Unter der Haube – Teil 3: Build to last und API-Evolution unter die Lupe genommen

Das einzig Beständige ist die Veränderung

VON MARTIN LIPPERT

Die Eclipse-Technologie ist mittlerweile zu einer Plattform für viele Entwicklungsprojekte geworden. Ein Ziel, an dem das Eclipse-Projekt viele Jahre hart gearbeitet hat und welches nicht ohne entsprechende Anstrengungen erreichbar ist. Eine Plattform zu entwickeln bedeutet mehr, als ausschließlich ein gut modularisiertes und sauber implementiertes Softwaresystem zu realisieren. Ein bedeutender Aspekt ist das API der Plattform und deren evolutionäre Weiterentwicklung. In diesem Teil der „Unter der Haube"-Reihe werfen wir einen Blick auf die Entwicklung von APIs und analysieren, wie solche APIs evolutionär weiterentwickelt werden können.

Eine Plattform zeichnet sich dadurch aus, dass sie ein API, ein Application Programming Interface, besitzt und dieses API Klienten der Plattform zur Verfügung stellt. Dieses API kann von Klienten der Plattform genutzt werden, um beispielsweise Anwendungen oder eine erweiterte Plattform zu realisieren.

Innerhalb der letzten Jahrzehnte haben wir gelernt, dass solche APIs nur sehr selten von Anfang an vollständig und fehlerfrei entworfen und implementiert werden können. Immer wieder stellen Klienten Anforderungen, die von dem API (noch) nicht erfüllt werden. APIs müssen im Laufe der Zeit verändert werden. Warum sollten auch für APIs andere Grundsätze gelten, als für „normale" objektorientierte Systeme, die wir aus gutem Grund evolutionär und inkrementell entwickeln? Und gerade für eine Plattform ist es von entscheidender Bedeutung, dass sie neuen Anforderungen der Klienten nachkommt und sich weiterentwickelt. Die Akzeptanz einer Plattform hängt davon maßgeblich ab.

Bei „normalen" objektorientierten Systemen gehen wir in der Regel von einer Closed-World Assumption aus: Wir nehmen an, dass wir beispielsweise eine Methode mit einem automatisierten Refactoring umbenennen können und die IDE alle Aufrufe dieser Methode entsprechend anpasst. Bei einer Plattform können wir von dieser Annahme nicht mehr ausgehen. Die Klienten des API, die also, dem Beispiel folgend, die Methode am API aufrufen, sind in der Regel unbekannt. Wenn innerhalb der Plattform die Methode verändert wird, kann die IDE nicht mehr alle Referenzen auf die Methode automatisch anpassen. Die Klienten müssen separat an das veränderte API adaptiert werden. Dies kann, wenn sich das API einer Plattform sehr umfangreich verändert, mit großen Aufwänden verbunden sein. Je größer diese Aufwände werden, desto unattraktiver wird die Plattform für Klienten.

Bei der Plattformentwicklung bewegen wir uns also in einem Spannungsfeld: Auf der einen Seite soll die Plattform möglichst stabile APIs anbieten, damit Klienten nicht aufwendig an neue Versionen des API angepasst werden müssen. Auf der anderen Seite erwarten wir auch von einer Plattform, dass sie sich kontinuierlich weiterentwickelt und verbessert.

Die Eclipse-Plattform ist beiden Kräften stark ausgeliefert, da in den vergangenen Jahren sehr viele Systeme entstanden sind, die auf ihr basieren. Teilweise bestehen diese Systeme aus vielen tausend Plug-ins und würden einen immensen Migrationsaufwand verursachen, sollte sich das API der Eclipse-Plattform an entscheidenden Stellen verändern. Deshalb hat sich das Eclipse-Team der Idee des „Build to Last" verschrieben.

Build to Last

In ihrer EclipseCon-Keynote von 2005 vergleichen Erich Gamma und John Wiegand die evolutionäre Weiterentwicklung des Eclipse-Systems mit der Darstellung von Steward Brand, wie Gebäude über die Zeit hinweg lernen (Steward Brant: How Buildings Learn: What Happens After They're Built, Penguin, 1995). Eine wichtige Erkenntnis dabei ist, dass Gebäude, die eine lange Zeit über bestehen wollen, anpassbar und veränderbar sein müssen. Natürlich bezieht sich dies nicht auf alle Eigenschaften eines Gebäudes in gleichem Maße. Die tragenden Wände des Gebäudes, die grundlegende Architektur gewissermaßen, haben natürlich deutlich mehr Bestand und werden seltener verändert als die technische Innenausstattung des Gebäudes.

Unter der Haube – die Kolumne

- Eclipse Runtime
- Entwicklungsprozess des Eclipse-Plattform-Projekts
- **Build to last und API-Evolution**
- More to follow

Verglichen mit dem Eclipse-System bedeutet dies, dass die grundlegenden, tragenden Wände des Systems durch die Plug-in-Architektur realisiert werden. Die APIs zwischen den einzelnen Komponenten bilden die technische Grundausstattung des Eclipse-Gebäudes.

Für die Eclipse-APIs bedeutet dies, dass sie eine enorm wichtige Rolle spielen, da sie eine relativ lange Zeitspanne überdauern müssen. Sie sollten deshalb konsistent und präzise definiert werden und vor allem die Implementierungsdetails innerhalb der Komponente vor den Klienten verbergen. Innerhalb der Eclipse-Plattform wird deshalb das API eines Plug-ins strikt von den internen Details unterschieden. Die Implementierungsdetails einer Komponente erkennt man übrigens an dem Zusatz *internal* im Package-Namen.

Auf das API-Design wird innerhalb der Eclipse-Entwicklung sehr viel Wert gelegt, schon aufgrund der Tatsache, dass ein einmal veröffentlichtes API für eine sehr lange Zeit unterstützt und erhalten werden muss. Existieren also Zweifel an der Qualität des API oder an der dauerhaften Verwendbarkeit, werden die entsprechenden Klassen oder Interfaces zunächst nicht als offizielles API veröffentlicht, sondern verbleiben als interne Implementierungen innerhalb eines Plug-ins. Ein Mittel, um diese Zweifel von vornherein auszuräumen, ist, ein API nie ohne einen entsprechenden Klienten, also niemals auf Vorrat oder reine Vermutung hin zu implementieren.

Um zu vermeiden, dass Klienten ihren Code aufwendig an eine neue Version der Eclipse-Plattform anpassen müssen, gilt die so genannte Binary Compatibility als das oberste Ziel bei der Weiterentwicklung der APIs der Eclipse-Plattform. Binary Compatibility bedeutet dabei, dass vorhandene Klienten mit neuen Versionen der Plattform zusammenpassen und laufen müssen, ohne neu kompiliert werden zu müssen (mehr zu Binary Compatibility findet sich in der Java Language Specification, Kapitel 13: java.sun.com/docs/books/jls/second_edition/html/binaryComp.doc.html#44872). Offiziell wird die Binary Compatibility zwar nur innerhalb von Major Releases sichergestellt (also beispielsweise zwischen Version 3.0 und 3.1), aber in der Vergangenheit hat man stets auch zwischen Major Releases versucht, eine möglichst umfangreiche Binary Compatibility zu gewährleisten. Die Umstellung der Eclipse Runtime zwischen Version 2.1 und 3.0 veranschaulicht eindrucksvoll, mit welchem enormen Aufwand diese Kompatibilität realisiert wurde, um Klienten möglichst wenig Umstellungsaufwand aufzubürden.

Böse Überraschungen

Versuchen wir also, das API der Plattform binärkompatibel zu halten, während wir sie weiterentwickeln. Auf den ersten Blick scheint dies zunächst gar nicht so schwierig zu sein. Okay, wir können Klassen und Methoden nicht mehr so einfach umbenennen oder in ein neues Package verschieben, aber ansonsten ist doch alles möglich, oder?

Mitnichten. Ein Plattform-API kompatibel zu existierenden Klienten weiterzuentwickeln, ist eine äußerst schwierige Aufgabe. Jim de Rivières hat in seinem Artikel „Evolving Java-based APIs“ genau unter die Lupe genommen und aufgelistet, welche Änderungen an einem API noch kompatibel sind und welche nicht. Ein einfaches, aber schon nicht mehr triviales Beispiel ist das Hinzufügen einer Methode.

- Fügt man die Methode einem API-Interface hinzu, sind Klienten in der Regel nicht mehr kompatibel, wenn sie das Interface implementieren, da sie keine Implementierung für die neue Methode mitbringen (es sei denn, sie haben „aus versehen“ bereits eine entsprechende Methode in der implementierenden Klasse, bei der es dann jedoch fraglich ist, ob diese Methode auch die gleiche Funktionalität erbringt, wie in dem Interface neuerdings gefordert).
- Fügt man eine Methode einer API-Klasse hinzu, scheint zunächst einmal alles in Ordnung zu sein (solange es keine abstrakte Methode ist, dann läge der Fall ähnlich wie bei einer neuen Methode in einem Interface). Der Schein trügt jedoch, wenn Klienten Subklassen dieser API-Klasse implementiert haben. Befindet sich in einer solcher Subklasse zufälligerweise bereits die Methode, die in der API-Klasse neu hinzugekommen ist, befinden wir uns wiederum in der Situation, dass die Methode „aus versehen“ überschrieben wird und es äußerst unwahrscheinlich ist, dass dieses ungewollte Überschreiben die gleiche Semantik wie die neue Methode am API realisiert.

Schon an diesem einfachen Beispiel sehen wir, dass Änderungen an einem API sehr sorgfältig durchgeführt werden müssen, damit Binärkompatibilität sichergestellt werden kann. Aber was können wir tun, um einerseits binärkompatibel zu bleiben und andererseits das API trotzdem evolutionär weiter zu entwickeln?

Abstrakte Klassen vs. Interfaces

Veränderungen an API-Interfaces führen sehr schnell zu inkompatiblen Änderungen und sind somit überwiegend nicht möglich, ohne binär inkompatibel zu werden. Eine abstrakte Klasse kann demgegenüber einfacher zu ändern sein. Eine neue Methode kann in einer abstrakten API-Klasse leer implementiert werden und so verhindern, dass Klienten inkompatibel werden. Abstrakte Klassen verhalten sich also freundlicher gegenüber Änderungen als Interfaces. Sie sind allerdings nicht immer anstelle von Interfaces einsetzbar. Während Klienten entscheiden können, *welche* API-Interfaces in einer ihrer Klassen implementiert werden, kann eine Klient-Klasse immer nur von *einer* anderen (abstrakten) Klasse erben.

Am API einer Plattform können Interfaces also nicht grundsätzlich durch abstrakte Klassen ersetzt werden. Daher bietet es sich an, API-Interfaces grundsätzlich als unveränderlich anzunehmen. Soll das API-Interface trotzdem einmal verändert werden, hat es sich in der Eclipse-Entwicklung als nützlich herausgestellt, ein zweites Interface zu implementieren, welches das eigentliche API-Interface um die entsprechenden neuen Methoden erweitert. Diese erweiternden Interfaces werden fortlaufend nummeriert. Beispielsweise existiert im Eclipse-SDK das Interface *IWorkbenchPart2*, welches *IWorkbenchPart* erweitert und um zusätzliche Operationen anreichert. Im aktuellen Eclipse 3.2-Release finden sich eine ganze Reihe solcher Extension Interfaces, für das *ITextViewerExtension*-In-

terface existiert beispielsweise bereits das Interface *ITextViewerExtension6*.

Darüber hinaus ist in der Eclipse-Plattform mit dem *IAdaptable*-Interface eine Form des Extension-Objekt-Patterns implementiert, mit dem es möglich ist, eine Klasse auf den Typ eines anderen Interfaces zu adaptieren.

Einfach neu?

Neben dem Versuch, existierende APIs kompatibel zu verändern und mit entsprechenden Hilfsmitteln vorsichtig zu erweitern, ist eine zweite Vorgehensweise häufig attraktiver: Das alte API unverändert unterstützen und ein komplett neues API daneben zu stellen. Nachteil dieser Variante ist allerdings, dass so die Menge der APIs mit der Zeit stark anwächst. Neben den veralteten APIs, die immer noch unterstützt und gepflegt werden müssen, kommen immer neue hinzu, die inhaltlich die alten APIs ersetzen. Für den Klienten wird es immer schwieriger, die Menge der APIs zu durchschauen und das richtige API zu identifizieren. Und für die Plattform wird es immer schwieriger, alle Varianten des API zu unterstützen – auch wenn eine Variante des API bereits seit mehreren Versionen veraltet ist.

Teilen und Herrschen

Bisher haben wir lediglich API-Veränderungen auf der Sprachebene von Java betrachtet. Neben diesen reinen Sprach-API-Veränderungen können in Eclipse weitere Änderungen dem Plattform-Entwickler das Leben schwer machen: Änderungen an der Plug-in-Strukturierung.

Wollen wir beispielsweise ein Plug-in *org.eclipse.ui* in zwei neue Plug-ins *org.eclipse.ui.1* und *org.eclipse.ui.2* aufteilen, kommt es fast automatisch zu einer Inkompatibilität, obwohl wir den Sourcecode des Plug-ins selbst überhaupt nicht verändert haben. Klienten-Plug-ins referenzieren aber das Plug-in *org.eclipse.ui* als Abhängigkeit und gehen davon aus, dass von dem Plug-in ein entsprechendes API zur Verfügung gestellt wird. Wird das Plug-in aufgeteilt, greift diese Abhängigkeit des Klienten ins Leere und das Klienten-Plug-in funktioniert nicht mehr mit der neuen Version der Plattform.

Glücklicherweise können wir solchen Plug-in-Aufteilungen begegnen, indem wir das alte Plug-in in der Plattform belassen, die beiden neuen Plug-ins als Abhängigkeiten eintragen und derem API von dem alten, nun leeren Plug-in reexportieren lassen (siehe beispielsweise *org.eclipse.ui*).

Müssen an einem API Veränderungen durchgeführt werden, die sich nicht mehr durch die bisher genannten Techniken binärkompatibel halten lassen, muss man zu anderen Mitteln greifen. Als beispielsweise zwischen der Eclipse-Version 2.1 und 3.0 die Runtime komplett ausgetauscht wurde, war es letztendlich unmöglich, eine komplette Binärkompatibilität herzustellen, indem einfach bestehende Plug-ins unverändert blieben und neue daneben gestellt wurden. Damals hat man sich dazu entschieden, eine komplett neue Runtime separat zu implementieren und ein Compatibility-Plug-in zu implementieren. Dieses Compatibility-Plug-in hatte die Aufgabe, das alte Runtime API auf das neue Runtime API soweit wie möglich abzubilden.

Trotzdem besaßen natürlich existierende Klienten-Plug-ins eine Abhängigkeit zu *org.eclipse.core.runtime*. Diese Abhängigkeit wird von der neuen Runtime automatisch in eine Abhängigkeit zu *org.eclipse.core.runtime.compatibility* umgesetzt, sofern es sich um ein altes, Eclipse-2.x-basiertes Klienten-Plug-in handelt. Dadurch blieben alte Plug-ins binärkompatibel, wenn auch nur durch einen kleinen Trick.

Klienten migrieren

Auch wenn sich APIs nur binärkompatibel verändern, kommen Klienten letztendlich nicht darum herum, ihren Code irgendwann auf eine neue Plattform-Version zu migrieren. Schließlich möchte man von den neuen Möglichkeiten der Plattform profitieren.

Eine bekannte Technik, um Veränderungen am API einer Plattform für den Klienten abzufedern ist, die veralteten Teile des API zwar weiterhin zu unterstützen, aber als *@deprecated* zu markieren. Damit erhält der Klient des APIs automatisch vom Compiler Hinweise darüber, welche Teile des APIs, die er verwendet, veraltet sind. Wenn die Kommentare der *@deprecated* Tags ausführlich genug geschrieben sind, kann der Entwickler an ihnen sogar ablesen, was er tun muss, um

seinen Code auf die neue Variante des API umzustellen. Mit der Version 3.2 kann man mittels eines Refactoring-Skript solche Aufrufe auch durch einen Quick-Fix automatisiert beheben lassen.

Über das *@deprecated* Tag hinausgehend hat Stefan Roock in einem Buch über Refactorings in großen Projekten weitere Tags vorgestellt, die es dem Plattform-Entwickler erlauben, das API um weitere Informationen anzureichern. Diese weiterführenden Informationen erlauben es dem Nutzer des Plattform-APIs, sowohl vergangene Veränderungen detailliert nachzuvollziehen als auch auf zukünftige Veränderungen vorbereitet zu sein. Gerade Veränderungen an Vererbungsbeziehungen, die sehr schnell unangenehm werden können, spielen hierbei eine große Rolle.

Mit der Eclipse-Version 3.2 wird es darüber hinaus für Plattform-Entwickler möglich sein, Refactorings, die sie am API durchführen, aufzuzeichnen. Diese aufgezeichneten Refactoring-Skripte können den JAR Libraries der Plattform beigelegt und von Klienten auf ihren Code angewendet werden. Eclipse führt die Refactoring-Skripte auf Wunsch auf dem Klienten-Code aus, als ob dieser Code zur Verfügung gestanden hätte, als das Refactoring ursprünglich auf der Plattform ausgeführt wurde. Damit können Klienten die API-Refactorings der Plattform automatisch nachziehen und ihren Code so an die neue Version der Plattform anpassen. Diese Funktionalität ist natürlich auf solche Veränderungen am API der Plattform beschränkt, die mit den automatisierten Refactorings der Eclipse-IDE durchgeführt und so in Skriptform aufgezeichnet wurden. Alle anderen, manuell durchgeführten Veränderungen am API bleiben davon, leider, unberührt.

Fazit

Wir haben gesehen, dass es für eine Plattform eine entscheidende Rolle spielt, wie sie mit einmal publizierten APIs umgeht. Und obwohl wir hier hauptsächlich die Eclipse-Plattform als Beispiel betrachtet haben, können diese Einsichten auch in anderen Softwareprojekten, auch In-House-Projekten, eine große Rolle spielen. Gerade, wenn eine unternehmenseinheitliche Plattform geschaffen werden soll, auf deren Basis unterschiedliche Teams Anwendungen implementieren.

Das Design dieser Published APIs wird dabei zunehmend wichtiger und kann schnell dazu führen, dass schlecht entworfene oder einfach inkompatibel geänderte APIs den Ruf einer Plattform beschädigen und Klienten der Plattform vergraulen. Für die Entwickler einer Plattform ist es allerdings mit einem enormen Aufwand verbunden, die einmal publizierten APIs der Plattform auch in zukünftigen Versionen binärkompatibel zu halten und trotzdem evolutionär weiterzuentwickeln, um mit neuen Anforderungen und verbesserten Architekturen aufwarten zu können.

Wir sehen allerdings auch, dass immer mehr veraltete, aber aus Kompatibilitätsgründen gepflegte APIs das gesamte API einer Plattform immer stärker verschmutzen und die Weiterentwicklung der Plattform immer weiter einschränken. Optimierungen selbst an internen Implementierungen werden zunehmend schwieriger, weil veraltete APIs funktionsfähig erhalten werden müssen. Aus technischer Sicht würde eine Plattform also sicherlich von einer großen Last befreit werden, wenn veraltete APIs in einer zukünftigen Version einfach einmal entfernt würden. Auch die Nutzer dieser Plattform werden nach einigem Umstellungsaufwand sicherlich die Vorteile dieser „Entrümpelung“ zu schätzen wissen.

Mein Dank geht an Bernd Kolb für Feedback zu einer Draft-Version dieses Artikels.

Martin Lippert ist Senior-IT-Berater bei it-agile. 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@it-agile.de.

Links & Literatur

[1] Bill Venners: Design Principles from Design Patterns, A Conversation with Erich Gamma, Part III: www.artima.com/lejava/articles/designprinciples.html

[2] Jim des Rivières: Evolving Java-based APIs: www.eclipse.org/eclipse/development/java-api-evolution.html

[3] Stefan Roock, Martin Lippert: Refactorings in großen Softwareprojekten: Komplexe Restrukturierungen erfolgreich durchführen, dpunkt 2004