



Darf es etwas mehr sein?

Einführung in den „Extension Point“-Mechanismus von Eclipse

Manfred Hennig, Heiko Seeberger

Bei der Entwicklung eines modernen Rich Clients erscheinen die Assoziationen Aggregation/Komposition und Vererbung oft als nicht ausreichend, da sich die Abhängigkeitsrichtung oder die Kopplung als ungeeignet für das zu bearbeitende Problem herausstellen. In vielen Fällen kann hier die Anwendung des „Extension Point“-Konzepts von Eclipse helfen. Aber Achtung, ein unbedachter Einsatz dieses Konzepts kann schnell zu schwer verständlichem Programmcode führen.

Eclipse ist bekanntermaßen mehr als nur eine IDE (integrated development environment) für die Java-Entwicklung. Seit der ersten Version ist Eclipse streng komponentenorientiert aufgebaut und stellt damit eine offene Service-Plattform dar, die durch sogenannte Plug-Ins beliebig ergänzt werden kann. Den Plug-Ins kann wiederum über Erweiterungspunkte, sogenannte Extension Points, Funktionalität hinzugefügt werden. Auch die Kopplung der Eclipse-Plug-Ins untereinander beruht u. a. auf dem „Extension Point“-Mechanismus.

Allerdings war Eclipse vor Version 3 noch nicht optimal als Plattform für einen Rich Client geeignet, da nur die GUI der IDE erweitert werden konnte, was man der Applikation ansah. Mit Version 3.0 änderte sich das. Die proprietäre Eclipse Runtime wurde auf das OSGi (Open Services Gateway Initiative)-Konzept durch Integration des Equinox-OSGi-Frameworks umgestellt. So basieren jetzt auch die Eclipse-Plug-Ins auf den OSGi-Komponenten, den Bundles. Weiterhin wurde die Eclipse Rich Client Platform (RCP) eingeführt, die es jetzt ermöglicht, eigenständige Applikationen zu bauen, die nicht auf der Eclipse IDE aufsetzen und die trotz-

dem Eclipse-Konzepten wie Views, Perspektiven, aber auch den „Extension Point“-Mechanismus nutzen können. Eine solche Nutzung erfolgt i.d.R. über vorhandene Extension Points der entsprechenden Eclipse Plug-Ins. Aber auch für eigene Applikationen stellt die Möglichkeit, Extension Points zur Verfügung zu stellen, die von potentiellen Hosts verwendet werden, ein interessantes Leistungsmerkmal dar.

In das „Extension Point“-Konzept, das in der IDE durch das Plug-In Development Environment (PDE) mannigfaltig unterstützt wird und dessen Handhabung mit der Eclipse Version 3.2 nochmals verbessert wurde, soll nun nachfolgend anhand der Entwicklung einer kleinen Rich Client-Applikation eingeführt werden.

Warum Extension Points?

Bevor wir mit einem konkreten Beispiel beginnen, machen wir noch einen kleinen theoretischen Exkurs, um die Extension Points gegenüber anderen Techniken zu positionieren. Dabei werden wir sehen, dass bei der Entwicklung einer Eclipse-Applikation unter Einhaltung des Komponenten-Ansatzes die gängigen objektorientierten Mittel wie Vererbung, Aggregation und Komposition nicht ausreichen. Nachfolgend seien a und b Plug-Ins und A und B Klassen innerhalb der Plug-Ins. Das Plug-In a soll eine eigenständige Basiskomponente sein, die durch b ergänzt wird, b ist also von a abhängig.

Vererbung

In einem ersten Ansatz könnte man die beschriebene Beziehung über eine Vererbung (s. Abb. 1) implementieren, in der B die Spezialisierung von A darstellt. Plug-In b nutzt damit a, was dann auch der Abhängigkeitsrichtung entspricht. Plug-In b ist damit die aktive und a die passive Komponente.

Ein Komponenten-Szenario, in dem a als eigenständige Komponente läuft und bei Bedarf durch Plug-In b funktional ergänzt werden kann, ist aufgrund der Nutzungsrichtung nicht möglich. Weiterhin bilden A und B (und damit auch a und b) durch die Vererbung eine Einheit. Der Komponentengedanke der „losen“ Kopplung ist damit korrumpiert. Allgemein gilt, dass eine Vererbungsbeziehung Plug-Ins stark koppelt und grundsätzlich vermieden werden sollte. Die Definition eines Interfaces in Plug-In a hingegen, das in Plug-In b implementiert wird, ist keine Vererbung im Sinne von Einheit, sondern eine Typisierung und soll verwendet werden!

Aggregation/Komposition

Bei der Aggregation/Komposition ist das Problem analog. Plug-In b nutzt Plug-In a. Bis auf die Tatsache, dass die Kopplung schon etwas loser ist, könnte auch hier aufgrund der Nutzungsrichtung a nicht durch b ergänzt werden.

Es gilt also, das Problem zu lösen, die Nutzungsrichtung umzudrehen, ohne die Abhängigkeitsrichtung zu ändern.

„Extension Point“-Mechanismus

Beim „Extension Point“-Mechanismus sieht Plug-In a eine Registrierung vor, an der sich Instanzen vormerken lassen können, um Plug-In a zu ergänzen. Die registrierende Stelle nennt man Extension Point, eine Registrierung Erweiterung bzw. Extension. Wird in A eine Stelle erreicht, die ergänzt werden kann/soll, kann bei a nachgefragt werden, ob sich jemand registriert hat. Wenn ja, wird diese Funktionalität ausgeführt. Es ergibt sich das in Abbildung 2 dargestellte Szenario.

Die Nutzungsrichtung ist nun also entgegengesetzt, sodass B von A gerufen wird, ohne dass sich die Abhängigkeitsrichtung geändert hat. Damit A Funktionalität von B ausführen kann, gibt es eine Zugriffsvereinbarung, i.d.R. ein Interface. Wie wir später sehen werden, sind aber auch andere Mechanismen möglich. Entscheidend ist die Aufrufrichtung gemäß dem Hollywood-Prinzip „We call you, don't call us“. Plug-In a lässt sich damit beliebig erweitern, ohne dass a verändert werden muss. Im Eclipse-Sprachgebrauch nennt man die den Extension Point zur

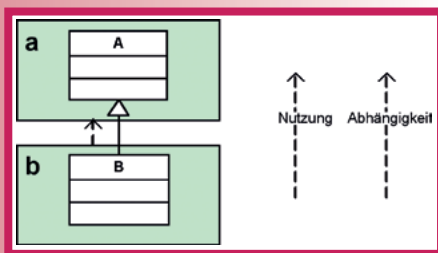


Abb. 1: Nutzungs- und Abhängigkeitsrichtung bei der Vererbung

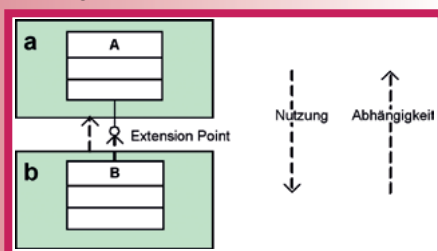


Abb. 2: Nutzungs- und Abhängigkeitsrichtung beim „Extension Point“-Mechanismus



Verfügung stellende Einheit „Enabler“ (hier A) und die den Extension Point erweiternde Einheit „Extender“ (hier B).

Der Umgang mit Extension Points und Extensions erschließt sich allerdings nicht immer direkt. Wie soll z. B. mit dem Fall eines korrupten Extenders umgegangen werden? Wann lade ich am besten eine Extension? Um diese und viele Fragen mehr zu beantworten und um einen einheitlichen Umgang mit den Eclipse Extension Points zu erreichen, wurden von den Entwicklern von Eclipse „Best Practices“ als Design-Regeln aufgestellt, an die man sich halten sollte. So sagt die „Good Fences“-Regel, dass sich der Enabler vor korrupten Extendern selber schützen muss, und empfiehlt, den korrupten Extender aus der Liste der Extensions zu entfernen. Die „Lazy Loading“-Regel gibt vor, Extensions erst dann zu laden, wenn sie benötigt werden. Wichtige Regeln [GaBe03] und ihre Aussagen sind in Tabelle 1 aufgelistet.

Zusammenfassung der Theorie

Die bisherige Theorie soll allerdings nicht dazu verleiten, jedes Problem über einen Extension Point zu lösen. Man muss sich klar sein, dass der Code zur Implementierung eines Enablers relativ umfangreich und eher schwer lesbar ist. Die Suchmöglichkeiten der Eclipse IDE finden zwar alle Enabler/Extender-Beziehungen, aber trotzdem ist ein Extender im Code definitiv nicht auf den ersten Blick als ein solcher erkennbar, da jede Klasse ein potentieller Extender ist. Ebenso kann jedes Interface die Vereinbarung zwischen Enabler und Extender darstellen, ohne dass es sich explizit als solches ausweist.

Die Lösung liegt wie so oft in der richtigen Mischung der Konzepte. Beim Design einer Anwendung sollte man darauf achten, dass die Extension Points nicht über Hand nehmen, gemäß dem Motto „weniger ist mehr“. Extension Points koppeln Plug-Ins aneinander und sollten nicht (ausschließlich) für die Nutzung innerhalb des definierenden Plug-Ins gebaut sein. Sie sollten vom definierenden Plug-In nur genutzt werden, um die „Fair Play“-Regel zu erfüllen.

Ganz wichtig ist die Frage nach der Abhängigkeitsrichtung. Nur wenn aus Sicht der Basis erweiternde Funktionalität zur Verfügung gestellt werden soll, kann ein Extension Point zum Einsatz kommen. Muss hingegen Basisfunktionalität genutzt

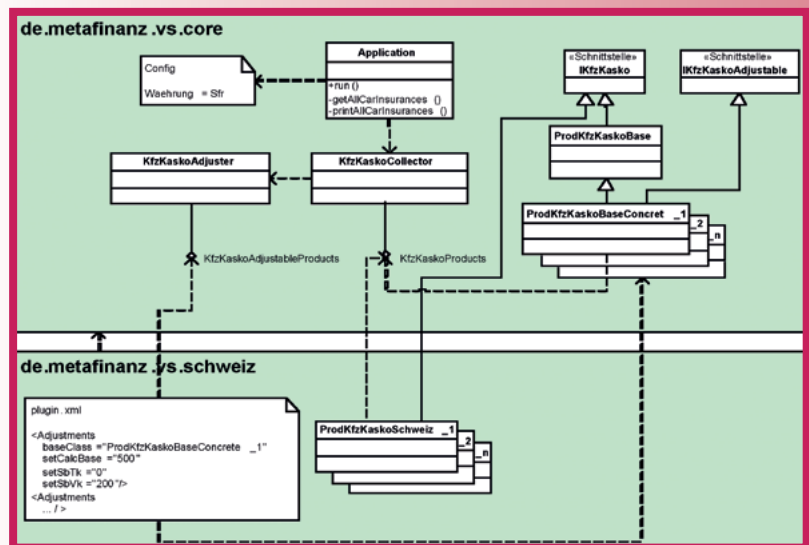


Abb. 3: Statisches Klassendiagramm zur Verwaltung von Kfz-Kasko-Produkten

Regel	Aussage
Contribution	Alles ist ein Beitrag, es gibt keinen Kern.
Lazy Loading	Beiträge sind nur zu laden, wenn sie benötigt werden.
Sharing	Hinzufügen, nicht ersetzen.
Conformance	Beiträge bedienen die erwarteten Schnittstellen korrekt.
Relevance	Trage nur etwas bei, wenn es erfolgreich arbeitet.
Safe Platform	Wenn man einen Extension Point zur Verfügung stellt, muss dieser gegen Fehlverhalten des Extenders geschützt sein.
Invitation	Ermögliche anderen, deinen Beitrag zu erweitern.
Fair Play	Alle Nutzer eines Extension Point sind bei der Nutzung gleichberechtigt, inklusive dessen Autors.
Explicit Extension	Beschreibe explizit, wo die Plattform erweitert werden kann (public ist nicht published).
Diversity	Extension Points unterstützen viele Extensions.
Good Fences	Wenn die Kontrolle an die Extension geht, dann muss der eigene Code geschützt sein. Verursacht die Extension einen Fehler, dann entlade sie und rufe sie nicht mehr auf.
Stability	Ist ein Extension Point zur Verfügung gestellt, soll er nicht mehr geändert werden (bzgl. des APIs).
Defensive API	Veröffentliche nur dann ein API, wenn du sicher bist, dass es allen Anforderungen Stand hält.
Responsibility	Identifiziere dein Plug-In klar als den Problemverursacher.

Tabelle 1: Wichtige Design-Regeln für die Entwicklung mit Eclipse RCP



werden, sind andere Mechanismen geeigneter. Prinzipiell ist dann eine Aggregation der Vererbung vorzuziehen. Hier sei auf das „Extension Object“-Muster hingewiesen, das ebenso wie die Vererbung die Möglichkeit bietet, das API einer bestehenden Klasse zu erweitern. Die Eclipse-Plattform unterstützt die Anwendung des „Extension Object“-Musters direkt [GaBe289].

Grundsätzlich sollte die Entscheidung, einen Extension Point einzusetzen, immer allen Einsatzregeln genügen. Kommt es zu Verstößen, sollte das Design auf jeden Fall nochmals überdacht werden. Man muss sich immer bewusst sein, was in welcher Situation wichtig ist: einfach lesbarer Code, absolute Dynamik beim Laden/Entladen von Plug-Ins, möglichst hohe Entkopplung, Aufruflichkeit, Beschränkungen der Sichtbarkeit (Kapselung), ...

Idealerweise resultiert ein Kern, der über Anknüpfungspunkte verfügt, und eine erweiternde Schale. Der Kern sollte auch bei Änderungen die Anknüpfungspunkte wahren und ohne erweiternde Schale lebensfähig sein. Durch Definition der Sichtbarkeit sollte der Kern seine Nutzung so einschränken, dass sein API möglichst stabil gehalten werden kann. Man beachte, dass sich nicht jeder an die „Explicit Extension“-Regel hält und viele danach handeln, dass alles was nach außen hin sichtbar ist, auch als API genutzt werden kann.

Ein Anwendungsbeispiel

Führen wir uns folgenden Fall vor Augen. Eine europaweit agierende Versicherungsgesellschaft hat eine Auswahl von Basisversicherungsprodukten, die konzernweit eingesetzt werden. Die Verwaltung aller Produkte kann im Konzern einheitlich gehandhabt werden. Die einzelnen Versicherungsgesellschaften sollen in der Lage sein, die Konzernprodukte mit eigenen Produkten zu ergänzen. Nachfolgend sollen nur Kfz-Kasko-Produkte betrachtet werden und die wiederum sehr (sehr!!) vereinfacht:

- ▼ Die Kfz-Kasko-Produkte werden durch ihre Teil-/Vollkasko-Kombination (Tk/Vk) charakterisiert.
- ▼ Ein Kfz-Kasko-Produkt kann auf ein Kraftfahrzeug angewendet werden, das über seine Risikoklasse repräsentiert wird.
- ▼ Aufgrund der Risikoklasse und der Teilkasko/Vollkasko-Kombination kann ein 100%er Jahresbeitrag berechnet werden.

Folgende Randbedingungen soll die Applikation berücksichtigen:

- ▼ Unterschiedliche Währungen erfordern unterschiedliche Währungseinheiten.
- ▼ Aufgrund unterschiedlicher Währungen müssen die Basisprodukte im Preis und in den Teilkasko/Vollkasko-Kombinationen angepasst werden können, damit z. B. die Deutsche und die Schweizer Gesellschaft die Konzernsoftware gleichermaßen nutzen kann.

Im konkreten Beispiel erstellen wir ein Kern-Plug-In, das auf Euro basiert. Das Kern-Plug-In soll dann von einem Schweizer Plug-In erweitert werden.

Design

Das Klassendiagramm in Abbildung 3 stellt eine Lösung für das beschriebene Szenario dar und soll uns als Diskussionsgrundlage dienen.

Hier gibt es die Extension Points `KfzKaskoProducts` und `KfzKaskoAdjustableProducts`. `KfzKaskoProducts` registriert Kfz-Kasko-Produkte, die vom `KfzKaskoCollector` abgeholt werden. `KfzKaskoAdjustableProducts` registriert Änderungen, die an den Basisprodukten vorgenommen werden müssen. Der `KfzKaskoAdjuster` holt die neuen Werte ab und passt mit ihnen die über `KfzKaskoAdjustableProducts` festgelegten Basisprodukte an. Sobald der `KfzKaskoCollector`

for alle Produkte abgeholt hat, kann er den `KfzKaskoAdjuster` mit Anpassungen beauftragen, insofern Anpassungen notwendig sind. Jedes zu verwaltende Kasko-Produkt muss hierbei das Interface `IKfzKasko` implementieren. Wenn ein Kasko-Produkt zusätzlich noch anpassbar sein soll, muss es auch das Interface `IKfzKaskoAdjustable` implementieren.

Es stellt sich nun die Frage, ob die beiden Extension Points als Design-Mittel berechtigt sind und warum für die Abfrage der Währungseinheit eine Property-Datei verwendet wird und kein Extension Point. Dazu betrachten wir die Design-Regeln zusammen mit der dargestellten Theorie.

Argumente für die Extension Points `KfzKaskoProducts` und `KfzKaskoAdjustableProducts`:

- ▼ Ein Kfz-Kasko-Produkt und seine Anpassbarkeit sind zentrale Forderungen an die Software und damit an die Flexibilität der Architektur (-> zentrale Aspekte und damit „nicht zu viel“).
 - ▼ Die Extension Points werden von einem Plug-In genutzt, das die Extension Points nicht selbst definiert. Die durch das Einsatzszenario geforderte Abhängigkeitsrichtung und die gewünschte Flexibilität an eine Komponentenarchitektur kann gut über Extension Points erreicht werden (-> Kopplung von Plug-Ins, Abhängigkeitsrichtung).
 - ▼ Die Nutzung des Extension Point `KfzKaskoProducts` innerhalb des Kern-Plug-Ins entspricht der „Fair Play“-Regel.
 - ▼ Die Basisversicherungsprodukte können per Extension Point `KfzKaskoAdjustableProducts` angepasst werden (-> Invitation-Regel).
 - ▼ Es können beliebig viele weitere Produkte von Extendern hinzugefügt werden (-> Diversity-, Sharing-Regel).
 - ▼ Alles ist ein Beitrag und damit jedes weitere Kfz-Kasko-Produkt auch (Contribution-Regel).
 - ▼ Alle anderen Regeln sprechen eher die Art der Implementierung an und spielen damit erst bei der späteren Umsetzung eine Rolle.
- Argumente gegen den Einsatz eines Extension Point, um die Währungseinheit zu ermitteln:
- ▼ In der gesamten Applikation wird genau eine Währungseinheit verwendet (in Deutschland sind das Euro, in der Schweiz Sfr und in England Pfund). Es gäbe also exakt nur eine Extension (Verletzung der Diversity-Regel).
 - ▼ Verwendung von Extension Points auch für Kleinstaspekte macht die Anwendung unübersichtlich (-> zu viele Extension Points).
 - ▼ Der Zugriff auf Property ist übersichtlicher und erfordert weniger Code. Die höhere Flexibilität eines Extension Point bringt hier keinen Nutzen.

Vorgehen bei der konkreten Implementierung

Damit sind alle Vorbereitungen getroffen und wir können mit der konkreten Implementierung beginnen.

Schritt 1

Als erstes erstellen wir über die Eclipse IDE unser Kern-Plug-In. Dazu wählen wir den Projekttyp „Plug-In Project“. Beim Einrichten des Projekts können wir die Default-Einstellungen des Wizards weitgehend beibehalten. Wichtig sind allerdings folgende Aktivierungen/Deaktivierungen (s. Abb. 4), denn wir wollen der Einfachheit halber nur eine eigene Rich Client-Anwendung ohne GUI bauen:

- ▼ This plug-in will make contributions to the UI (No)
- ▼ Would you like to create a rich client application? (Yes)

Am Schluss verwenden wir das vom Wizard vorgeschlagene „Headless Hello RCP“-Template. Die Aktivierung zum Bau-



en eines Rich Clients und die Verwendung des „Headless Hello RCP“-Templates richten für uns gleich die Klasse `Application` mit der Methode `run()` als Extender für den Extension Point `org.eclipse.core.runtime.applications` ein. Beim Start des Rich Clients wird dann die Methode `run()` aufgerufen, analog wie bei einer normalen Java-Anwendung die Methode `main()`.

Bei der Erstellung unseres Schweizer Plug-Ins verfahren wir analog, nur dass wir keinen weiteren Einsprungpunkt benötigen. Bei der Rich Client-Applikation wählen wir also „No“ und deaktivieren auch alle Template-Vorschläge.

Schritt 2

In die vom Wizard generierte Klasse `Application` implementieren wir innerhalb der Methode `run()` unsere Steuerlogik (s. Listing 1).

Schritt 3

Betrachten wir als nächstes die beiden Extension Points. Ein Extension Point stellt sich als ein XML-Schema mit Vereinbarungen dar, die von einer Extension erfüllt werden müssen. Eine typische Vereinbarung ist ein Interface, das der Extender implementiert, auf den die Extension verweist. In unserem Fall brau-

```
public Object run(final Object args) throws Exception {
    // Waehrungseinheit einlesen
    ...
    // Produktaufstellung für ein Auto der Risikoklasse 19
    final int risikoklasse = 19;
    final List<IKfzKasko> l = new ArrayList<IKfzKasko>();
    if (getAllCarInsurances(l)) {
        printAllCarInsurances(l); // Produkte auflisten
    }
    return IPlatformRunnable.EXIT_OK;
}
```

Listing 1: Steuerlogik zur Verwaltung der Kfz-Kasko-Produkte

```
public interface IKfzKasko {
    public int getSbTk();
    public int getSbVk();
    public int getCalcBase();
    public double calculate(int rk);
}

public interface IKfzKaskoAdjustable {
    public void setSbTk(int sbTk);
    public void setSbVk(int sbVk);
    public void setCalcBase(int calcBase);
}
```

Listing 2: Interfaces der unterschiedlichen Kasko-Produkte

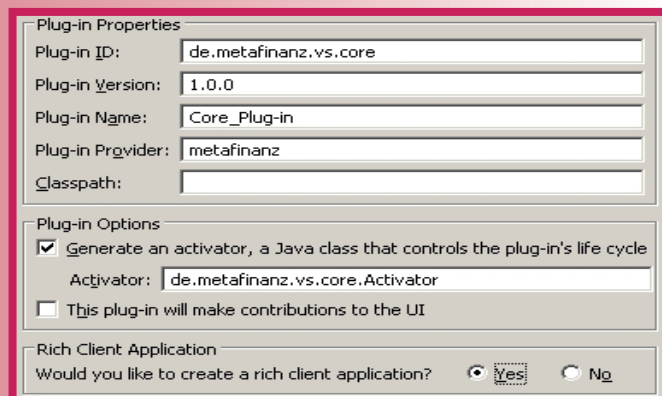


Abb. 4: Anlegen eines Plug-In-Projekts

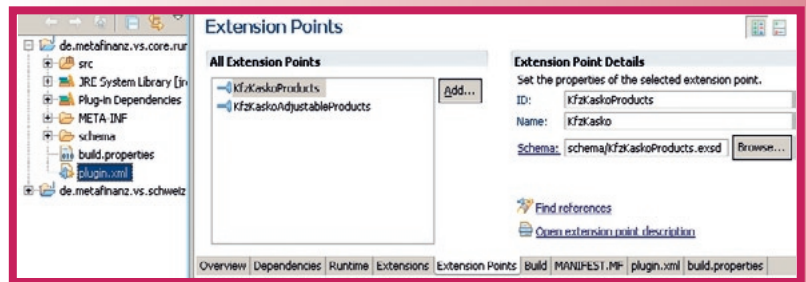


Abb. 5: Extension Points im „Plug-in Manifest“-Editor

chen wir also ein Interface für ein Kfz-Kasko-Produkt und ein Interface für ein anpassbares Kfz-Kasko-Produkt (s. Listing 2).

Schritt 4

Mit diesen Informationen können wir das Einrichten der Extension Points angehen.

Durch Doppelklick auf die Datei „plugin.xml“ (s. Abb. 5) wird diese im „Plug-in Manifest“-Editor geöffnet. Dort aktivieren wir die Auswahl „Extension Points“. Mit „Add“ können wir einen neuen Extension Point im gewählten Plug-In hinzufügen. Als Basisinformationen müssen ID und Name des Extension Point und Name der Schemadatei mit den Extension Point-Vereinbarungen angegeben werden. Danach verzweigen wir in die Schemadatei, um die Vereinbarungen zu formulieren, wofür uns wiederum ein grafischer Editor zur Verfügung steht.

Schritt 5

In Abbildung 6 fügen wir das Element `KfzKasko` hinzu und erweitern es mit den Attributen `class` und `name`. Als Detailinfo hinterlegen wir das Interface `de.metafinanz.vs.core.app.IKfzKasko` unserer Kfz-Kasko-Produkte. Beim Attribut `name` kann eine sprechende Bezeichnung für den Extension Point angegeben werden, der dann bei der Zuordnung Extension/Extension Point automatisch angezeigt wird. Das Element `KfzKasko` muss dann noch als Sequenz unter das Element `extension` positioniert werden. Hier können wir dann auch angeben, dass beliebig viele oder gar keine Kasko-Extensions erlaubt sind. Bei den Attributen, die wir unter `KfzKasko` definieren, sprechen wir von den „Configuration Elements“. Unsere Konfiguration wird dann in der Schemadatei `KfzKaskoProducts.exsd` hinterlegt. Die den Extension Point identifizierenden Daten werden in die `plugin.xml` (s. Listing 3) geschrieben.

Bei unserem zweiten Extension Point `KfzKaskoAdjustableExpId` verfahren wir analog. Anstelle des Elements `KfzKasko` führen wir ein Element `Adjustable` ein und hinterlegen dort die String-Attribute `setSbTk`, `setSbVk` und `setCalcBase`. Weiterhin benötigen wir dort noch ein Attribut `baseClass` mit dem Interface `IKfzKaskoAdjustable`. Wir verwenden hier den Typ `java`, um später beim Erstellen der Extension die vor Fehlern schützende Möglichkeit zu nutzen, nach der anzupassenden Kasko-Klasse browsen zu können. Eine String-Repräsentation wäre ausreichend, da hier nie eine Instanz angefordert werden muss. Damit sehen wir auch, dass nicht unbedingt immer ein Interface oder eine Klasse die Vereinbarung bilden muss. Strings oder auch Ressourcen sind ebenfalls möglich.

Schritt 6

Nachdem nun beide Extension Points definiert sind, können wir unseren Enabler (s. Listing 4) programmieren. Die Fehlerbehandlung in den folgenden Code-Snippets ist allerdings nur rudimentär und genügt den Regeln „Save Platform“ und „Good Fences“ nicht.

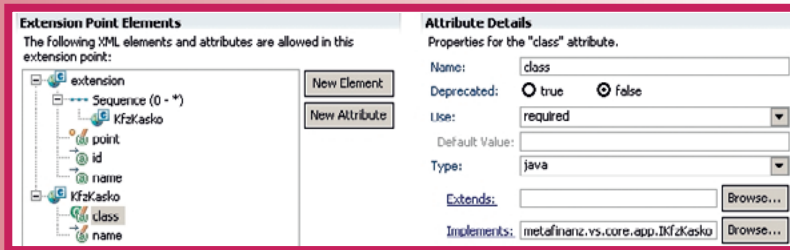


Abb. 6: Grafischer XML-Editor für die „Extension Point“-Definition

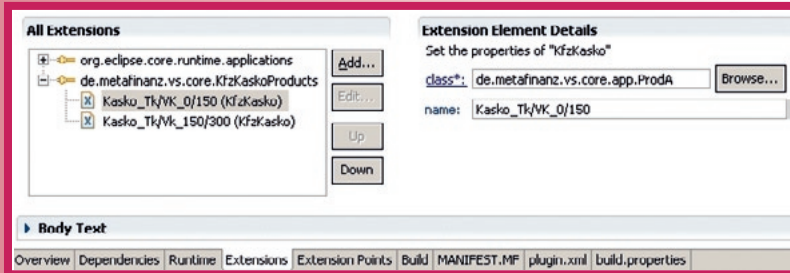


Abb. 7: Extension de.metafinanz.vs.core.KfzKaskoProducts im „Plug-in Manifest“-Editor

```
<plugin>
<extension-point
id="KfzKaskoProducts" name="KfzKasko"
schema="schema/KfzKaskoProducts.exsd"/>
</plugin>
```

Listing 3: Extension Point „de.metafinanz.vs.core.KfzKaskoProducts“ in der Plugin.xml

```
private static final String KASKO_ID =
"de.metafinanz.vs.core.KfzKaskoProducts";
private static final String ADJ_ID =
"de.metafinanz.vs.core.KfzKaskoAdjustableProducts";

public boolean getAllCarInsurances(final List<IKfzKasko>
kaskoList) {
try {
final IConfigurationElement[] kasko =
Platform.getExtensionRegistry().
getConfigurationsFor(KFZ_KASKO_EXP_ID);
final IConfigurationElement[] kaskoAdj =
Platform.getExtensionRegistry().
getConfigurationsFor(
KFZ_KASKO_ADJUST_EXP_ID);
// Diversity-Regel: beliebig viele Extensions erlauben
for (IConfigurationElement e : kasko) {
final Object o = e.createExecutableExtension("class");
if (o instanceof IKfzKasko) {
final IKfzKasko ksk = (IKfzKasko)o;
if (kaskoAdj != null &&
ksk instanceof IKfzKaskoAdjustable) {
// Invitation-Regel: Anpassungen durchführen lassen
adjust((IKfzKaskoAdjustable)ksk, kaskoAdj);
}
// Sharing-Regel: Hinzufügen und nicht ersetzen
kaskoList.add(ksk);
}
}
} catch (final Exception ex) {
return false; // Fehlerbehandlung ...
// (rudimentäre "Good Fences"-Regel)
}
return true;
}
```

Listing 4: Enabler zur Ermittlung potentieller Kasko-Produkte

Die Methode `getAllCarInsurances()` befindet sich nun in der Rolle eines Enablers. Im ersten Schritt holt man von der `org.eclipse.core.runtime.platform` über die Extension Registry alle „Configuration Elements“ zum gewünschten Extension Point ab (Diversity-Regel). Man beachte, dass der Extension Point über den Plug-In-Namen und seine ID durch Punkt getrennt qualifiziert werden muss. Danach arbeiten wir unsere im Schritt 5 definierten „Configuration Elements“ aus. Die Auswertung ist hier recht einfach, da nur das Attribut `class` definiert wurde, über das eine Implementierung des Interfaces `IKfzKasko` durch die Extension vereinbart ist. Diese Vereinbarung stellen wir über eine `instanceof`-Abfrage sicher. Wenn die Instanz auch noch das Interface `IKfzKaskoAdjustable` implementiert, dann müssen wir auch noch per `adjust()` (s. Listing 5) eventuelle Anpassungen vornehmen lassen (Invitation-Regel). Danach können wir die Extension, also eine `KfzKasko`-Implementierung, unserer Kasko-Liste hinzufügen (Sharing-Regel), was dann auch schon den Großteil unseres Business Case ausmacht.

Schritt 7

Listing 6 zeigt eine konkrete Kasko-Basisprodukt-Implementierung. Die Implementierung aus dem Schweizer Plug-In sieht dann sehr ähnlich aus, wobei `IKfzKaskoAdjustable` nicht implementiert werden muss und natürlich `calculate()`, das große Geheimnis eines jeden Versicherers, anders aussehen wird.

Schritt 8

Damit können wir `ProdA` am Extension Point `KfzKaskoProducts` mit Hilfe des „Plug-in Manifest“-Editors gemäß „Conformance“-Regel registrieren und erfüllen damit weiterhin die „Fair Play“- und die „Contribution“-Regel.

Mit `Add` wählen wir den zu erweiternden Extension Point `KfzKaskoProducts` aus und fügen ihn in die Liste der Extensions ein. Mit rechtem Maus-Klick auf die hinzugefügte Extension und „New“ können wir `KfzKasko`, also unser Erweiterungsdetail, aus-

wählen und setzen. Die Details beschränken sich auf unser `ProdA` und optional einen sprechenden Namen.

Wenn wir jetzt das Programm starten (Start Konfiguration: „Eclipse Application“/„run an application“: „de.metafinanz.vs.core.application“), werden die von `de.metafinanz.vs.core` selbst registrierten Produkte aufgelistet:

```
Produktaufstellung für ein Kfz in Risikoklasse 19
Kasko 0/150: 1.320,90 Euro
Kasko 150/300: 1.153,11 Euro
```

Schritt 9

Bevor wir die Schweizer Produkte und Extensions hinzufügen, müssen wir noch die Sichtbarkeit der beiden Plug-Ins explizit regeln. Hierfür wählen wir in `de.metafinanz.vs.core` im „Plug-in Manifest“-Editor den Punkt „Runtime“ aus und geben das Package `de.metafinanz.vs.core.app` als Export an. In `de.metafinanz.vs.schweiz` müssen wir im „Plug-in Manifest“-Editor über den Punkt „Dependencies“ das Plug-In `de.metafinanz.vs.core` als „Required“ auswählen, oder feingranularer das Package `de.metafinanz.vs.core.app` importieren.

Die Extension für ein Basisprodukt „ProdB“ entspricht der „Conformance“-Regel und ist natürlich auch ein Beitrag (Contribution-Regel) und könnte dann wie in Abb. 8 dargestellt gestaltet werden. In der `Plugin.xml` werden die Extensions dann wie in Listing 7 eingetragen.

Wenn wir jetzt die Applikation starten, erhalten wir die beiden Basisprodukte in angepasster Form und das Schweizerer Produkt:

```
Produktaufstellung für ein Kfz in Risikoklasse 19
Kasko 0/200: 2.142,00 Sfr
Kasko 200/500: 1.666,00 Sfr
Kasko 300/1000: 1.507,50 Sfr
```



Damit sind wir am Ende unserer Einführung angelangt. Bei den gezeigten Features des PDE haben wir uns auf das Notwendigste beschränkt, um einen kleinen Rich Client zu erstellen. Wir haben also nur die sprichwörtliche Spitze des Eisbergs dargestellt, womit wir Ihnen noch unendlich viel Spielraum für eine weitere Vertiefung des Themas lassen ☺.

Literatur und Links

- [Blo01] J. Bloch, Effective Java, Addison-Wesley Professional, 2001, S.87
 [Daum06a] B. Daum, Java-Entwicklung mit Eclipse 3.2., dpunkt.verlag, 2006
 [Daum06b] B. Daum, Rich Client Platforms und Rich Internet Applications, in: JavaSPEKTRUM, 6/2006
 [GaBe03] E. Gamma, K. Beck, Contributing to Eclipse, Addison-Wesley Professional, 2003
 [GaBe289] E. Gamma, K. Beck, [GaBe03], S.289 ff
 [LiV004] M. Lippert, M. Völter, Rich Clients mit Eclipse 3, 2004, <http://www.voelter.de/data/articles/jmEclipseRCP.pdf>

```
private static void adjust(final IKfzKaskoAdjustable kasko,
    final IConfigurationElement[] adj) {
    final IConfigurationElement e = getAdjustmentsOf(kasko, adj);
    if (e != null) {
        kasko.setSbTk(Integer.valueOf(e.getAttribute("setSbTk")));
        kasko.setSbVtk(Integer.valueOf(e.getAttribute("setSbVtk")));
        kasko.setCalcBase(Integer.valueOf(e.getAttribute("setCalcBase")));
    }
}

private static IConfigurationElement getAdjustmentsOf(
    final IKfzKaskoAdjustable kasko,
    final IConfigurationElement[] adj) {
    for (IConfigurationElement ce : adj) {
        final String cl = ce.getAttribute("baseClass");
        if (kasko.getClass().getName().equals(cl)) {
            return ce;
        }
    }
    return null;
}
```

Listing 5: Enabler zur Anpassung von Kasko-Produkten

```
// Skeletal Implementation [Blo01] aller Kasko Basisprodukte
abstract public class ProdKfzKaskoBase implements IKfzKasko {
    abstract public int getSbTk();
    abstract public int getSbVtk();
    abstract public int getCalcBase();
    public double calculate(int rk) {
        // result = calculate from getCalcBase(), getSbTk(), getSbVtk(), rk
        return result;
    }
}

// Konkrete Implementierung eines Kasko Basisproduktes
public class ProdA extends ProdKfzKaskoBase
    implements IKfzKaskoAdjustable {
    private int sbTk = 0;
    private int sbVtk = 150;
    private int calcBase = 300;
    public int getSbTk() { return sbTk; }
    public int getSbVtk() { return sbVtk; }
    public int getCalcBase() { return calcBase; }
    public void setCalcBase(int calcBase) { this.calcBase = calcBase; }
    public void setSbTk(int sbTk) { this.sbTk = sbTk; }
    public void setSbVtk(int sbVtk) { this.sbVtk = sbVtk; }
}
```

Listing 6: Konkretes Kasko-Basisprodukt und seine „skeletal implementation“

```
<plugin>
<extension
    id="KfzKaskoAdjustableProducts"
    name="KfzKaskoAdjust"
    point="de.metafinanz.vs.core.KfzKaskoAdjustableProducts">
<Adjustments
    baseClass="de.metafinanz.vs.core.app.ProdA"
    name="adust_Tk/Vk/Base_to_0/200/500"
    setCalcBase="500"
    setSbTk="0"
    setSbVtk="200"/>
<Adjustments
    baseClass="de.metafinanz.vs.core.app.ProdB"
    name="adust_Tk/Vk/Base_to_200/500/500"
    setCalcBase="500"
    setSbTk="200"
    setSbVtk="500"/>
</extension>
<extension
    id="KfzKaskoProducts"
    name="KfzKasko"
    point="de.metafinanz.vs.core.KfzKaskoProducts">
<KfzKasko
    class="de.metafinanz.vs.schweiz.ProdCH"
    name="Kasko_Tk/Vk_300/1000"/>
</extension>
</plugin>
```

Listing 7: Extension zum Extension Point de.metafinanz.vs.core.KfzKaskoAdjustableProducts in der Plugin.xml

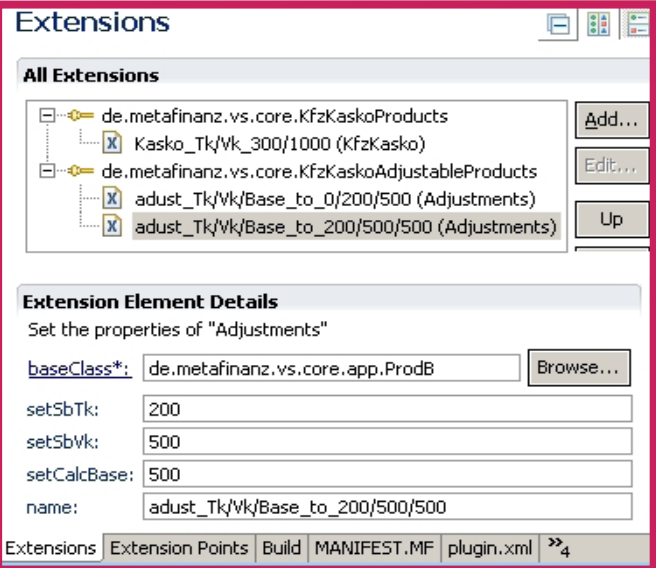


Abb. 8: Extension de.metafinanz.vs.core.KfzKaskoAdjustableProducts im „Plug-in Manifest“-Editor



Manfred Hennig ist Angestellter der metafinanz GmbH und arbeitet als Entwickler, Berater und Projektleiter schwerpunktmäßig in den Versicherungsbereichen Flottenmanagement und Tarifierung. Sein spezielles Interesse gilt derzeit der Entwicklung mit Eclipse RCP.
E-Mail: manfred.hennig@metafinanz.de.



Heiko Seeberger leitet die Market Unit Enterprise Architecture der metafinanz GmbH. Er erstellt seit etwa zehn Jahren Enterprise Applications mit Java, wobei sein aktueller Fokus auf Eclipse und AspectJ liegt. Er ist Committer der Open-Source-Projekte AJEER und ContractJ.
E-Mail: heiko.seeberger@metafinanz.de.