Plug-Ins – Applikationsspezifische Komponenten

Markus Völter, voelter@acm.org, www.voelter.de
Klaus Marquardt

Erweiterbarkeit ist eines der großen Themen in der Software-Architektur. Moderne Systeme schmücken sich gerne mit diesem Attribut – und meinen damit oft ganz unterschiedliche Dinge. Tatsächlich wird Erweiterbarkeit vom Benutzer fast nie als zentrale Eigenschaft eines Systems wahrgenommen. Doch gibt es Anwendungsbereiche, in denen eine gezielte Erweiterbarkeit den essentiellen Bedürfnissen der Kunden entspricht. Diese Kunden verlangen gleichzeitig nach einer hohen, nahtlosen Integration ihrer Anwendungen, um aus dem Zusammensetzen verschiedener Funktionen Vorteile zu ziehen. Für den Spagat zwischen Flexibilität durch getrennt entwickelte Erweiterungen und nahtloser Integration in eine gemeinsame Applikation gibt es eine – gelegentlich übersehene – Art der Komponentenarchitektur: Plug-Ins.

Komponentenarchitekturen

Komponentenarchitekturen sind mit großen Versprechungen angetreten. Im Unterschied zu Klassen und Funktionen stellen sie eine gröbere Granularität der verwendeten Bausteine zur Verfügung, wodurch sie einfacher zu verstehen und zu managen sind. Durch die Entwicklung einzelner unabhängiger Komponenten mit einheitlichen Schnittstellen sollten große Systeme durch reines Zusammenstecken realisiert werden können, Applikationen von den komplizierten Pfriemeleien technischer Infrastruktur befreit werden, und letztlich die schon von der Objektorientierung versprochene Wiederverwendbarkeit Realität werden. Neben der groben Granularität ist dabei die „Separation of Concerns“, die Trennung verschiedener Zuständigkeiten, das wichtigste Konzept: die Komponenten sollen reine Businesslogik enthalten, derweil sich die Laufzeitumgebung (der Container) um technische Mühsal wie Transaktionen, Verteilung und Persistenz kümmert [*Völter et al 2002*]. Diese Aufteilung entspricht auch der Organisationsstruktur vieler Unternehmen im Enterpriseumfeld.

Die Architektur eines Systems ist die technische Antwort auf die Komplexität des Problems, kombiniert mit dem Versuch, durch die Lösung keine zusätzliche Komplexität zu erschaffen. Inzwischen haben wir uns wieder daran erinnern lassen müssen, dass Komponentenarchitekturen nicht trivial sind, und falsch eingesetzt sogar eigene Probleme erschaffen. Komplexe Anwendungen und Systeme entstehen nicht von selbst aus Komponenten, und jegliche Integration ist mit Reibungsverlusten verbunden. Nichts kann die Kunst der Architektur ersetzen [*Buschmann 2003*]. Besonders in technischen

Anwendungsgebieten hat sich als alternative Systemsicht die Plug-In-Architektur *2001*] bewährt. Anders als bei Komponentenarchitekturen wie Enterprise Java Beans [*EJB*], COM+ [COM+] oder CORBA Component Model [CCM] liegen die Grenzen der Komponenten hierbei nicht zwischen Applikation und Infrastruktur, vielmehr wird die Applikation selbst in Einzelteile zerlegt, diese dann abgetrennt und bei Bedarf wieder in das System hineingesteckt – daher der Name Plug-In.

Bekannte Beispiele

Die populärsten Anwendungen, die sich Plug-Ins zunutze machen, sind Internet-Browser wie Netscape und Internet Explorer. Webbrowser erkennen während des Laden einer Webseite, dass ein Plug-In benötigt wird. Das Plug-In wird dann gegebenenfalls installiert, und der Browser übergibt die weitere Verarbeitung und Kontrolle an das Plug-In. Das geschieht für den Benutzer oft unsichtbar und ohne dass die Applikation dafür verlassen werden müsste.

Weniger dynamisch werden die Plug-Ins von Applikationen wie Adobe Acrobat oder Rational Rose verwendet. Die Vollversion zum PDF-Reader wird durch eine Reihe von Plug-Ins zu einem nützlichen Tool – es gibt Plug-Ins zum Editieren, für das Anheften von Kommentaren, etc. Diese Plug-Ins werden an der Applikation registriert und bekommen Platz für ihre eigenen Menu-Einträge, sind also für den Benutzer als Funktionen erreichbar. Die Plug-Ins arbeiten dann auf dem Datenmodell der Applikation und ergänzen es unter Umständen sogar. Interessant ist, dass Adobe zunächst nicht alle Plug-Ins innerhalb des Projektes entwickelt hat, sondern auch später erstellte Plug-Ins betreut und schließlich als integrale Bestandteile verkauft.

Ähnlich ist die Situation im Modellierungswerkzeug der Firma Rational: Die Plug-Ins werden hier Add-Ins genannt und erfüllen Aufgaben wie Code-Generierung, HTML- oder XMIExport. Je nach Lizenz werden mehrere Add-Ins bereits mitgeliefert. Jedes Add-In wird an der Applikation installiert und dort mit Hilfe eines eigenen Add-In-Managers aktiviert. Add-Ins arbeiten ebenfalls mit dem Datenmodell der Applikation, können dieses sogar erweitern und eigene GUI-Elemente mitbringen (zusätzliche Menueinträge und Tab-Reiter), die an von der Applikation definierten Stellen eingeklinkt werden. Wie Adobe benutzt Rational die Plug-In-Schnittstelle auch für eigene Funktionen. Anders als bei Adobe liegt hier die Add-In Schnittstelle und das Datenmodell jedem Benutzer offen, so dass jeder eigene Add-Ins entwickeln und vertreiben kann. Als Beispiel sei hier das XMI-Export Add-In der Firma Unisys erwähnt.

Die derzeit populärste Anwendung mit Plug-In Architektur ist das Eclipse-Projekt [*Eclipse*]. Eclipse ist eine Plattform zur Entwicklung von IDEs. Der Kern von Eclipse ist eine Management-Umgebung für Plug-Ins, die eigentliche Funktionalität wird von den einzelnen Plug-Ins bereitgestellt. Eclipse definiert die Schnittstellen zwischen dem Kernsystem und den Plug-Ins, und stellt eine eigene Umgebung zur Entwicklung weiterer Plug-Ins zur Verfügung (Abbildung 1). Inzwischen lebt das Open-Source Projekt davon,

dass es Plug-Ins für Anwendungen gibt, die sich vermutlich nicht einmal die Initiatoren hätten träumen lassen: Codegeneratoren, Datenbank-Browser, UML Werkzeuge, Wikis. Besonders interessant an der Eclipse-Interpretation des Plug-In-Konzeptes ist, dass hier die Plug-Ins auf einander aufbauen können. Jedes Plug-In gibt bei der Installation bekannt, welche anderen Plug-Ins es als vorhanden voraussetzt, und kann aber seinerseits wieder Schnittstellen (Extension Points) für weitere, noch spezifischere Plug-Ins definieren. Diese Kombinatorik würde jeden kommerziellen Hersteller vermutlich zur Verzweiflung bringen, befruchtet aber die Open-Source Community ungemein.

Beispiele aus technischen Domänen

Technische Anwendungen verknüpfen oft verschiedene Geräte miteinander. Die einfachsten Beispiele von Plug-Ins sind Gerätetreiber, die spätestens seit der Einführung von USB wirklich einfach steckbar sind (wer jemals mit CP/M oder Unix gearbeitet hat, weiß, dass es auch mühsam geht). Je höher der gewünschte Grad an Integration der Treiber ist, desto weniger sehen diese wie Treiber aus und entwickeln sich zu komplexen Anwendungen – die nur eben alleine keinen Sinn machen, sondern eine Applikation als Heimat benötigen.

Roche Diagnostics hat ein Integrationssystem für kleinere, spezialisierte medizinische Labore im Angebot, das auf einer Plug-In Architektur aufbaut. Ähnlich wie größere Labor-Informationsysteme (LIS) kann auch der 1996 eingeführte Laboratory Systems Manager LSM [*Roche*] verschiedene Analysegeräte anschließen. Er erreicht allerdings einen hohen Grad an Integration dadurch, dass er nicht nur das typischerweise an eine serielle Schnittstelle gebundene Protokoll zu verschiedenen Analysegeräten unterstützt, sondern auch auf ihre Spezifika eingeht. So gibt es für jedes angebundene Gerät ein eigenes Icon in der LaborÜbersicht, eigene Algorithmen zur Erstellung der Ladelisten, und eigene Bildschirme, die dem Labortechniker einen grafischen Zustand des Automaten anzeigt.

Einer nahtlosen Integration hat sich auch das Anästhesiesystem Zeus der Dräger Medical verschrieben [*Dräger*]. Der integrierte Arbeitsplatz für Anästhesisten kombiniert verschiedene Vitalparameter und lebenserhaltende Funktionen miteinander, wie zum Beispiel Beatmung, Gasdosierung, EKG und Medikamentendosierung. Die einzelnen Funktionen werden als Plug-In in die Applikation integriert. Mit Hilfe unterschiedlicher Plug-In Konfigurationen lässt sich eine ganze Familie zusammengehöriger Produkte erstellen. Hier bedarf einzelne Zusammenstellung der gesetzlichen Zulassung und kann nicht dem Kunden überlassen werden – andererseits ermöglicht die dadurch erzwungene In-House Entwicklung eine viel engere Integration der Plug-Ins. Das Zusammenstecken ist zum Beispiel nicht an Bildschirm-Bereiche gebunden ist, sondern wird von einem umfangreichen Kernsystem verantwortungsbasiert gesteuert. Insbesondere sieht die Zeus-Architektur ein Integrations-Plug-In vor (siehe Abbildung 2), das vorhandene Plug-Ins für Sensoren und Aktoren intelligent miteinander verknüpft und so einen Arbeitsplatz ausmacht, der für den Anwender tatsächlich mehr als die Summe der Einzelteile bedeutet.

Was sind denn nun Plug-Ins?

Ganz kurz: Plug-Ins sind Ergänzungen zu einer Applikation, die von dieser vorhergesehen sind, die sie aber nicht selbst mitbringen kann oder will. Ein Plug-In deckt dabei eine fachliche Funktion so vollständig ab, dass keine andere Komponente des Systems irgendetwas spezifisches darüber wissen müsste. Für die dazu nötigen Schnittstellen ist die Applikation verantwortlich.

Plug-Ins sind Komponenten im Sinne von unabhängigen Bausteinen, die aber in einer anderen Dimension „Separation of concern" betreiben als die gängigen Komponentenarchitekturen. Eine Plug-In Architektur trennt das System zunächst innerhalb der Anwendung in immer gleiche Funktionsgruppen, die unterschiedlich implementiert werden können. Innerhalb der jeweiligen Plug-Ins wird dann auch wieder nach technischen Gesichtspunkten getrennt. Die Plug-Ins einer Applikation tun im Prinzip alle das gleiche – zumindest in der Außensicht, wie zum Beispiel bei Druckertreibern oder Browser-Plug-Ins. Es bleibt aber unbenommen, für verschiedene Aufgaben unterschiedliche Arten von Plug-Ins zu definieren.

Gemeinsam ist allen Plug-Ins, dass sie speziell für eine Applikation (wie Eclipse oder Acrobat) geschrieben sind und nur in deren Kontext ablaufen können. Sie arbeiten mit den Daten der Anwendung auf spezifische Weise und fügen sich nahtlos in das vorhandene Konzept und (so ein GUI vorhanden ist) Look and Feel ein. Allerdings gibt es erhebliche Unterschiede in der Implementierung der Plug-In Schnittstelle und im Grad der Freiheit, die dem Plug-In gelassen wird.

Ähnlich wie Komponenten mit ihrem Container, sind Plug-Ins mit ihrer Applikation über einen gegenseitigen Vertrag gekoppelt (siehe Abbildung 3). Dieser von der Applikation definierte Vertrag zwingt die Plug-Ins zur Registration und zur Einhaltung bestimmter Erwartungen, und definiert andererseits, worauf sich die Plug-Ins zur Erledigung ihrer Aufgabe stützen dürfen. Ein wichtiger Unterschied zu Komponenten: Plug-Ins wissen nichts von einander und bauen nicht aufeinander auf. Das gilt konsequent auch für komplexe Anwendungen wie Eclipse und Zeus, hier allerdings mit der graduellen Abweichung, dass Plug-Ins einer höheren Integrationsstufe auf die darunter liegenden zugreifen können.

Plug-Ins bestehen aus einer Reihe von zusammengehörigen Einzelteilen, die nur in ihrer Gesamtheit sinnstiftend sind. Daher müssten wir eigentlich den Begriff Plug-In Package verwenden. Ein Plug-In Package zur Integration eines neuen Analyseautomaten in ein Labor bringt z.B. einen eigenen Prozeß für das Kommunikationsprotokoll mit, eine Anzahl Algorithmen zur Probenverteilung und Unterstützung der Qualitätssicherungsverfahren (über Strategy-Pattern realisiert [*Gamma et al 1996*]), etliche gerätespezifische Dialoge, sämtliche Texte für alle unterstützen Sprachen, ein Icon für die Laborübersicht – und aus Sicht des LSM letztlich auch das Analysegerät selbst. Damit ist jedes Plug-In hier eine vollständige Repräsentation eines bestimmten Analysegeräts, so dass die eigentliche

Applikation keine spezifischen Kenntnisse darüber benötigt und sich auf die Definition der Integrationspunkte beschränken kann – der Punkte, an denen das System den generischen Arbeitsablauf verlässt und die Kontrolle an ein für diese Aufgabe registriertes Plug-In übergibt.

Der Vertrag

Der Vertrag, den die Applikation für ihre Plug-Ins vereinbart, ist das Kernstück der Plug-In Architektur. Im Unterschied zu gängigen Komponenten-Architekturen enthält der Vertrag das globale Wissen über die Domäne, ähnlich einem Framework [*Fayad et al 2000*]. Zunächst sind keine technischen Restriktionen zu beachten – die technischen Mittel werden von der Applikation definiert und können auf Komponentenarchitekturen aufsetzen, ein eigenes Framework definieren, oder (und das ist der häufigste Lösungsansatz) eine Hybridlösung mit verschiedenen Techniken implementieren. Im Zeus beispielsweise gehören zu dem Plug-In Vertrag ein spezifisches GUI-Framework, ein medizinisches Applikations-Framework, deploybare Komponenten, eigene Prozesse zur Gerätekommunikation, sowie eine Anzahl von "Kleinigkeiten" wie Resource-Dateien für Sprachunterstützung, HTML-Dateien für die Online-Hilfe, und die Definition der für das Plug-In erhältlichen Optionen.

"Der Vertrag enthält das globale Wissen über die Domäne…", das klingt erst einmal kompliziert. Für Außenseiter ist es das auch; diese Domänen-Spezifika sind auch der Grund, weshalb man in absehbarer Zeit kein „Handbuch über Plug-In Architekturen“ im Buchhandel finden wird. Für die Hersteller der Applikation und der Plug-Ins, die sich in der Anwendungsdomäne gut auskennen, ist diese Hürde hingegen niedriger. So kennt sich jeder Rose Add-In Entwickler im Meta-Modell der UML aus, und jeder Analysegerätehersteller kann mit Patienten, Aufträgen, Proben und Resultaten umgehen und weiß, was Rilibäk ist.

Die Beschreibung der Domäne ist nichtsdestotrotz der wohl schwierigste Teil. Sie muß so spezifisch sein, dass die Anwendung alle Informationen sammeln kann, die zur Integration der installierten Plug-Ins notwendig sind. Andererseits muß sie so allgemein gehalten werden, dass sie keine Annahmen über die spezifischen Features und interne Funktionsweise bestimmter Plug-Ins macht, die diese eventuell nicht erfüllen können. Hier liegt letztlich das Kernwissen und damit der Geschäftserfolg der Hersteller.

Um diesen Spagat hinzubekommen, bedienen sich Plug-In Applikationen verschiedener Tricks. So gibt es typischerweise verpflichende und optionale Vertragsbestandteile: die verpflichtenden sind zum sinnvollen Betrieb nötig, die optionalen hingegen erlauben einzelnen Plug-Ins, ihre Spezifika im System auch unterzubringen. Häufig dürfen Plug-Ins das Datenmodell der Applikation erweitern entsprechend ihrer Bedürfnisse. So können sie von der allgemeinen Infrastruktur profitieren und ihre Funktionen anbieten – allerdings ohne die Möglichkeit, von irgend einem anderen Bestandteil des Systems verstanden zu werden. Plug-In Architekturen kommen ohne diese zentrale Intelligenz aus.

Das lässt sich wieder am bestem am Beispiel erklären: in Rational Rose können Add-Ins das Datenmodell erweitern und eigene Tabs installieren, die diese zusätzlichen Informationen anzeigen und bearbeiten lassen. Dadurch, dass Modell und Datensicht hierbei in der gleichen Plug-In Package liegen, steht dem Benutzer die Funktion vollständig und an seine Bedürfnisse angepaßt zur Verfügung. Gleichzeitig kann das Plug-In auf die Implementierung der Dienste der Applikation zurückgreifen und braucht keine eigene Infrastruktur zu errichten.

Der Vertrag muß jedoch auch eine Reihe von technischen Dingen regeln. Zu diesen gehören Installation, Registration, Lebenszyklus der Plug-Ins, Dienste und Prozessmodel, und nicht zuletzt die Gestaltung des grafischen User Interface (GUI) – sofern vorhanden. Abbildung 4 zeigt die wichtigsten Aspekte des Vertrag im Detail.

- Installation und Registration legen fest, woran die Applikation überhaupt merkt, dass ein Plug-In verfügbar ist, und welche Schritte notwendig sind, um es zum Leben zu erwecken.
- Im Lebenszyklus kann ein dynamisches Laden und Entladen sowie die Kardinalität von Plug-In Instanzen geregelt sein – dabei kann vollständige Dynamik erzielt oder auch gerade verhindert werden, wie z.B. in medizinischen Anwendungen sinnvoll.
- Die Dienste, die dem Plug-In zur Verfügung stehen, sind von essentieller Bedeutung. Das beginnt bei scheinbaren Trivialitäten wie Fehler- und Exceptionhandling und geht über das Prozessmodel, z.B. für die Kommunikation mit angeschlossenen Geräten, bis hin zur Frage, auf welche Dinge ein Plug-In Zugriff hat: andere Plug-Ins, Teile der Host-Applikation, oder das Betriebssystem, etc.
- Die GUI-Gestaltung ist oft das Kernstück des Vertrags. Müssen die Plug-Ins eine bestimmte Bibliothek benutzen, um Homogenität zu erreichen, oder dürfen sie sogar nur bereitgestellte Objekte parametrisieren (wie in der Menu-Struktur in Rose)? Entspricht jedem Plug-In ein privates Stückchen Bildschirm (wie in Eclipse), mehrere zweckgebundene Stückchen, die von der Applikation in entsprechenden Szenarien bzw. Use-Cases aktiviert werden, oder wird das private Fachwissen der Plug-Ins benutzt, um die generische Anzeige der Applikation mit Sinn zu füllen (wie im Zeus)?

Zusammenspiel von Applikation und Plug-In am Beispiel des Zeus GUI

Abbildung 5 zeigt einen Ausschnitt aus dem Zeus GUI: eine Druckkurve aus der Beatmung sowie die derzeit aktuellen Werte und eine Historie des Atemwegsdrucks. Wie kommen die Messwerte bis zur Anzeige, und wer ist für was zuständig?

Die Applikation (Olymp) stellt die fundamentalen Typen wie Parameter, ValueData und TraceData sowie eine Datenbank für Echtzeit-Meßwerte zur Verfügung.

Das Ventilations-Plug-In wird hier unterteilt in ein Extension Plug-In für die Kommunikation mit dem Prozessor, der die eigentliche Messwerterfassung vornimmt, und ein Extension Plug-In, das die Anzeige unterstützt. Das Kommunkation-PlugIn empfängt die Daten und stellt sie in die Datenbank ein.

Das Olymp-Kernsystem erlaubt dem Anwender die Konfiguration der auf dem Bildschirm angezeigten Daten, gemäß der Anzeige-Elemente, die die Applikations-PlugIns registriert haben, und gesteuert durch Leitlinien aus dem Integrations-PlugIn für den Zeus. Es stellt eine GUI Bibliothek zur Verfügung mit aktiven Elementen zur synchronen Darstellung über alle angeschlossenen Plug-Ins. Diese lesen entsprechend der Konfiguration periodisch aus der Datenbank aus und stellen die Kurven dar, damit ein gleichmäßiger Vorschub gewährleistet ist. Bei der Darstellung einzelner Werte rufen sie das zur Kommunikation korrespondierende Anzeige-PlugIn auf, das auf Besonderheiten prüft und zum Beispiel Markierungen an der Kurve anbringt. Obwohl das Olymp-System die Information über diese Marker transportiert hat, kennt es deren Bedeutung nicht und wüßte keine angemessene Reaktion oder Darstellung einzuleiten.

Wenn das Kommunikations-PlugIn einen Alarm in das Olymp-System einstellt, sorgt die Applikation für die Verbreitung durch das System. Verschiedene Benutzer können sich als Observer [*Gamma et al 1996*] auf einzelne Alarme registrieren lassen, und so kann das entsprechende Anzeige-PlugIn den alarmierten Messwert farbig hinterlegen.

Nahtlose Integration: Integrationspunkte

Wie nahtlos diese Integration auf den Benutzer wirkt, hängt von der Zahl der Integrationspunkte und damit von der Komplexität des Interfaces zwischen Plug-In und der Applikation ab. Die Integrationspunkte sind Bestandteil des gegenseitigen Vertrags Sie bestimmen, in welcher Weise die Applikation überhaupt erweiterbar ist, und wie eng die Ergänzung mit dem Applikationskern zusammenspielt.

Aus pragmatischen Erwägungen unterscheiden wir zwischen komplexen und einfachen Integrationspunkten, die entsprechend von den Extension Plug-Ins bzw. Extension Utilities einer Plug-In Package gefüllt werden (siehe Abbildung 4). Extension Plug-Ins sind vollwertige Komponenten mit ausführbarem Code, die eigene Strategien zum Ausfüllen ihrer Funktionalität bereitstellen. Dem gegenüber steuern Extension Utilities keine Intelligenz bei, sondern liefern Plug-In spezifische Information über generische Dateitypen, die direkt von der Applikation verarbeitet werden können. So ist das Starten eines Kommunikationsprozesses ein Beispiel für die Aktivierung eines Extension Plug-Ins, das Darstellen eines Icons oder das Laden einer Resource-Datei hingegen die Aktivierung eines Extension Utilities. Diese unterschiedlichen Bestandteile der Plug-In Package benutzen auch jeweils verschiedene Bestandteile des Plug-In Vertrags.

Damit landen Plug-Ins in der Mitte zwischen verschiedenen Welten. Ein Teil ihres Lieferumfangs, die Plug-In Utilities, beschränkt sich auf Konfiguration und einfache Information. Ein anderer Teil, die Plug-In Extensions, entsprechen von ihrer Funktionsweise den Hot Spots eines Frameworks [Pree 2000], und sind technisch oft gängigen Komponenten ähnlich; tatsächlich bedienen sich Plug-In Applikationen häufig kommerzieller Lösungen.

Die Technik der Integrationspunkte, ihre Art und Anzahl sind so unterschiedlich wie die Landschaft der Applikationen. Eine überschaubare Anzahl trägt zur Verständlichkeit der Architektur bei, eine Vielzahl ermöglicht weitergehende Integration. Internet Browser beispielsweise leben von ihrem schmalen Interface. Im Zeus wiederum stehen für jeden Anwender-relevanten Datentyp (wie Messwerte, Alarme) eigene Integrationspunkte zur Verfügung, so dass eine intelligente Auswertung möglich wird. Zudem gibt es eine standardisierte Menu- und Dialogstruktur, in die sich Plug-Ins in vorgegebener Granularität einklinken können. Insgesamt kommen so viele Dutzend Integrationspunkte zusammen, damit der Arbeitsplatz für den Anwender eben nicht mehr aus sichtbar zusammengesteckten Einzelteilen besteht.

Plug-In Applikationen machen ohne eine Füllung ihrer Integrationspunkte nur bedingt Sinn. Sie lassen sich zwar ausführen, und Browser und Modellierungswerkzeug bieten auch noch sinnvolle Features – an den technischen Anwendungen oder Eclipse verliert man aber schnell das Interesse, ein Laborsystem ohne angeschlossene Analyseautomaten ist sinnbefreit.

Was unterscheidet Plug-Ins von Komponenten?

Im Prinzip lösen Komponenten ähnliche Problem wie Plug-ins. In beiden Fällen geht es um Erweiterbarkeit – die Möglichkeit, Applikationsfunktionalität zusammenzustecken“. Es gibt jedoch auch einige Unterschiede:

- Im Falle von Plug-In Architekturen sind die Hostapplikationen oft auch schon ohnePlug-Ins sinnvoll. Plug-Ins erweitern typischerweise die Funktionalität um bestimmte Aspekte (wie beispielsweise das PDF-PlugIn für Netscape). Im Falle von Komponenten ist das anders. Der Container, der ja sozusagen der Hostapplikation entspricht, verrichtet ohne installierte und aktivierte Komponenten keine sinnvollen Dienste.
- Komponenten stellen abgeschlossene Funktionsbausteine dar. Eine Komponente kann üblicherweise unabhängig von anderen Komponenten installiert werden. Sie bringt alles mit was sie benötigt. Bei Plug-Ins findet eine viel engere Integration statt. Plug-Ins können die Hostapplikation bzgl. verschiedenster Aspekte erweitern, die nicht so schön lokalisierbar sind wie bei Komponenten. Die Verträge zwischen Plug-Ins und der Hostapplikation sind daher auch spezifischer als bei Komponenten und Containern.

- Gängige Komponenteninfrastrukturen (EJB, COM+, CCM) trennen die Funktionalität auf zwischen technischen Belangen (Container) und Anwendungslogik (Komponenten). Plug-In Architekturen trennen in erster Dimension verschiedene Bereiche der Anwendungslogik, und erst in der zweiten Dimension (durch verschiedene Integrationspunkte) auch Technik und Anwendungswissen.

- Plug-Ins stellen üblicherweise Erweiterungsmöglichkeiten bezüglich einer ganz speziellen Applikation zur Verfügung, sie sind domänen- bzw. applikationsspezifisch. Komponenten und Container sind dahingegen üblicherweise viel generischer – daher sind auch die Schnittstellen schmaler und die Integration nicht so eng.

- Plug-Ins sind vollständige Erweiterungen, die ihr gesamtes Fachwissen über mehrere technischen Ebenen verteilt mitbringen können. Diese thematische Abgeschlossenheit fehlt anderen Komponenten-Architekturen. Plug-Ins können einzeln vertrieben und installiert werden, und ermöglichen dadurch ein eigenes Lizensierungsmodell

Zusammenfassung

Anhand einiger Beispiele haben wir beleuchtet, was die Kernpunkte einer Plug-In Architektur sind, und welche Charakteristika sie von anderen erweiterbaren Architekturen unterscheiden. Eine Plug-In Architektur nimmt im System eine primäre Trennung vor zwischen verschiedenen Bereichen der Fachlichkeit. Die eigentliche Applikation ist reduziert auf grundlegende Kenntnisse der Anwendungsdomäne sowie die Fähigkeit, gezielt auf fachliche Plug-Ins zurückzugreifen. Zentrale Punkte dieser Architektur sind der Plug-In Vertrag, der von der Applikation definiert wird, und hierbei insbesondere die Art und Anzahl der Integrationspunkte. Anstelle einer "Separation of Concerns" bei anderen Komponentenarchitekturen könnte man also hier von einer "Separation of R Responsibilities" sprechen.

Wann ist eine Plug-In-Architektur ratsam?

Eine Plug-In Architektur sollte in Erwägung gezogen werden, wenn die Art der Erweiterbarkeit durch die Anwendungsdomäne klar umrissen ist, unabhängige Entwicklung und Pflege der Bestandteile wichtig sind, und eine größere Zahl von Integrationspunkten den Kundennutzen erhöht. Von Vorteil ist es, wenn die bestehende Organisationsform sich entlang der funktionalen Bestandteile und fachlichen Verantwortungen orientiert, da dann die Schnitte im System nicht gegen die Organisation stehen und ärgerliche Reibungsverluste verhindert werden können.

Ein letztes Wort zum Aufwand und zur Projektorganisation. Derzeit sind Plug-In Architekturen immer noch handgemacht, wenn auch unter Verwendung estehender Software-Strukturen. Daher muss der erreichbare Nutzen in vernünftigem Verhältnis zum

Aufwand stehen. Das ist meist erst dann der Fall, wenn mehrere substanzielle Plug-Ins fest geplant sind und weitere mittelfristig hinzukommen werden. Zudem ist es gerade in frühen Phasen der Entwicklung sinnvoll, die Applikation und die ersten Plug-Ins von demselben Team entwickeln zu lassen. So kann Erfahrung aufgebaut werden, welche Integrationspunkte tatsächlich sinnvoll und nötig sind und wie sie technisch aussehen sollen.

Literatur

Buschmann 2003	Frank Buschmann: Art of Architecture - Über die Kunst gute Software- Architekturen zu bauen, OOP 2003
CCM online	http://www.omg.org/technology/corba/corba3releaseinfo.htm
COM+	David Iseminger (ed): COM+ Developers' Reference Library. Microsoft Press 2000
Dräger	online: http://www.draeger.com/de/MT/pud/produkte/op/arbeitsplaetze/zeus/zeus.jsp
Eclipse	online: http://eclipse.org
EJB	online: http://java.sun.com/products/ejb/