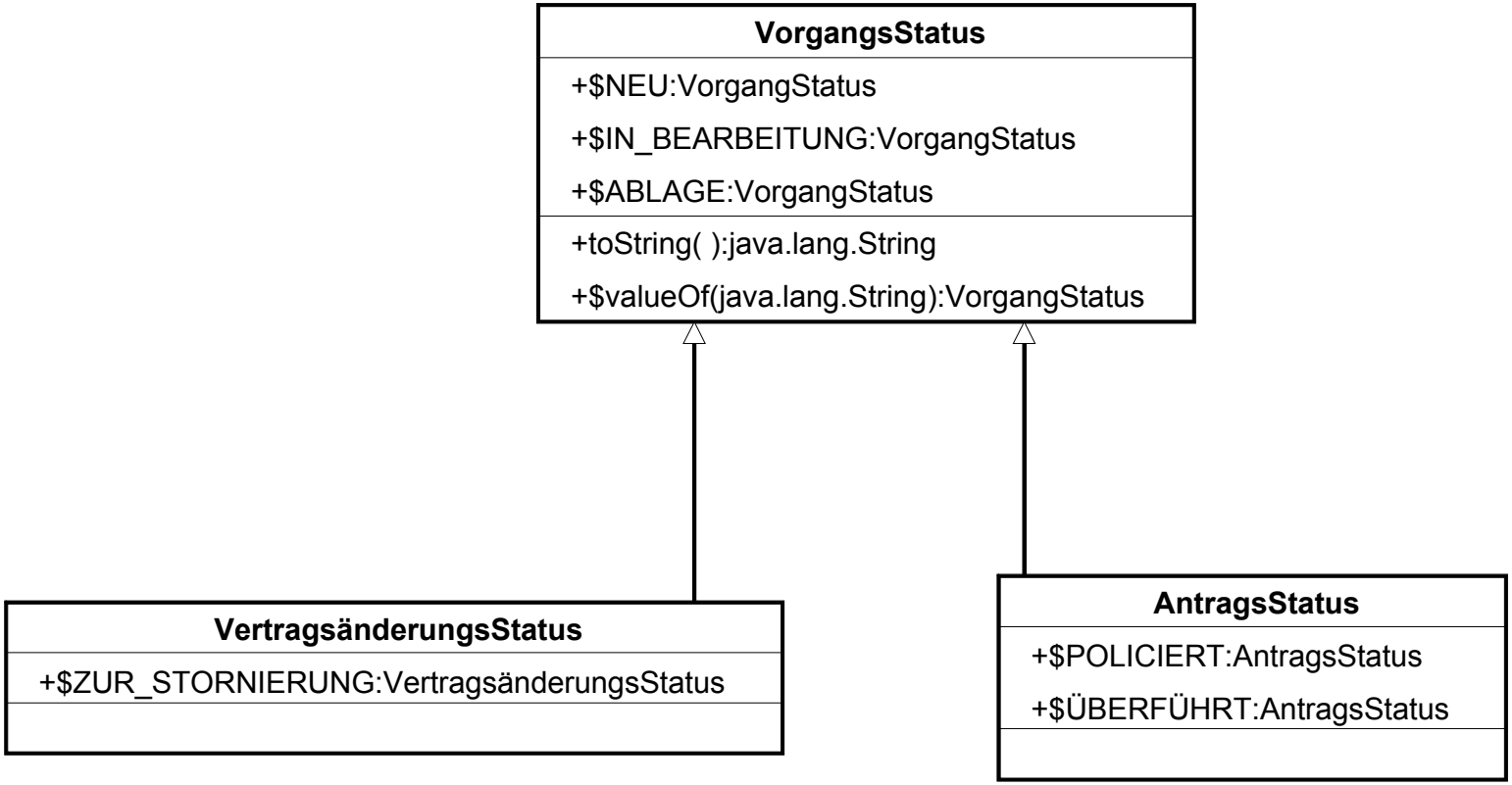
- Dokumente lassen sich flexibel aus Bausteinen zusammensetzen (macht natürlich nur Sinn, wenn das Dokument selbst so eine Struktur nahe legt)
- UIs lassen sich flexibel hinzufügen
- Wenn man einen bestimmten Bausteintyp in einem weiteren Dokument verwendet, hat der automatisch schon einen passenden Bearbeiter



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.

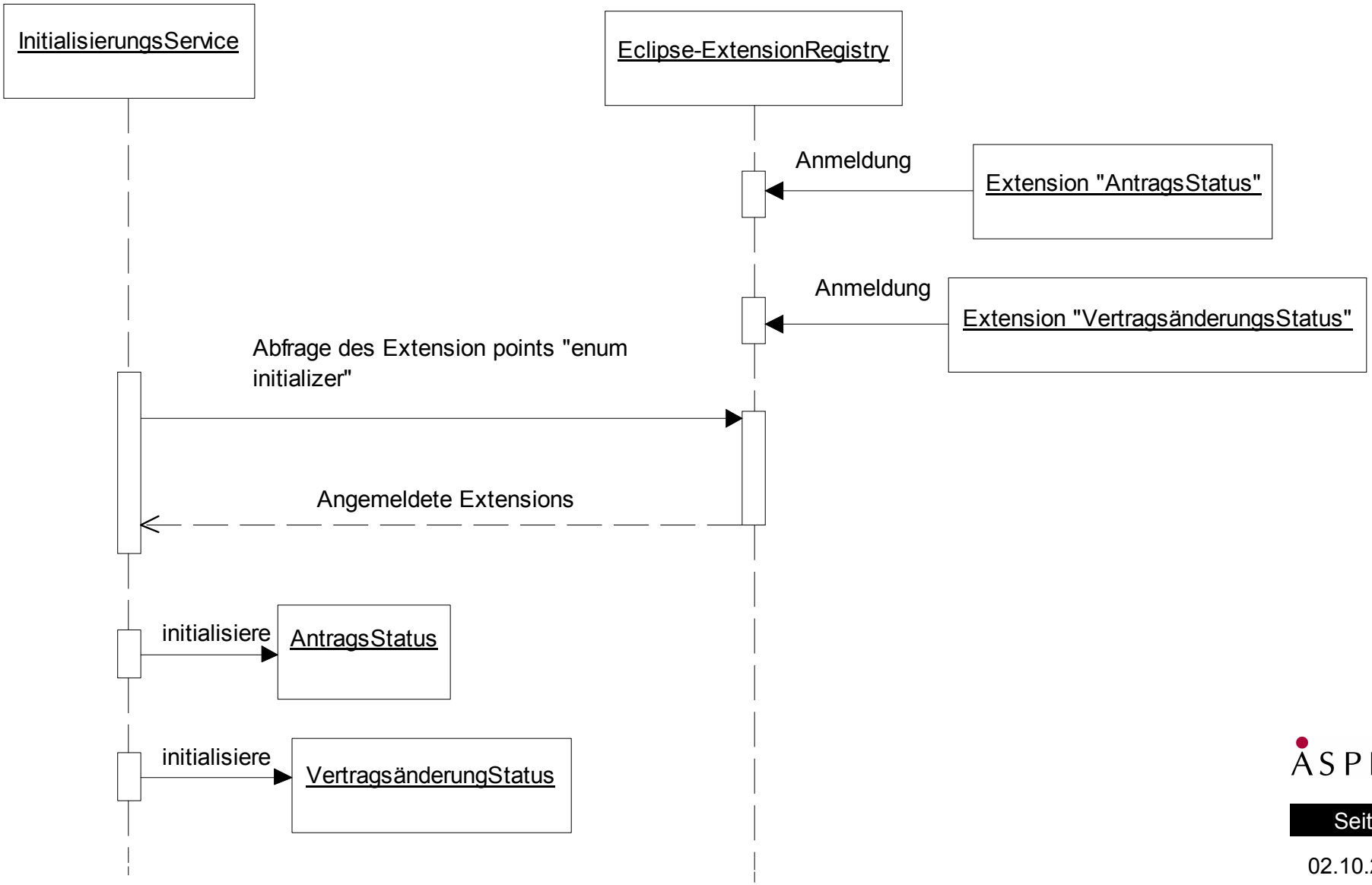
Wir benötigen also eine verlässliche Initialisierung der Klassen erweiterbarer Wertetypen zum Programmstart!

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

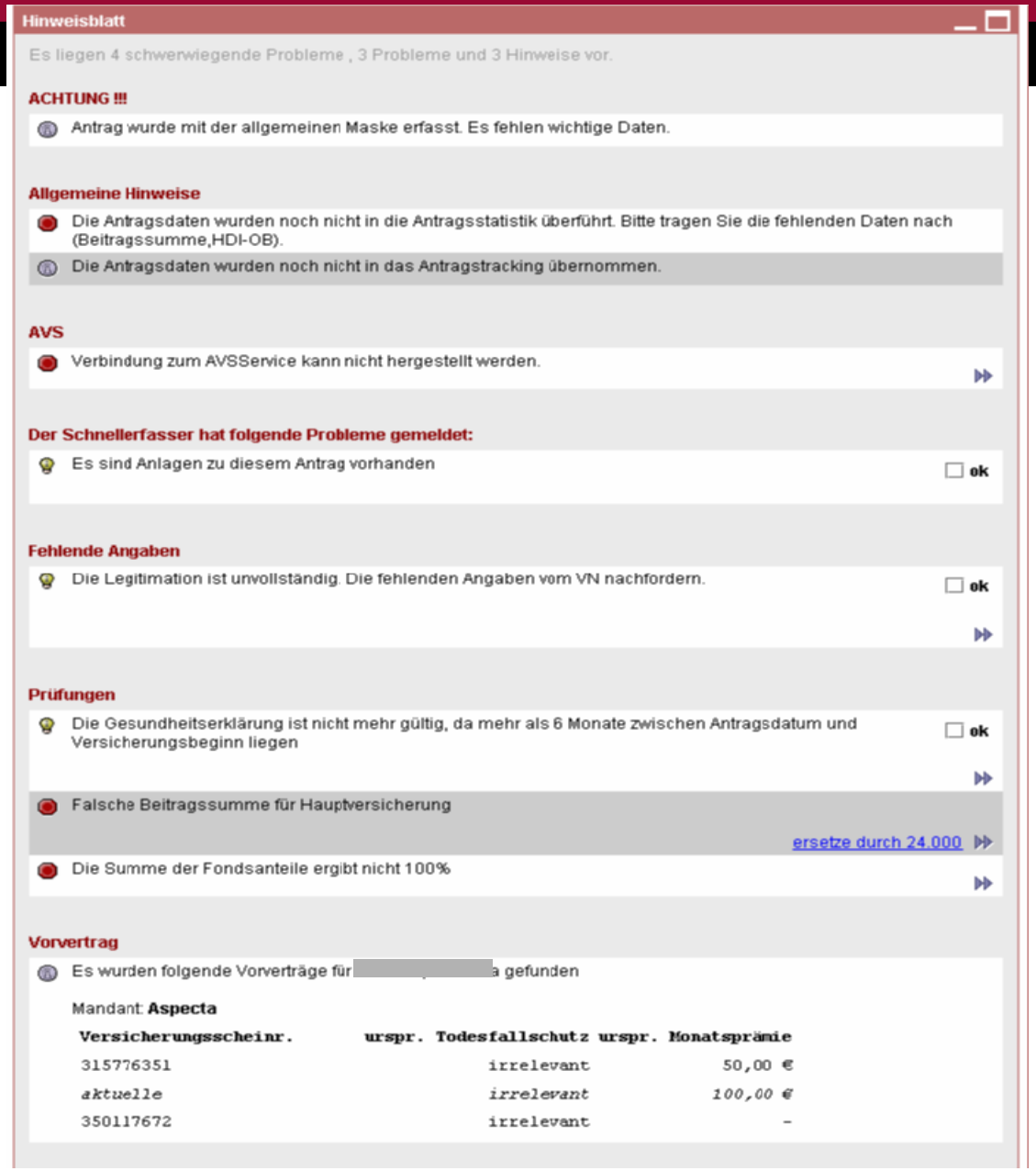


Beispiel: Marker

Wie zeigt man Prüfergebnisse und andere Hinweise an?

- Das Marker-Konzept:
  - Jedes Dokument in einer Mappe kann so genannte „Marker“ liefern
  - Ein Marker kann einen Hinweis geben oder Problemstellen aufzeigen
  - Ein Marker enthält einen Text, der den Hinweis oder das Problem beschreibt
  - Ein Marker kann zu einer thematischen Kategorie gehören
- Marker können generisch am UI dargestellt werden
  - Marker erscheinen automatisch auf dem übergreifenden Hinweisblatt der Vorgangsmappe
  - Marker erscheinen zusätzlich automatisch an der entsprechenden Problemstelle im Bearbeiter
  - Marker werden per HTML dargestellt
- Woher die Marker kommen, hängt von der konkreten Konstruktion ab
  - Prüfungen hängen Marker als Prüfergebnisse an Dokumente an
  - Dokumente können auch je nach ihrem Zustand selbst Marker liefern

Beispiel: Marker am UI

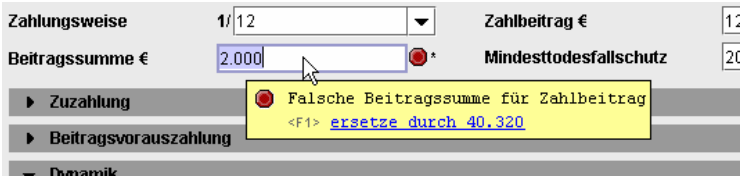


Beispiel: Marker-Fixes

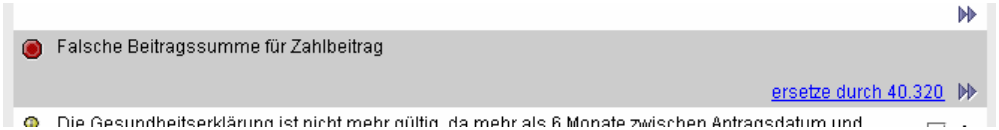
Bei einigen Problemen kann das System direkt helfen:

- Per Extension-Point können so genannte Marker-Fixes hinzugesteckt werden:
  - Orientiert sich an den Quick-Fixes in der Eclipse-IDE
  - Ein Marker-Fix bietet eine Lösung für ein Problem an
  - Wird am UI dargestellt und kann per Hyperlink aktiviert werden
  - Das System ermittelt zu allen Markern passende Marker-Fixes
  - Für ein Problem können auch mehrere unterschiedliche Marker-Fixes existieren

Im Antrags-Bearbeiter:



Auf dem Hinweisblatt:



Beispiel: Neue Produkte

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

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: Verteilte Services

Ausgangssituation:

- Services sind als POJOs realisiert

Die Idee:

- Services mittels Spring-Remoting remote-fähig machen und in das System einklinken

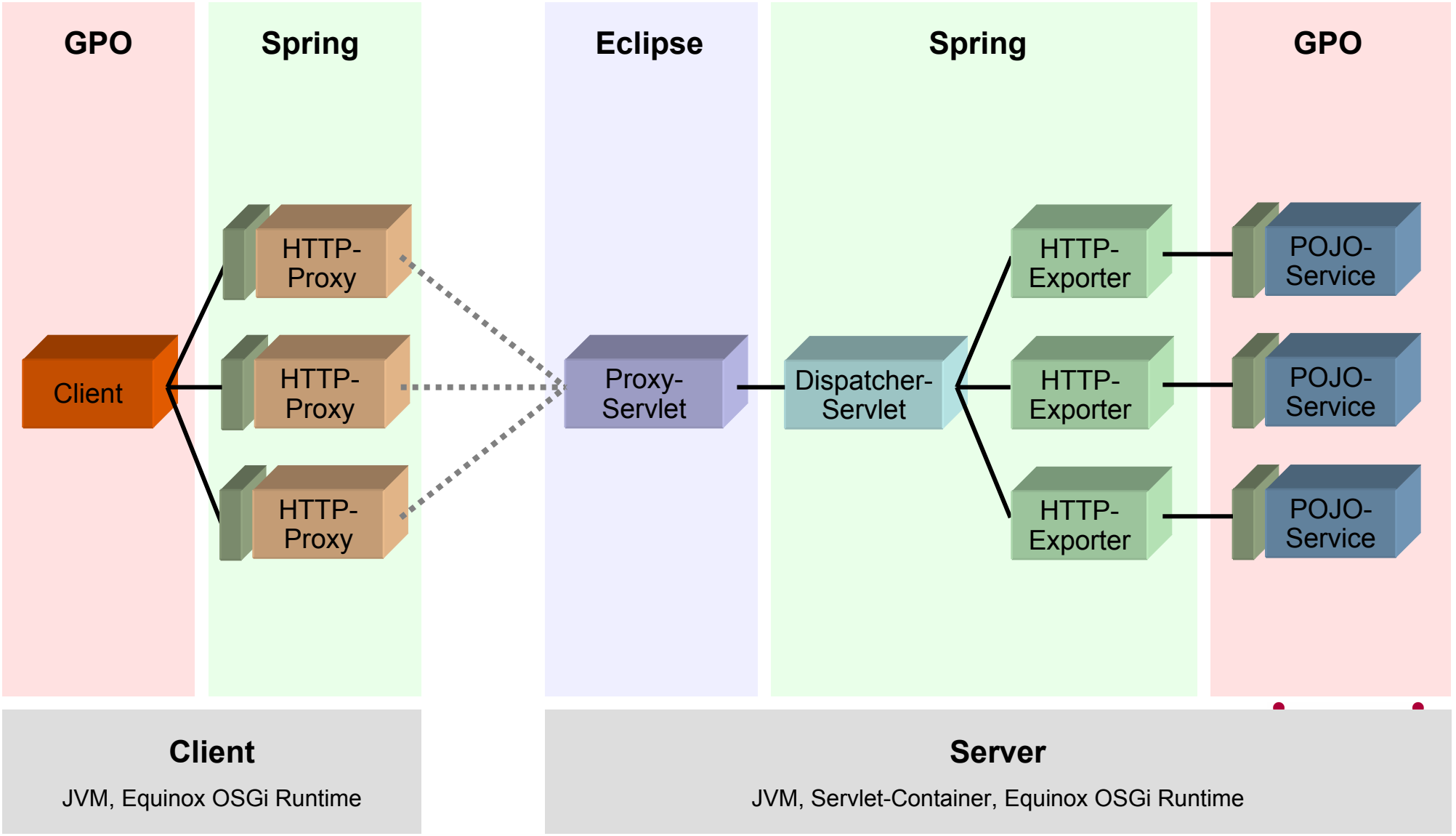
Das Spring Framework:

- Spring wird als ein Plugin dem System hinzugefügt
- Mit der Spring-OSGi-Integration (in Spring 2.1) können alle Plugins das Spring Framework nutzen

Verteilte Services:

- Werden per Spring-Definition remote-fähig gemacht
- Und über den Service-Extension-Point wird beispielsweise der Remote-Proxy angezogen

Verteilte Services mit Spring und Eclipse



Lessons Learned

Wir fassen zusammen:

Was haben wir gelernt?

Was würden wir wieder so machen?

Was würden wir anders machen?

Lessons Learned: Modularisierung

Was sind die Schwierigkeiten?

- Getrennte Module können schwierig werden, wenn…
  - Bibliotheken Java-Reflection verwenden (müssen), z.B. Persistenz-Frameworks
  - Bibliotheken implizit dynamische Abhängigkeiten erzeugen (beispielsweise Hibernate mit generierten Proxies)
  - Bibliotheken merkwürdiges Classloading machen
  - Libraries eigene Versionen von Open-Source-Bibliotheken mitbringen

  - *Manchmal ist das eigentliche Problem schwer zu identifizieren*

- **Aber:** Es gibt immer eine Lösung!
  - Buddy-Loading
  - Wrapper

  - Eclipse 3.2 ist da um vieles freundlicher geworden… ☺

Lessons Learned: Stabile Plattform

Wie ist unser Stand?

+ Wir haben getrennte Builds für die Plattform und die Säulen, aber:
– Eine stabile Plattform ergibt sich nicht von allein, nur weil man getrennte Builds hat

Was haben wir gelernt? Was würden wir anders machen?

! Mit einer kleinen, schlanken Plattform starten
! Diese kleine Plattform schrittweise erweitern
! Explizit API-Design als Aufgabe verstehen und entsprechende Reviews bei Aufnahme in die Plattform durchführen
! Getrennte Teams unterstützen den Plattform-Gedanken, können aber auch hinderlich sein

Lessons Learned: Extension-Points

Erweiterbarkeit auf Basis von Extension-Points:

+ Der Extension-Point-Mechanismus ist einfach zu verstehen und einfach zu verwenden (kann selbst ohne OSGi-Runtime genutzt werden)
+ Erlaubt es, sehr flexible Architekturen zu realisieren
+ Erlaubt relativ einfach „späte Erzeugung“ und dadurch gute Skalierbarkeit

! Bei variablen Teilen immer überlegen, ob die Flexibilität nicht durch einen Extension-Point herausgezogen werden sollte
! „Always have a client“ (YAGNI, keine EP auf Vorrat)

– Zu viele Extension-Points erschweren die Übersicht, Dokumentation notwendig

Lessons Learned: Interfaces und abstrakte Klassen

Was hilft, mit „published APIs“ umzugehen?

- Interfaces:
  - + Definieren die eigentlichen Begriffe und Services des Plugins
  - + Sind für Extension-Points elementar wichtig
  - + „Programming to an interface, not an implementation“
- Abstrakte Klassen:
  - + Können als Default-Implementationen von Interfaces helfen, API-Änderungen abzufedern

Lessons Learned: Zuschnitt von Plugins

Eine immer wiederkehrende Frage:

Wie schneidet man Plugins?

- Unsere Plugins (insgesamt ~260) sind relativ klein geschnitten
  - \+ Feine Granularität, angebundene Systeme beispielsweise separat in einzelnen Plugins gekapselt
  - \+ Dadurch einzelne Funktionsbereiche gut voneinander getrennt und separat nutzbar
  - – Viele Projekte, viele Abhängigkeiten
  - ! Wir würden in Zukunft zu weniger Plugins tendieren
- Im Eclipse-SDK sind die Plugins recht groß geschnitten
  - \+ Wenige Projekte, bessere Übersicht
  - – Gröbere Granularität, daher viele Teile nur „als Ganzes“ verwendbar

Lessons Learned: Library-Plugins

Verpackt man Libraries in eigene Plugins oder direkt in das Plugin, welches die Library verwendet?

- In jeweils eigene Plugins:
  - + Klar definierte Abhängigkeiten
  - + Auch Versions-Abhängigkeiten können dediziert deklariert und damit behandelt werden
  - – Es werden immer mehr Plugins
- In das verwendende Plugin:
  - + Weniger Projekte, einfaches Handling, auch bei Updates der Library
  - – Teilweise die gleiche Library dutzendfach im Workspace
  - – Man muss sehr vorsichtig mit den Exports sein, sonst gibt es Versions-Konflikte und ClassCast-Exceptions

Lessons Learned: Persistenz mit JDO

Persistenz ist überwiegend mit JDO realisiert
(einer kommerziellen Implementation)

Was haben wir gelernt?

- ! Verwende niemals ein Framework entgegen seines intendierten Einsatz-Szenarios
- ! JDO ist deutlich langsamer als direkter JDBC-Zugriff (Faktor 5-10)
- ! Entscheide dich nie für ein so grundlegendes technisches Element aufgrund von politischen oder strategischen Überlegungen
- ! Implementiere immer einen Real-World-Prototypen mit der gewählten Technologie, bevor sie in der Breite eingesetzt wird, auch unter dem Performance-Gesichtspunkt

- – Closed-Source-Implementation erweist sich immer wieder als hinderlich im Projektalltag
- – Enhancement des Bytecodes beim Kompilieren verzögert den Auto-Build enorm, kein flüssiges Arbeiten in Eclipse mehr

- – Spätes Ersetzen des Persistenz-Frameworks kann aufwendig sein, selbst wenn der Code gut gekapselt ist

Lessons Learned: Refactoring

Je größer das System wird und je länger es weiterentwickelt wird (oder werden soll), desto wichtiger sind Refactorings!

Unsere Erfahrungen:

- ! Refactorings müssen kontinuierlich durchgeführt werden
- ! Refactorings müssen dann durchgeführt werden, wenn der Code-Smell bemerkt wird
- ! Aufgeschobene Refactorings führen zu immer größer werdenden Aufwänden

- – Published-APIs machen Refactorings teilweise schwieriger
- – Die Sicht auf Softwareentwicklung als Projekt erweist sich gerne mal als hinderlich in diesem Zusammenhang:
  - Das Projektziel selbst ist in der Regel **nicht**, ein wartbares, gut strukturiertes System zu implementieren
  - Produkt-Sicht oftmals hilfreicher

- ? Immer wieder offener Punkt:
  - Wie plane ich Refactorings ein? Verstecke ich Refactoring-Aufwände in den normalen Features?

Lessons Learned: Frameworks

Es gibt häufig Diskussionen darüber, wie hilfreich Frameworks wirklich sind.

- Unsere Erfahrungen:
  - + Wir haben gute Erfahrungen mit kleinen und leichtgewichtigen Frameworks gemacht.
    - z. B. für Dokument-Bausteine und Baustein-Bearbeiter
  - + Lassen sich einfach implementieren
  - + Erlauben schnelle Entwicklung in die Breite
  - – Aber: API-Problematik kommt sehr schnell ins Spiel

Lessons Learned: Testen von Beginn an

Automatisierte Tests sind überlebensnotwendig, vor allem, je größer ein System wird und je länger es entwickelt wird

- Unsere Erfahrungen:
  - ! JUnit-Tests sind gut, müssen aber schnell ausführbar sein und dürfen den eigenen Testdaten-Bestand nicht zerstören
  - ! Man muss frühzeitig mit automatisierten Integrations- bzw. Akzeptanztests (beispielsweise mit FIT) beginnen
    - Später sind solche Tests schwer und aufwendig zu implementieren
  - ! Testplan bzw. Drehbuch für manuelle Tests durch die Anwender notwendig
  - ! Für die Entwicklung ist ein hoher Grad an Automatisierung zwingend erforderlich
    - Der manuelle Test-Aufwand erstickt sonst die kurzen Release-Zyklen
    - Die Qualität des Systems sinkt kontinuierlich
  - ! Und nicht zuletzt: Alle Entwickler müssen an einem Strang ziehen

Lessons Learned: Nicht alles im UI-Thread

Kann ich die UIs „einfach“ so runterprogrammieren?

- **Achtung!!!**
  - ! Man muss sehr genau darauf achten, was man im UI-Thread macht und was nicht
    - Siehe auch diverse Bücher zu Swing
  - ! Eine Richtlinie: Möglichst wenig im UI-Thread tun
    - Damit macht das UI einen „flüssigen“ Eindruck
  - ! Die Folge: Man muss sehr sauber mit dem Swing-Thread-Modell umgehen und auf Multithreading achten

- Es hilft, wenn man passende Richtlinien und/oder passende kleine Frameworks zur Hand hat, die einem die wesentlichen UI-Threading-Aufgaben abnehmen

Lessons Learned: Prüfungen

Wie und wo implementiert man fachliche Prüfungen?

- Unser erster Ansatz: Sehr starke Modularisierung
  - – Führte zu Schwierigkeiten bei abhängigen Prüfungen und Prüf-Reihenfolgen
  - – Viele Dinge wurden doppelt und dreifach geprüft (kostet Performance und erschwert die Wartung)
- Marker haben sich bewährt!
  - \+ Prüf-Ergebnisse als Marker sind sehr flexibel

Fragen ???

Fragen jederzeit gerne!

Folien online auf der Seite des Arbeitskreises oder unter:
http://www.it-agile.de/newsfeed.html

Ivo Eitner: ieitner@aspecta.com
Markus Herzog: maherzog@aspecta.com
Gernot Neppert: geneppert@aspecta.com
Martin Lippert: martin.lippert@akquinet.de

akquinet

it-agile

ASPECTA