Unter der Haube, Teil 4: Mit Extension Points flexible Architekturen realisieren

Von Steckdosen und Steckern

VON MARTIN LIPPERT

Die Eclipse-Plattform besticht unter anderem dadurch, dass sie extrem flexibel erweiterbar ist. Jeder kann Funktionen an nahezu jeder beliebigen Stelle hinzufügen. Trotzdem vermittelt die Anwendung an der Oberfläche ein einheitliches und hochgradig integriertes Bild. In diesem Teil der „Unter der Haube"-Reihe betrachten wir den Erweiterungsmechanismus von Eclipse und diskutieren, wie sich dieser Mechanismus auch außerhalb von der Eclipse-IDE oder Rich-Client-Anwendungen nutzen lässt, um große Systeme flexibel aus Komponenten zusammenzustecken.

In jedem Buch und in jeder Einführung zum Thema Eclipse findet sich ein Abschnitt, der das Konzept von Plug-ins erläutert. Früher verstand man unter diesem Plug-in-Mechanismus sowohl die Modularisierungstechnologie (also dass, was heute die OSGi-Runtime realisiert) als auch den Extension-Point-Mechanismus – jene Technologie, mit der man beispielsweise die Eclipse-IDE um eigene Views oder Actions erweitern kann. Solche Erweiterungen lassen sich einfach innerhalb von separaten Plug-ins implementieren und dem System hinzufügen, ohne das eigentliche System selbst zu verändern, globale Konfigurationsdateien anzupassen oder Ähnliches. Dieser Mechanismus hat sich mittlerweile tausendfach bewährt (zählt man alleine nur die Contributions der anderen Eclipse-Projekte).

Besonders interessant dabei sind zwei Dinge: Zum einen stellt sich das System an der Oberfläche wie aus einem Guss dar – eine Eigenschaft, die nicht selbstverständlich für Systeme ist, die so stark erweiterbar und zusammensteckbar sind. Zum anderen erlaubt der Mechanismus, dass jede Komponente selbst wiederum neue Erweiterungspunkte definieren kann. Damit werden auch neue Komponenten erweiterbar und setzen das Erweiterungskonzept konsequent fort.

Anstatt den Extension-Point-Mechanismus selbst im Detail einzuführen und seine Verwendung auf der technischen Ebene im Detail zu erläutern, widmet sich dieser Artikel den grundsätzlichen Eigenschaften des Mechanismus und zeigt, wie sich auch jenseits der vorhandenen Eclipse-Extension-Points oder gar jenseits der gesamten Rich Client Platform flexible Architekturen mit dem Extension-Point-Mechanismus realisieren lassen und in welchem Verhältnis der Extension-Point-Mechanismus zu beispielsweise dem Dependency-Injection-Mechanismus steht.

Unterschiedliche Extension Points

Extension Points werden dort definiert, wo das System offen für Erweiterungen sein soll. Dabei soll in der Regel die Funktionalität des Anbieters eines Extension Point (beispielsweise die Workbench von Eclipse) um weitere Funktionen oder Inhalte erweitert werden (beispielsweise eine neue View), ohne dass der Anbieter die konkreten Erweiterungen direkt kennt. Schließlich wollen wir verhindern, dass beispielsweise die Workbench von Eclipse alle auf der Welt verfügbaren Views kennen muss, um sie gegebenenfalls anzuzeigen.

Während in klassischen monolithischen Systemen das Hauptfenster beispielsweise alle Menü-Punkte oder Views direkt kennt und aufruft, kehrt sich diese Abhängigkeit bei einer Extension-Point-basierten Architektur um: Der Lieferant einer Extension (also das Plug-in, welches die konkrete View oder eine neue Menü-Aktion implementiert) kennt den Extension Point, für den die Extension realisiert wurde (die Workbench zum Beispiel) und ist damit abhängig vom Anbieter des Extension Point.

Zur Laufzeit wird der Anbieter des Extension Point analysieren, welche

Extensions vorhanden sind, und entscheiden, wie er mit diesen Extensions umgehen möchte (beispielsweise die verfügbaren Views in einer Liste zur Auswahl anbieten und bei Bedarf erzeugen). Das bedeutet auch, dass der Anbieter eines Extension Point definiert, was er von Extensions erwartet (dies geschieht häufig in der Schema-Definition des Extension Point, gelegentlich aber auch nur in der Dokumentation). Für Views definiert die Workbench beispielsweise, dass eine View unter anderem einen Namen haben und eine Klasse mitbringen muss, die mindestens das Interface *IViewPart* implementiert.

Zwar definiert der Anbieter eines Extension Point selbst, was er von einer Extension zu „seinem" Extension Point erwartet und wie er mit diesen Extensions umgeht, jedoch kann man Extension Points in drei grobe Kategorien unterteilen:

- Content: Der Extension-Point-Mechanismus setzt nicht voraus, dass eine Extension immer Code „contributen" muss. Eine Extension kann im Grunde genommen aus beliebigen Ressourcen bestehen. Voraussetzung ist nur, dass sie dem entspricht, was der Anbieter des Extension Point erwartet. Im Falle des Hilfe-Systems von Eclipse erwartet der Extension Point beispielsweise Inhalt für die Online-Hilfe in Form von XML- oder HTML-Dateien.
- Optional & Multiple: Dies ist im Grunde genommen die klassische Form eines Extension Point, da der Anbieter des Extension Point offen für beliebig viele (und dadurch eben auch keine einzige) Extensions ist. Das ist bei den meisten UI Extension Points der Fall, beispielsweise Views, Editors, Perspectives, Actions etc. Die meisten Extension Points sind auf diese Art und Weise implementiert. Der Extension-Point-Mechanismus ist im Grunde genommen genau für diesen Einsatzzweck konzipiert. Genau betrachtet ist der Content Extension Point auch vom Typ „Optional & Multiple".
- Single & Required: In diesem Fall erwartet der Anbieter des Extension Point, dass genau eine Extension vorhanden ist und kann auch nur in diesem Fall seine eigenen Aufgaben korrekt erfüllen. Wenn mehrere Extensions zu so einem Extension Point gefunden werden, muss der Anbieter anhand von definierten Kriterien entscheiden, welche Erweiterung aktiv werden soll. Ist keine Extension vorhanden, kann der Anbieter seine eigene Aufgabe nicht erfüllen. Die Runtime von Eclipse definiert beispielsweise den Extension Point „applications". Mit ihm kann man der Runtime mitgeben, welche Eclipse-Anwendung beim Start-up hochgefahren werden soll. Über eine Command-Line-Option teilt man der Runtime mit, welche der vorhandenen „applications" beim Start-up genommen werden soll.

Extension-Points vs. Dependency Injection

Gerade das letzte Modell für Extension Points, Single & Required, legt den Verdacht nahe, dass zwischen dem Extension-Point-Mechanismus und einem gängigen Dependency-Injection-Mechanismus ein Zusammenhang bestehen könnte. In der Tat legt das „Single & Required"-Modell diese Vermutung sehr nahe, da eine solche einfache Required-Beziehung *das* Standard-

Beispiel für Dependency Injection ist. Wäre dieses Modell die einzige Variante, in der der Extension-Point-Mechanismus verwendet wird, könnte man zu Recht den Extension-Point-Mechanismus als ein Dependency-Injection-Framework bezeichnen.

Allerdings ist das „Single & Required"-Modell nicht die einzige Art und Weise, wie Extension Points verwendet werden – geschweige denn das bevorzugte Modell. Stattdessen legt es der Extension-Point-Mechanismus nahe, sich über echte *Erweiterungs*punkte eines Systems (bei der Framework-Entwicklung werden solche Elemente eines Frameworks auch als Hot-Spots bezeichnet) Gedanken zu machen und solche Erweiterungsmöglichkeiten explizit zu machen.

> Seit Eclipse 3.2 ist die Extension Registry sowohl mit anderen OSGi-Implementierungen nutzbar als auch völlig ohne.

Optionale Erweiterungen lassen sich mit den gängigen Dependency-Injection-Mechanismen natürlich auch realisieren, indem die injizierten Dependencies von der abhängigen Klasse als optionale Bestandteile interpretiert und letztendlich genauso wie optionale Extensions behandelt werden. Allerdings muss man sich dann über eine Skalierbarkeit des Mechanismus eigene Gedanken machen.

Skalierbarkeit

Was passiert beispielsweise, wenn man mehrere hundert Erweiterungen zu einem Erweiterungspunkt hat, aber nicht immer alle wirklich aktiviert werden müssen? Der Extension-Point-Mechanismus bietet hier sehr elegante und effiziente Mechanismen an, die auch für mehrere tausend Plug-ins noch ohne Probleme skalieren. Da die wesentlichen Informationen, die zur Anzeige der Extensions an der Oberfläche nötig sind, im deklarativen Anteil der Extension abgelegt werden (also in der XML-Definition der Extension), ist es zur Anzeige nicht nötig, per se alle Objekte zu laden und die beherbergenden Plug-ins zu aktivieren. Erst wenn eine ausgewählte Extension tatsächlich verwendet werden soll, können das entsprechende Plug-in aktiviert und die Klassen geladen werden. Gerade wenn das System aus vielen tausend Plug-ins mit etlichen Extensions besteht, von denen aber nur selten viele Extensions zum gleichen Extension Point gleichzeitig verwendet werden, erweist sich dieser Mechanismus als sehr effizient.

Ist der Extension Point hingegen so konzipiert, dass immer alle Extensions instanziiert werden müssen, bevor das System arbeiten kann, profitiert man nicht von dem Aufwand, den die Entwickler des Extension-Point-Mechanismus in die Skalierbarkeit der Implementierung gesteckt haben. Man sollte sich also tunlichst überlegen, ob man wirklich immer alle Extensions auch instanziieren muss oder ob man gegebenenfalls mit deklarativen Daten auskommt, bis die Extension tatsächlich verwendet wird. Damit erhält man sich die Möglichkeit, dass das System auch noch mit einer sehr großen Anzahl von Erweiterungen zurechtkommt.

Mehr als flexible UIs

Obwohl die meisten Extensionn Points aus dem Eclipse SDK sicherlich bekannt sind für UI-bezogene Erweiterungen, ist das Konzept der Extension Points keineswegs auf UI-Erweiterungen beschränkt. Ganz im Gegenteil: Extension Points lassen sich auf allen Ebenen einer Enterprise-Anwendung nutzen. Gerade die selbst definierten Extension Points spielen dabei eine entscheidende Rolle. Sie können dazu dienen, aufzählbare Enum-Typen per Extensions zu erweitern, die Persistenz-Schicht um zusätzliche Definitionen zu ergänzen oder in einer serviceorientierten Architektur neue Services bereitzustellen – um nur einige Beispiele zu nennen.

Denken wir darüber hinaus an klassische Multi-Tier-Anwendungen und den Einsatz der Eclipse-Technologie auch für den Server-Teil einer solchen Anwendung, erscheint es besonders interessant, die gleichen Plug-ins mit den gleichen Extension Points und Extensions sowohl auf dem Client als auch auf dem Server einzusetzen.

Extension Points auch ohne OSGi

Wie man sieht, kann der Extension-Point-Mechanismus auf vielen verschiedenen Ebenen eingesetzt werden und dazu dienen, eine flexible Architektur aufzubauen. Es stellt sich deshalb natürlich die Frage, ob der Mechanismus die Eclipse-Plug-in-Technologie voraussetzt und man deshalb gezwungen ist, seine Anwendung auf Basis von Plug-ins zu implementieren. Die gute Nachricht ist: Seit der Version 3.2 ist die Extension Registry (so der Komponenten-Name für den Extension-Point-Mechanismus) sowohl mit anderen OSGi-Implementierungen nutzbar als auch völlig ohne. Dazu muss der Extension Registry lediglich in einer selbst implementierten Strategie mitgeteilt werden, wie sie an die Extension-Point- und Extension-Definitionen kommt. Der einfachste Fall wäre sicherlich, die passenden XML-Dateien im Dateisystem abzulegen und der Registry zum Parsen zu überlassen. Aber auch andere Implementierungen sind durchaus denkbar.

Fazit

Der Extension-Point-Mechanismus lässt sich für große Enterprise-Anwendungen sehr gut einsetzen, um flexible Systeme zu realisieren, die auch in Zukunft noch gut erweiterbar sind. Dazu müssen die Extension Points natürlich entsprechend definiert sein. Nicht selten aber lassen sich neue Extension Points auch recht einfach aus einer bestehenden Anwendung herausfaktorisieren und erlauben es so, eine bestehende Anwendung für Erweiterungen zu öffnen. Ein rundum gelungener Mechanismus, den ich für größere Anwendungen nicht mehr vermissen möchte. Sind Sie auch auf den Geschmack gekommen?

Martin Lippert ist Senior-IT-Berater bei der akquinet agile GmbH (früher it-agile GmbH). Er arbeitet dort als Coach und Berater für agile Softwareentwicklung, Refactoring und Eclipse-Technologie und ist Committer im Eclipse Equinox Incubator-Projekt. Kontakt: martin.lippert@akquinet.de.

Links & Literatur

[1] Kent Beck, Erich Gamma: Contributing to Eclipse – Principles, Patterns, and Plugins, Addison-Wesley, 2003