Anwendungsentwicklung mit Plug-in-Architekturen: Erfahrungen aus der Praxis

Jörg Rathlev
Fachbereich Informatik, Universität Hamburg
rathlev@informatik.uni-hamburg.de

Abstract: In der Anwendungsentwicklung kommen zunehmend Plug-in-basierte Ansätze zum Einsatz. Die Verwendung Plug-in-basierter Techniken hat Auswirkungen auf die Entwicklung der Software, in der die spezifischen Eigenschaften von Plug-ins berücksichtigt werden müssen. Dieser Beitrag identifiziert basierend auf Erfahrungen aus der Praxis Entwurfsfragen, die sich beim Einsatz von Plug-ins stellen, und diskutiert Entwurfsalternativen und deren Auswirkungen.

1 Einleitung

Die Anforderungen an ein Softwaresystem unterscheiden sich je nach Anwendungskontext und können sich im Laufe der Zeit verändern. Um kostspielige Neuentwicklungen zu vermeiden, sollte Software daher anpassbar und erweiterbar gestaltet sein. Plug-in-basierte Entwicklung ist ein Ansatz, der verspricht, die Implementierung von erweiterbarer und anpassbarer Software zu unterstützen und zu vereinfachen. Unter anderem durch die Verbreitung Plug-in-basierter Plattformen wie Eclipse gewinnt dieser Ansatz an Bedeutung.

Die Entscheidung, ein Softwaresystem Plug-in-basiert zu entwickeln, hat Auswirkungen auf dessen Architektur und die Organisation des Entwicklungsprojektes. Offensichtlich muss das System geeignet in Plug-ins aufgeteilt werden. Daneben gibt es aber noch weitere, auf den ersten Blick weniger offensichtliche Auswirkungen, die aus den spezifischen Eigenschaften von Plug-ins folgen. In der praktischen Anwendung sehen Entwickler sich daher mit einer Reihe neuer Entwurfsfragen konfrontiert, wenn sie Plug-in-basierte Techniken einsetzen.

Dieser Beitrag identifiziert basierend auf Erfahrungen aus der Praxis die Gründe, aus denen Plug-in-Architekturen eingesetzt werden, sowie Entwurfsfragen und organisatorische Herausforderungen, die sich beim Einsatz von Plug-ins stellen.

Die in diesem Beitrag präsentierten Ergebnisse basieren auf Erfahrungen aus vier verschiedenen Softwareprojekten. Bei dem ersten Projekt handelt es sich um ein seit fünf Jahren laufendes Industrieprojekt, in dem eine integrierte Werkzeugumgebung für die Steuerung und Überwachung technischer Anlagen entwickelt wird. Der Autor dieses Beitrags war im Rahmen einer Forschungskooperation für drei Jahre als Softwareentwickler in diesem Projekt tätig.

Das zweite Projekt ist ein Forschungsprojekt, das vom Autor in beobachtender Rolle be-

gleitet wird. In diesem Projekt soll ein komponentenbasiertes Rahmenwerk für die Entwicklung von Leitstand-Systemen entwickelt werden. Sowohl das erste als auch das zweite Projekt basieren in der Implementierung auf der Eclipse Rich Client Platform.

Um die Erkenntnisse abzusichern, wurden Interviews mit Entwicklern aus zwei weiteren Projekten durchgeführt. Bei dem dritten Projekt handelt es sich um die Entwicklung eines Plug-in-Rahmenwerks für die .NET-Plattform, das von einem Hamburger Unternehmen für Anwendungen im Bereich betrieblicher Umweltinformationssysteme eingesetzt wird. Das vierte Projekt ist ein Forschungsprojekt an der Universität Hamburg, in dem dieses Rahmenwerk eingesetzt wird, um ein System zu entwickeln, das den Handel mit Emmissionsrechten unterstützt.

Außerdem wurden die Organisation und Architektur des Eclipse-Projektes untersucht. Als Informationsquellen dienten hier die eigenen Erfahrungen des Autors mit der Eclipse-Plattform sowie die verfügbare Literatur und Dokumentation.

Dieser Beitrag gibt nachfolgend zunächst eine kurze Einführung in die Plug-in-basierte Entwicklung. Anschließend wird beschrieben, mit welcher Motivation in den beobachteten Projekten Plug-in-basierte Techniken eingesetzt wurden. Danach werden die beobachteten organisatorischen und technischen Herausforderungen und Fragestellungen diskutiert.

2 Plug-ins und Plug-in-Architekturen

Der Begriff *Plug-in* wird mit verschiedenen Bedeutungen verwendet, sowohl in der Literatur als auch in der Praxis.

Im Allgemeinen versteht man unter einem Plug-in eine Erweiterung einer vorhandenen Anwendung, wobei die Erweiterung erst zur Laufzeit von der Anwendung dynamisch geladen wird [Mar06]. Dies ermöglicht es, die Erweiterung unabhängig von der Anwendung auszuliefern und nachträglich zu installieren. Die Integration der Erweiterung in die Anwendung erfolgt über eine Plug-in-Schnittstelle. Ist die Spezifikation dieser Schnittstelle offengelegt, so können Plug-ins nicht nur vom Hersteller der Anwendung selbst, sondern auch von Dritten entwickelt werden. Plug-ins können aber in der Regel nur die Basisanwendung erweitern, die Erweiterung anderer Plug-ins ist nicht möglich.

Eine solche Plug-in-Schnittstelle wird heute von vielen am Markt etablierten Softwareprodukten bereitgestellt. Zu den bekanntesten Beispielen gehören Bildbearbeitungswerkzeuge und Web-Browser.

So genannte *reine Plug-in-Architekturen* verbinden das Plug-in-Konzept mit Konzepten der komponentenbasierten Entwicklung [Rat08]. Bei einer solchen Architektur werden Plug-ins als das grundlegende Zerlegungskonzept für das gesamte Softwaresystem verwendet. Die Software wird nicht in eine Anwendung und Plug-ins getrennt, sondern sie wird vollständig aus Plug-ins gebaut. Technisch besteht kein Unterschied zwischen den Plug-ins, die die Basisanwendung implementieren, und den Plug-ins, die diese erweitern. Jedes Plug-in kann Schnittstellen zur Erweiterung durch andere Plug-ins bereitstellen [Bir05].

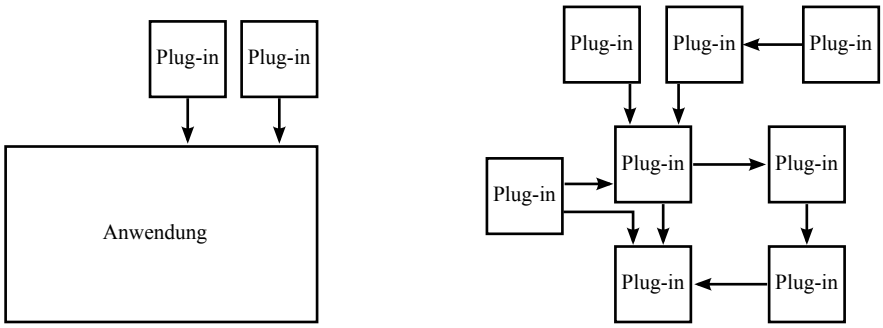

Abbildung 1: Erweiterbare Anwendungen und reine Plug-in-Architekturen

Anders als bei einer erweiterbaren Anwendung kann bei einer Anwendung mit einer Plug-in-Architektur nicht nur das bestehende System erweitert werden, sondern Anwender oder Drittentwickler können auch Teile der Basisanwendung austauschen oder entfernen und die Anwendung so an spezielle Anwendungskontexte anpassen.

In Abbildung 1 werden der klassische Plug-in-Begriff für erweiterbare Anwendungen und das Konzept der reinen Plug-in-Architektur schematisch gegenübergestellt.

Plug-ins sind Softwarekomponenten im Sinne von [SGM02], die Code zur Benutzung durch andere Plug-ins exportieren bzw. von anderen Plug-ins importieren können. Neben dieser Benutzungsbeziehung verfügen Plug-in-basierte Systeme aber vor allem über einen *Erweiterungsmechanismus*. Über diesen kann ein Plug-in zur Laufzeit dynamisch ermitteln, welche Erweiterungen von anderen Plug-ins angeboten werden, und auf diese zugreifen [Rat08]. Der Erweiterungsmechanismus kann technisch auf unterschiedliche Weise realisiert werden, die konkrete Realisierung spielt für die Betrachtungen in diesem Beitrag jedoch keine Rolle.

Im weiteren Verlauf dieses Beitrags wird der Begriff „Plug-in“ im Sinne von Plug-ins in einer reinen Plug-in-Architektur verwendet, wenn nicht explizit etwas anderes gesagt wird.

3 Gründe für die Wahl einer Plug-in-Architektur

Für die Entscheidung, eine Anwendung Plug-in-basiert zu entwickeln, gibt es verschiedene Gründe. Naheliegend ist die Wahl einer Plug-in-Architektur, um die Erweiterbarkeit einer Anwendung zu verbessern.

Durch die Erweiterbarkeit ermöglichen es Plug-ins, die Entwicklungszyklen verschiedener Bestandteile einer Anwendung voneinander zu entkoppeln, sodass schnell ein Kernsystem fertiggestellt werden kann, das die Mindestanforderungen eines Anwendungsbereichs erfüllt. Zusätzlich gewünschte Funktionen können dann zu einem späteren Zeitpunkt in Form von Plug-ins erstellt werden, die dieses Minimalsystem erweitern [Mar06].

Während Erweiterbarkeit auch durch eine klassische Plug-in-Schnittstelle in einer erweiterbaren Anwendung erreicht werden kann, verspricht die Wahl einer Plug-in-Architektur zusätzliche Verbesserungen bei der Anpassbarkeit. In einer Plug-in-Architektur werden für den Austausch von Teilen eines Systems dieselben technischen Mittel verwendet wie für die Erweiterung des Systems. Teile eines Softwaresystems, die als Plug-ins implementiert sind, können einfach gegen eine alternative Implementierung ausgetauscht werden oder auch aus dem System entfernt werden. Für unterschiedliche Anwendungskontexte können so unterschiedliche Konfigurationen des Systems erstellt werden, die jeweils genau die für den Kontext benötigten Plug-ins umfassen. Voraussetzung hierfür ist natürlich, dass die Software geeignet in Plug-ins aufgeteilt wurde.

Eine weitere Motivation für die Wahl einer Plug-in-Architektur, die in allen für diesen Beitrag untersuchten Projekten genannt wurde, war der Wunsch, bei der Entwicklung der Software mit anderen Unternehmen kooperieren zu können oder eine Plattform zu etablieren, auf deren Basis auch Dritte individuelle Lösungen entwickeln können. Eine solche Kombination von kooperativer Entwicklung über Unternehmensgrenzen hinweg einerseits und individueller Anpassung andererseits wird auch als ein Software-Ökosystem bezeichnet [Bos09]. Für Unternehmen kann die Entwicklung einer Plattform einen Wettbewerbsvorteil darstellen. Cusumano fasst in [Cus10] verschiedene Arbeiten zum Thema Plattform-Strategien zusammen.

Schließlich wird die Verwendung einer Plug-in-Architektur teilweise damit begründet, dadurch die Funktionalität eines vorhandenen, Plug-in-basierten Rahmenwerks verwenden zu können. Zum Beispiel wurde in einigen Projekten die Entscheidung für die Eclipse Rich Client Platform unter anderem damit begründet, dass diese umfangreiche, vorgefertigte Funktionen für die Gestaltung der Benutzeroberfläche mitliefert.

4 Organisatorische Aspekte

Wie im vorangegangenen Abschnitt geschildert, war in allen untersuchten Projekten eines der Ziele, eine Plattform für die Entwicklung individuell angepasster Softwarelösungen zu etablieren. Dieses Ziel steht im Konflikt mit dem Ziel, für einen konkreten Kontext eine funktionsfähige, einsatzfertige Anwendung zu entwickeln. Zwischen beiden Zielen muss eine geeignete Balance gefunden werden.

In allen untersuchten Softwareprojekten lag eine Mischung aus Anwendungs- und Plattformentwicklung vor:

- In dem ersten Projekt war das Ziel, eine Anwendung zur Steuerung der eigenen technischen Anlagen zu entwickeln. Gleichzeitig wurde als Basis für diese Anwendung eine eigene, offene Plattform entwickelt mit dem Ziel, dass andere Einrichtungen mit ähnlichen Anforderungen diese Plattform nutzen können und sich nach Möglichkeit an ihrer Weiterentwicklung beteiligen.
- In dem zweiten Projekt soll eine Plattform für die Entwicklung von individuellen Leitstand-Implementierungen entwickelt werden. Dazu werden zunächst für ver-

schiedene Projektpartner individuelle Anwendungen implementiert. In diesen sollen dann wiederverwendbare, gemeinsame Bestandteile identifiziert und zu einer Plattform zusammengeführt werden.

- Das dritte Projekt war ein Projekt zur Plattformentwicklung, in dem ein .NET-Rahmenwerk entwickelt wurde. Dieses wird jetzt eingesetzt, um kundenspezifische Anwendungen zu entwickeln.
- Im vierten Projekt soll eine Anwendung entwickelt werden. Diese soll erweiterbar sein um individuelle Anpassungen, beispielsweise zur Anbindung an vorhandene Informationssysteme.

Bei der Anwendungsentwicklung ist das primäre Ziel, für konkrete Anwendungsfälle eine Lösung zu entwickeln, mit der die Anforderungen der Endanwender erfüllt werden, ohne dass weitere Anpassungen nötig sind. Die Anwendungsentwicklung findet meist im Rahmen von einzelnen Projekten statt. Der Projekterfolg ist dabei definiert durch die erfolgreiche Fertigstellung einer einsetzbaren Anwendung.

Bei der Plattformentwicklung dagegen sind die Zielgruppe Softwareentwickler, die die Plattform einsetzen. Eine Plattform muss ausreichend allgemein gehalten werden, um für verschiedene Anforderungen nutzbar zu sein. Die Plattformentwicklung ist meist eher langfristig ausgerichtet und bildet die Basis für zukünftige Projekte, führt jedoch als solche nicht zum Projekterfolg in Anwendungsprojekten.

Die Plattformentwicklung ist mit einem höheren initialen Aufwand verbunden als die Anwendungsentwicklung. Dies liegt zum einen daran, dass bei der Anforderungsermittlung die Anforderungen mehrerer Einsatzkontexte erfasst und berücksichtigt werden müssen. Zum anderen steigt auch der Testaufwand, weil zusätzliche Integrationstests notwendig werden.

Die Plattformentwicklung kann auch die Flexibilität einschränken, weil Schnittstellen einer gemeinsam genutzten Plattform nicht einseitig geändert werden können, ohne dass dadurch die Kompatibilität eingeschränkt würde. Bei der Anwendungsentwicklung dagegen ist es häufig möglich, Schnittstellen anzupassen, weil sich alle Nutzer der Schnittstelle unter der Kontrolle der Entwickler befinden. Bei einer Plattform, die auch von Dritten genutzt wird, ist dies nicht der Fall. Entwickler müssen dies berücksichtigen, wenn sie Schnittstellen entwerfen und entscheiden, welche Schnittstellen als Teil der Plattform freigegeben werden.

Für den Umgang mit dem Konflikt zwischen Plattform- und Anwendungsentwicklung haben wir in den untersuchten Projekten verschiedene Strategien beobachtet, die nachfolgend dargestellt werden.

Das zweite und vierte untersuchte Projekt befinden sich noch am Anfang ihrer Implementierungsphasen, sodass hier noch keine Aussagen getroffen werden können.

Im ersten Projekt gab es keine expliziten Richtlinien oder Prozesse für Änderungen an der Plattform. In der Praxis wurden geplante Änderungen an der Plattform über eine Mailingliste abgesprochen, die von Entwicklern aus den an der Entwicklung beteiligten Organisationen abonniert wurde. Kleinere Änderungen, beispielsweise das Hinzufügen einer

kleineren Funktion oder Bugfixes, wurden teilweise auch ohne Absprache vorgenommen.

Ein Risiko dieser Strategie ist, dass es unbeabsichtigt zu einer parallelen Entwicklung ähnlicher Funktionalität in verschiedenen Anwendungs-Plug-ins kommen kann. Dieses Risiko steigt insbesondere dann, wenn die Anwendungsentwicklung unter Termindruck stattfindet, weil Entwicklern dann die Zeit fehlt, eine gemeinsame Implementierung von Basisfunktionen zu entwerfen und in die Plattform einfließen zu lassen. In dem beobachteten Projekt entstanden mehrfach solche parallel entwickelten Funktionen, die umfangreiche Umbauten erforderten, um sie in die Plattform einfließen zu lassen.

Von einem externen Entwickler wurde außerdem die Einschätzung geäußert, dass der Aufwand, Änderungen an der Plattform zu koordinieren, zu hoch sei. Er hat daher von eigenen Anwendungs-Plug-ins gemeinsam benötigte Funktionen teilweise nicht in die „offizielle“ Plattform einfließen lassen, sondern zusätzlich organsationseigene Plattform-Plug-ins entwickelt.

Im dritten Projekt wird die Plattform nur noch im Rahmen von Projekten zur Anwendungsentwicklung weiterentwickelt. Neue Funktionen entstehen zunächst in der Anwendungsentwicklung und werden in die Plattform eingebracht, wenn es als sinnvoll erachtet wird, sie als Teil der Plattform zur Verfügung zu stellen. Diese Strategie ähnelt der Strategie, die bei der Weiterentwicklung der Eclipse-Plattform angewendet wird.

Bei der Weiterentwicklung von Eclipse werden neue Funktionen bewusst erst dann in die Plattform aufgenommen, wenn es auch einen konkreten Nutzer für die jeweilige Schnittstelle gibt. Dabei wird zwischen öffentlichen und veröffentlichten Schnittstellen unterschieden (vgl. dazu [Fow02]), d. h. es werden technisch auch solche Schnittstellen für die Nutzung durch andere Plug-ins exportiert, die noch kein Teil der offiziellen Plattform sind. Durch diese Vorgehensweise soll sichergestellt werden, dass eine Schnittstelle ausreichend ausgereift und im praktischen Einsatz getestet ist, bevor sie Teil der Eclipse-Plattform wird. Die Unterscheidung zwischen öffentlichen und veröffentlichten Schnittstellen erfolgt dabei mit Hilfe von Namenskonventionen [dR01, GB03].

5 Technische Aspekte

Neben den organisatorischen Fragen hat der Einsatz einer Plug-in-Architektur natürlich auch Auswirkungen auf den softwaretechnischen Entwurf eines Softwaresystems. Nachfolgend werden Entwurfsfragen diskutiert, die sich in diesem Kontext stellen.

5.1 Austausch und Wiederverwendung von Plug-ins

Der Einsatz von Plug-ins ermöglicht es, Teile einer Anwendung zur Ladezeit oder sogar erst zur Laufzeit auszutauschen oder zu entfernen. Dabei können zwei Arten von Austauschbarkeit unterschieden werden: erstens der Austausch eines Plug-ins gegen ein anderes Plug-in (Austausch) und zweitens der Einsatz eines Plug-ins als Erweiterung in einer

anderen Anwendung (Wiederverwendung).

5.1.1 Austausch

Durch den Erweiterungsmechanismus einer Plug-in-Architektur wird vorrangig der Austausch von Plug-ins unterstützt. Über diesen Mechanismus kann ein Plug-in zur Laufzeit dynamisch die installierten Erweiterungen auffinden und auf diese zugreifen. Ein Plug-in, das Erweiterungen akzeptiert, bildet somit eine Art Rahmenwerk für seine Erweiterungen.

Die Kontrolle darüber, wann welche Erweiterungen geladen werden, liegt bei den Plug-ins, die diese Erweiterungen verwenden. Dadurch werden nicht zwingend alle angebotenen Erweiterungen auch tatsächlich zur Laufzeit in das System eingebunden. Es stellt technisch keinen Fehler dar, wenn ein Plug-in eine Erweiterung anbietet, für die sich im System zur Laufzeit keine Nutzer finden. Somit können Erweiterungen auch für optionale Systemteile angeboten werden, von denen der Entwickler einer Erweiterung nicht wissen kann, ob sie in einem konkreten System installiert sind.

Diese Eigenschaft von Plug-in-Systemen ist wünschenswert, weil sie zu einer losen Kopplung von Plug-ins führt und die Voraussetzung dafür schafft, verschiedene Systemteile und Erweiterungen miteinander zu integrieren, ohne Abhängigkeiten zwischen diesen zu erzwingen.

Dennoch kann dieses Verhalten in der Entwicklungspraxis auch zu Schwierigkeiten führen. Vor allem die Fehlersuche wird erschwert, wenn eine Erweiterung entwickelt wurde, die vom System nicht wie erwartet eingebunden wird. Weil hier technisch kein Fehler vorliegt, fehlt Entwicklern in dieser Situation zunächst ein Anhaltspunkt, wo die Ursache liegen könnte. Eine triviale, in der Praxis aber häufig beobachtete Ursache für solche Probleme sind zum Beispiel Tippfehler bei der Angabe von textuellen IDs. Mit zunehmender Praxiserfahrung fällt es Entwicklern leichter, solche Fehler aufzuspüren.

5.1.2 Wiederverwendung

Die Wiederverwendung von Plug-ins in anderen Kontexten wird durch Erweiterungsmechanismen nur eingeschränkt verbessert. Theoretisch kann ein Erweiterungsmechanismus zwar verwendet werden, um Wiederverwendung zu ermöglichen. Insbesondere die oben genannte Möglichkeit, Erweiterungen für unterschiedliche Kontexte bereitzustellen, von denen dann jeweils nur die im konkreten Kontext nutzbaren aufgefunden und eingebunden werden, erlaubt sehr flexible Szenarien.

In der Praxis haben wir jedoch beobachtet, dass viele Plug-ins implizite Annahmen über ihren Ausführungskontext machen, durch die ihre Einsetzbarkeit in anderen Kontexten eingeschränkt wird. Dazu gehören beispielsweise Annahmen darüber, welche der angebotenen Erweiterungen in jedem Fall in das System eingebunden werden, oder Annahmen darüber, wie die Benutzeroberfläche des erweiterten Systems aufgebaut ist und wo Erweiterungen in diese eingebunden werden können.

Folgendes Beispiel aus dem ersten untersuchten Projekt hilft, dieses Problem zu illustrieren: Ein Plug-in, das urpsrünglich für die in dem Projekt entstandene Werkzeugum-

gebung entwickelt wurde, sollte nun auch für Softwareentwickler zur Verfügung gestellt werden, die es in ihre integrierte Entwicklungsumgebung einbinden möchten. Aus technischer Sicht schien dies möglich, da sowohl die Werkzeugumgebung aus dem Projekt als auch die Entwicklungsumgebung (die Eclipse IDE) auf der Eclipse-Plattform basierten. In der Praxis stellte sich jedoch heraus, dass die Nutzung des Plug-ins in der Eclipse IDE nur mit Einschränkungen möglich war, weil durch die abweichenden Menüstrukturen in der Eclipse IDE gegenüber der Werkzeugumgebung in der IDE nicht alle Menüpunkte eingebunden wurden.

Solche impliziten Annahmen können (zumindest mit den in den untersuchten Projekten eingesetzten Techniken) nicht vollständig explizit gemacht werden, ohne dadurch gleichzeitig die Wiederverwendbarkeit noch stärker einzuschränken. Würde ein Plug-in eine Abhängigkeit zu den anderen Plug-ins deklarieren, die einen den Annahmen entsprechenden Kontext bilden, so wären zwar die Abhängigkeiten explizit, ein Einsatz in einem anderen Kontext jedoch überhaupt nicht mehr möglich.

Entwickler müssen daher beim Entwurf von Plug-ins darauf achten, implizite Abhängigkeiten soweit wie möglich zu vermeiden, ohne dadurch die Verwendbarkeit des Plug-ins unnötig einzuschränken.

5.2 Verwendung unterschiedlicher Erweiterungsmechanismen

Aktuelle Plug-in-Rahmenwerke bieten häufig verschiedene Erweiterungsmechanismen nebeneinander an. In der Eclipse Rich Client Platform beispielsweise können sowohl Extension Points [GB03] als auch OSGi Services [WHKL08] als Erweiterungsmechanismus verwendet werden.

Für Entwickler wirft diese Situation einerseits die Frage auf, welchen der verfügbaren Mechanismen sie verwenden sollen, wenn sie eine Erweiterungsschnittstelle bereitstellen möchten (mehr zu dieser Frage im nachfolgenden Abschnitt), und andererseits die Frage, wie über verschiedene Mechanismen angebotene Erweiterungen miteinander kombiniert werden können.

Zu Problemen führt die Kombination verschiedener Mechanismen dann, wenn aus einer Erweiterung heraus, die über einen Mechanismus instanziert wird, auf Erweiterungen zugegriffen werden soll, die über einen anderen Mechanismus angeboten werden. Im ersten Projekt konnten wir dieses Problem mehrfach beobachten, weil dort der Bedarf bestand, aus über den Extension-Point-Mechanismus eingebundenen Erweiterungen heraus auf OSGi Services zuzugreifen. Ein direkter Zugriff ist hier nicht möglich, weil die über den Extension Point eingebundene Erweiterung von der Eclipse-Plattform instanziert wird und dabei keine Referenz auf die OSGi-Umgebung erhält.

Von den Entwicklern wurde dieses Problem dadurch umgangen, dass entsprechende Referenzen auf die OSGi-Umgebung oder die benötigten Services in statischen Variablen gespeichert wurden. Dadurch kam es jedoch mehrfach zu Fehlern durch unzutreffende Annahmen über die Lebenszyklen der Objekte, zum Beispiel weil versucht wurde, auf noch nicht verfügbare Services zuzugreifen.

Dasselbe Problem wurde auch im zweiten Projekt beobachtet. Hier wurde von Entwicklern der Vorschlag geäußert, eine bestimmte Konfiguration der Startreihenfolge der Plug-ins vorzuschreiben, um eine größere Kontrolle über die Lebenszyklen zu erlangen.

In der neuen Eclipse-Version 4.0, die im Juni 2010 veröffentlicht wurde, wird für Erweiterungen Dependency Injection unterstützt [Art09]. Das macht es möglich, für Erweiterungen deklarativ bestimmte Abhängigkeiten anzugeben, diese werden dann bei der Erzeugung der Erweiterung automatisch von der Eclipse-Plattform in die Erweiterung injiziert. Es muss sich noch zeigen, ob dieser Mechanismus die genannten Probleme beheben kann.

Einen experimentellen Dependency-Injection-Mechanismus für die Integration von OSGi Services hat Bartlett entwickelt [Bar].

Ein vergleichbares Problem, das ebenfalls auf inkompatible Lebenszyklen der von Plug-ins instanzierten Objekte zurückzuführen ist, kann sich bei der Portierung von Plug-in-basierten Anwendungen ins Web ergeben. Im Eclipse-Umfeld existiert mit der Rich Ajax Platform [RAP] ein Rahmenwerk, das diese Portierung ermöglichen soll. Im ersten Projekt haben wir dieses Rahmenwerk untersucht und dabei festgestellt, dass unterschiedliche Lebenszyklen das größte Hindernis bei der Portierung bilden. Für eine Web-Anwendung wäre ein zusätzlicher Session-Lebenszyklus notwendig, der in einer Desktop-Anwendung jedoch in der Regel nicht vorgesehen ist [CHRM09].

5.3 Erweiterbarkeit

Wird in einem Plug-in Funktionalität implementiert, die anderen Plug-ins zur Verfügung gestellt werden soll, so stellt sich die Frage, ob dies über einen Erweiterungsmechanismus realisiert werden soll oder ob das Plug-in eine entsprechende Programmierschnittstelle (API) exportieren soll. Wird ein Erweiterungsmechanismus verwendet, so übernimmt das Plug-in gegenüber Klienten die Rolle eines Rahmenwerks. Das bedeutet, es kontrolliert, wann welche Erweiterungen verwendet werden. Exportierte APIs hingegen können von anderen Plug-ins ähnlich wie eine gewöhnliche Bibliothek importiert und aufgerufen werden.

In keinem der untersuchten Projekte existierten Richtlinien für die Entwickler dazu, für welche Aufgaben ein Erweiterungsmechanismus verwendet werden sollte.

Wird ein Erweiterungsmechanismus verwendet und stehen in der verwendeten Plattform mehrere alternative Mechanismen zur Verfügung, so stellt sich zusätzlich die Frage, welcher von diesen verwendet werden sollte. Auch hierzu gab es in keinem Projekt Richtlinien. Für die Eclipse-Plattform finden sich im Internet Vergleiche der verschiedenen Mechanismen, beispielsweise [Bar07].

6 Auswirkungen auf die Benutzbarkeit

Eine Plug-in-Architektur zu verwenden, kann sich auch auf die Benutzbarkeit einer Anwendung auswirken. Bereits zuvor angesprochen wurde das Problem, dass Plug-ins implizite Annahmen über den Aufbau der Benutzeroberfläche machen können, in die sie eingebunden werden.

Eine andere Eigenschaft Plug-in-basierter Systeme, die zu Problemen führen kann, ist die verzögerte Aktivierung (Lazy Activation) von Plug-ins. Im Eclipse-Rahmenwerk wird die Implementierung eines Plug-ins erst direkt vor der Ausführung des Codes geladen. Zuvor verwendet die Plattform deklarative Informationen, um die von dem Plug-in beigetragenen Erweiterungen bereits an der Oberfläche darstellen zu können. Dadurch soll die Ladezeit von Eclipse-basierten Anwendungen verringert werden [GB03].

Für den Benutzer ist nicht erkennbar, zu welchen an der Oberfläche dargestellten Elementen die zugehörige Implementierung bereits geladen wurde. Dies kann zu nicht erwartungskonformen Verhalten der Anwendung führen, wenn der Anwender ein Element als „aktiv, aber im Hintergrund“ wahrnimmt, es tatsächlich jedoch noch nicht aktiviert wurde.

Im ersten Projekt trat dieses Problem im Zusammenhang mit Ansichten (Views) auf, die Nachrichten über das Netzwerk empfangen und auflisten. Befand sich eine solche Ansicht beim Start der Anwendung im Hintergrund, so wurde sie von Eclipse bereits dargestellt, der zugehörige Code jedoch noch nicht geladen. Der Benutzer nahm diese Ansicht jedoch als „im Hintergrund geöffnet“ wahr und erwartete, dass auch bereits im Hintergrund Nachrichten empfangen werden. Tatsächlich wurde die Netzwerkverbindung jedoch erst aufgebaut, sobald der Benutzer die Anwendung zum ersten Mal in den Vordergrund holte. Dieses Verhalten war für die Benutzer unerwartet. Es führte außerdem dazu, dass eine im Hintergrund geöffnete Ansicht sich in zwei unterschiedlichen Zuständen befinden konnte, die an der Oberfläche nicht unterscheidbar waren.

Um solche Probleme zu vermeiden, sollten Entwickler bei der Programmierung Anwendungen mit einer Plug-in-Architektur darauf achten, dass durch die Erweiterungsbeziehungen zwischen Plug-ins keine Inkonsistenzen zwischen dem tatsächlichen und dem vom Benutzer wahrnehmbaren Zustand der Anwendung entstehen können.

7 Zusammenfassung

Basierend auf Erfahrungen aus Softwareprojekten in der Praxis wurden in diesem Beitrag Besonderheiten bei der Entwicklung Plug-in-basierter Software dargestellt. Plug-ins ermöglichen es, Software zu entwickeln, die auch nach der Auslieferung beim Anwender verändert und aktualisiert werden kann. Dies schafft die Möglichkeit, Software über Organisationsgrenzen hinweg kooperativ zu entwickeln. Technisch ermöglicht wird diese Flexibilität durch einen komponentenbasierten Ansatz in Verbindung mit einem Erweiterungsmechanismus, der es ermöglicht, Plug-ins mit Erweiterungen dynamisch in ein System einzufügen und aus diesem zu entfernen.

Organisatorisch müssen Unternehmen, die Plug-ins gewinnbringend einsetzen möchten, eine geeignete Entwicklungsstrategie wählen. Der Konflikt zwischen der Plattform- und der Anwendungsentwicklung bildet hier ein Spannungsfeld, in dem geeignete Kompromisse gefunden werden müssen. Dieses Spannungsfeld hat auch Auswirkungen auf die Gestaltung der Architektur eines Softwaresystems, weil es den Entwurf und die Freigabe der Schnittstellen zwischen seinen Komponenten beeinflusst.

In der Praxis haben wir unterschiedliche Strategien beobachtet, mit denen Unternehmen diesen Konflikt zu adressieren versuchen. Bisher können wir keine eindeutige Antwort darauf liefern, welche Strategie in welcher Situation am besten geeignet ist.

Auf der technischen Seite stellt Entwickler vor allem die hohe Dynamik in Plug-in-basierten Systemen vor Herausforderungen. Entwickler müssen ermitteln, an welchen Stellen Erweiterbarkeit erforderlich ist, um fachlich motivierte Änderungsszenarien zu unterstützen, und an welchen Stellen eine klassische, programmatische Schnittstelle bevorzugt werden sollte, um den Entwicklungsaufwand zu reduzieren.

Als problematisch herausgestellt haben sich in der Praxis vor allem die Wiederverwendung von Plug-ins in Kontexten, für die diese ursprünglich nicht entworfen wurden, und die gleichzeitige Nutzung verschiedener Erweiterungstechniken. Bei letzterem müssen Entwickler damit umgehen, dass Erweiterungen von unterschiedlichen Containern instanziert werden. Aktuelle Neuentwicklungen wie Eclipse 4.0 versprechen hier Abhilfe durch die bessere Unterstützung von Techniken wie Dependency Injection, werden jedoch in der Praxis bisher kaum eingesetzt.

Literatur

[Art09] John Arthorne. *White Paper: e4 Technical Overview*, 2009. `http://www.eclipse.org/e4/resources/e4-whitepaper.php`, letzer Abruf 8.10.2010.

[Bar] Neil Bartlett. *Extensions2Services*, Projekt-Website. `http://github.com/njbartlett/Extensions2Services`, letzter Abruf 8.10.2010.

[Bar07] Neil Bartlett. *A Comparison of Eclipse Extensions and OSGi Services*, 2007. `http://www.eclipsezone.com/articles/extensions-vs-services/`, letzter Abruf 6.10.2010.

[Bir05] Dorian Birsan. On plug-ins and extensible architectures. *Queue*, 3(2):40–46, 2005.

[Bos09] Jan Bosch. From software product lines to software ecosystems. In Dirk Muthig und John D. McGregor, Hrsg., *SPLC*, Jgg. 446 of *ACM International Conference Proceeding Series*, Seiten 111–119. ACM, 2009.

[CHRM09] M. Clausen, J. Hatje, J. Rathlev und K. Meyer. Eclipse RCP on the Way to the Web. In *ICALEPCS 2009 – Proceedings*, Seiten 884–886, 2009.

[Cus10] Michael Cusumano. Technology strategy and management: The evolution of platform thinking. *Commun. ACM*, 53(1):32–34, 2010.

[dR01] Jim des Rivières. How to Use the Eclipse API, 2001. `http://www.eclipse.org/articles/article.php?file=Article-API-Use/index.html`, letzter Abruf 8.10.2010.

[Fow02] Martin Fowler. Public versus Published Interfaces. *IEEE Software*, 19(2):18–19, 2002.

[GB03] Erich Gamma und Kent Beck. *Contributing to Eclipse: Principles, Patterns, and Plug-Ins*. Addision-Wesley, Boston, 2003.

[Mar06] Klaus Marquardt. Patterns for Plug-ins. In *Pattern Languages of Program Design 5*, Seiten 310–335, Addison-Wesley, Upper Saddle River, NJ, 2006.

[RAP] o.V. *Rich Ajax Platform (RAP)*, Projekt-Website. `http://www.eclipse.org/rap/`, letzter Abruf 8.10.2010.

[Rat08] Jörg Rathlev. Plug-ins: an Architectural Style for Component Software. In Ralf Reussner, Clemens Szyperski und Wolfgang Weck, Hrsg., *Proceedings of the thirteenth International Workshop on Component-Oriented Programming (WCOP 2008)*, Seiten 5–9, 2008.

[SGM02] Clemens Szyperski, Dominik Gruntz und Stephan Murer. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, London, 2. Auflage, 2002.

[WHKL08] Gerd Wütherich, Nils Hartmann, Bernd Kolb und Matthias Lübken. *Die OSGi Service Platform*. dpunkt, Heidelberg, 2008.