Anwendungsintegration mit Generated Adaptive Frameworks

Auszug aus OOP 2001 – Proceedings

Spezielle Ausgabe für die OMG Information Days

Paris 19. Februar
München 20. Februar
Zürich 21. Februar
Wien 07. März

Anwendungsintegration mit Generated Adaptive Frameworks

Rüdiger Schilling
Delta Software Technology GmbH
57392 Schmallenberg
http://www.delta-software-technology.com/
rschilling@d-s-t-g.com

Kurzfassung. Die Integration der Anwendungssysteme ist mehr denn je zum kritischen Erfolgsfaktor des IT-Betriebs geworden - und das gilt nicht nur für die E-Business-Projekte, die in aller Munde sind. Während die Mehrzahl der EAI-Konzepte sich mit der Integration von Standardsoftware (ERP, CRM etc.) beschäftigt, geht es im Folgenden hauptsächlich um die Integration von Individualsoftware, von "Legacy"-Anwendungen. Es wird dargestellt, wie mittels *Generated Adaptive Frameworks*, eines methodischen Ansatzes mit strikter Schichtenarchitektur, Entwurfsmuster-Techniken und modernen Generatorkonzepten, diese Aufgabe gelöst wird. Erfahrungen mit SCORE/Integration Suite der Delta Software Technology bestätigen die Erwartungen an dieses Konzept.

1 Die Aufgabe

Enterprise Application Integration (EAI) ist das aktuelle Schlagwort zum Thema Anwendungsintegration. Dahinter verbergen sich eine ganze Reihe unterschiedlicher Ansätze und Lösungen. Angetrieben durch die Anforderungen von E-Business und E-Commerce steht zunächst die Integration bestehender Anwendungen für das Internet und im Internet im Vordergrund. Obwohl sie keinem allgemeinen Standard folgen, erleichtern die in der Regel vorhandenen und festgeschriebenen APIs kommerzieller Standardsoftware die Aufgabe und erlauben den Einsatz vorgefertigter Adapter. Das ist auch einer der Gründe, warum viele EAI-Produkte sich auf diesen Bereich konzentrieren. Die Integration individueller Anwendungssysteme wird dagegen meist recht stiefmütterlich behandelt. Dabei sind es häufig die individuellen Kernanwendungen, die für die Funktionsfähigkeit eines ganzen Unternehmens verantwortlich sind und zudem oft entscheidend zur Marktpositionierung beitragen.

Während bei Standardsoftware die Schnittstellen bereits vorgegeben sind, kommt es bei individueller Software darauf an, geeignete Schnittstellen erst zur Verfügung zu stellen - d.h. diese Anwendungen erst integrationsfähig zu machen. Dieses ist nicht etwa eine einmalige Problemlösung, eine Art *Bug-Fix*, sondern eine andauernde Aufgabe für die Software-Entwicklung und –Wartung.

Ein Unternehmen kann nicht alle Anwendungen auf einmal erneuern, sondern nur Stück für Stück, Anwendung um Anwendung. Neue und alte Anwendungen müssen koexistieren und kooperieren. Daraus ergibt sich ein völlig neues Bild der Software-Landschaft. Bisher war diese zweigeteilt in Neu-Entwicklung und Softwarepflege. Es wurden Anwendungen hochgezogen, dann für die Nutzung freigegeben, was gleichzeitig auch den Beginn der Anwendungspflege bedeutete.

Der Betrachtungswinkel bei Beschaffung, Betrieb und Unterhalt der Informatiklösungen eines Unternehmens erweitert sich von der Einzelanwendung auf das ganze Portfolio. Die Aufgabe, Anwendungen neu zu entwickeln und zu unterhalten bleibt, aber daneben sind neue Aufgaben entstanden, die an Bedeutung zunehmen:

- Anwendungen weiterentwickeln;
- Anwendungen plattformübergreifend integrieren und
- Software wiederverwenden.

1.1 Stovepipes und ihre Vermeidung

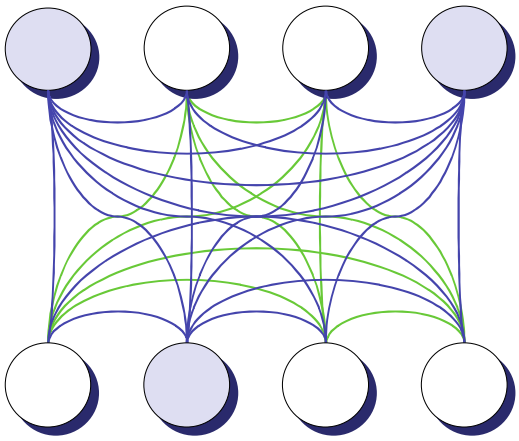

Abbildung 1: Stovepipe System

Aus der einschlägigen Literatur und der Werbung für EAI- oder Middleware-Lösungen sind sie nicht wegzudenken: Die Stovepipes. Sie sind Synonym für eine ineffiziente Punkt-zu-Punkt-Integration, die zu Komplexität und Systemkosten der Größenordnung N * N führt (N steht für die Anzahl der Knoten). Da sind zunächst nur zwei oder drei Anwendungen zu integrieren, die Probleme existieren noch gar nicht oder sind noch nicht sichtbar, die Anzahl der Schnittstellenabbildungen (A nach B, B nach A, A nach C usw.) ist beherrschbar. Mit der Einführung von E-Business-Systemen wird aus der Integration plötzlich eine Aufgabe, die Auswirkungen auf das Wohl und Wehe des ganzen Unternehmens hat. Angenommen in einem Unternehmen existieren nur fünf Anwendungen, die nach dem Punkt-zu-Punkt-Prinzip integriert sind, dann gibt es bereits bis zu 20 Schnittstellenabbildungen!

Wenn nun zur Unterstützung der E-Business-Bestrebungen nur weitere zwei Anwendungen hinzu kommen, kann sich die Zahl der Abbildungen leicht verdoppeln. Bedenkt man, dass die Integration der ersten Anwendungen sich vermutlich über einen Zeitraum von mehreren Jahren entwickelt hat, dann ist anzunehmen, dass aus den E-Business-Ambitionen in diesem Unternehmen vorläufig nichts wird.

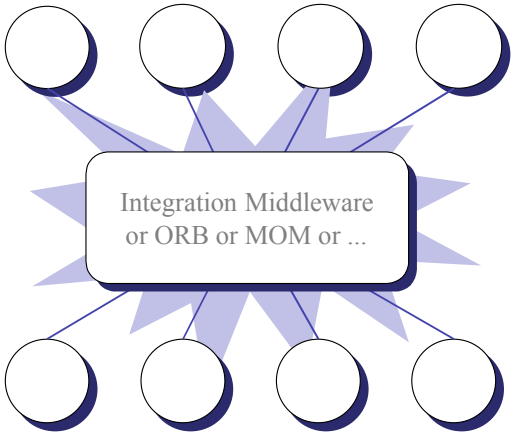

Abbildung 2: And suddenly ... a miracle occurs

Der Einsatz intelligenter EAI-Middleware hat u.a. das Ziel, die Komplexität heterogener Anwendungsnetze beherrschbar zu machen und die Entstehung neuer Stovepipes zu verhindern. Die Darstellung wird dann immer mit dem Hinweis auf die Reduktion der Schnittstellen-Komplexität auf die Größenordnung N verbunden. Diese Angabe ist meist nur richtig, soweit sie die technischen Schnittstellen zur Kommunikation betrifft. Die weit komplexere Abbildung der Syntax und gar Semantik der Anwendungsschnittstellen wird häufig weit weniger einheitlich gelöst.

Von wenigen Ausnahmen abgesehen, basieren die meisten EAI-Lösungen auf dem Konzept der Integration Server, manchmal auch Integration Broker genannt. Gemeinsam

ist all diesen Lösungen, dass es eine zentrale Instanz gibt, die Nachrichten empfängt, übersetzt und an die jeweiligen Zielanwendungen weiterreicht, die Antworten übersetzt etc.

Die Regeln für die Übersetzung der Nachrichten und die Steuerung des gesamten Systems werden in einem zentralen Repository o.ä. abgelegt und zur Ausführungszeit interpretiert. Entscheidend für die Beurteilung dieser Lösungen ist die Frage, ob dem Ganzen ein Referenzmodell zugrunde liegt oder nicht. Gibt es das Referenzmodell nicht und kann auch keines unterlegt werden, dann ist das Stovepipe-Problem nicht gelöst, sondern nur in den Server verlagert. Das Ergebnis sind "Canned Stovepipes" - Stovepipes in Dosen. Das ganze System hängt erheblich von der Verfügbarkeit des Servers ab und die Zentralisierung kann einen Engpass mit Performance-Problemen nach sich ziehen.

Die bestmögliche Lösung des Stovepipe-Problems besteht in der Integration auf Basis von Referenzmodellen. Es wird nicht jeweils von einer Anwendung zur anderen integriert, sondern von jeder Anwendung zum gemeinsamen Bezugspunkt (CPR – Common Point of Reference). So sind die Anwendungen weitgehend voneinander isoliert, Änderungen an den Schnittstellen einer Anwendung führen nicht zur Instabilität des gesamten Systems.

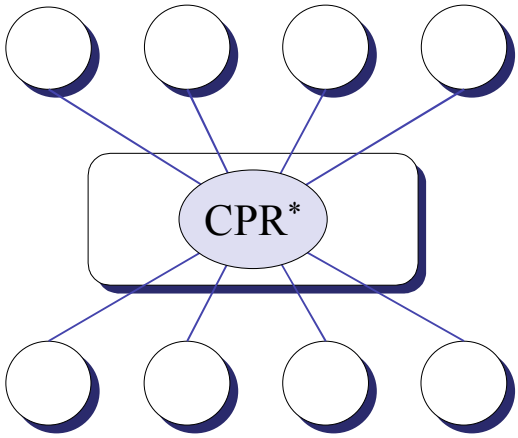

Abbildung 3: Integration with Reference Model

1.2 Lose vs. enge Kopplung

Ein Diskussionspunkt bei der Software-Integration ist die Frage, wie eng die Kopplung zwischen den Komponenten sein soll. Die beiden Extreme sind einerseits eng gekoppelte Systeme (*tightly coupled*) mit strenger Typenprüfung zur Compile-Zeit (z.B. basierend auf CORBA-IDL) und andererseits lose gekoppelte Systeme (*loosely coupled*) mit Metadatenprüfung zur Laufzeit (z.B. basierend auf XML oder CORBA-DII). Bei näherer Betrachtung stellt sich schnell heraus, dass es sich dabei gar nicht um sich ausschließende Alternativen handelt, sondern um Konzepte, die sich ergänzen. So wird in [Brown 1998] bei der Lösung des *Stovepipe Anti-Pattern* unterschieden zwischen vertikalen Schnittstellen und horizontalen Schnittstellen. Vertikale Schnittstellen sind solche innerhalb eines Systems, zwischen Sub-Systemen, die eng gekoppelt werden sollten. Horizontale Schnittstellen sind solche zwischen getrennten Systemen - möglicherweise über Unternehmensgrenzen hinweg; diese werden lose gekoppelt. Die lose gekoppelten Schnittstellen können dem *Adapter*- oder *Fassaden-Muster* folgend implementiert werden.

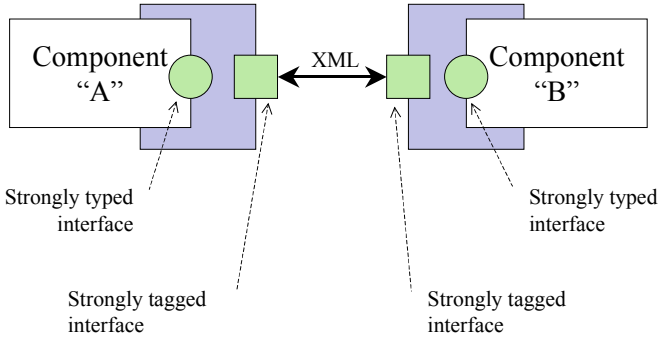

Abbildung 4: "2-Level-Interface"

Ein ähnliches Konzept verfolgt [Herzum 2000] mit einem *2-Level Interface*.

Es ist übrigens ein Irrtum, zu glauben, dass lose gekoppelte Systeme ohne Modellierung auskommen – die Konsistenzprüfung ist lediglich auf die Laufzeit verschoben (daher *strongly tagged interface*).

2 Eine Integrationsarchitektur

2.1 Komponenten-Technologie und Anwendungs-Integration

Components are for composition

So kurz und prägnant hat C. Szyperski die Zielsetzung der Komponenten-Architektur umschrieben. Wenn nun aber Komponenten dazu geschaffen sind, ohne zusätzlichen Aufwand zu größeren Systemen zusammengefügt zu werden, dürfte es zukünftig keine Integrationsprobleme mehr geben – mindestens dann nicht, wenn alle Anwendungen aus Komponenten bestehen. Abgesehen davon, dass schon der letzte Teilsatz dieser Überlegung schieres Wunschdenken darstellt, ist dieses Konzept auch aus einem weiteren Grund nicht so einfach in die Praxis umzusetzen: Komponenten benötigen eine entsprechende Infrastruktur, z.B. Application Server. Diese basieren leider auf unterschiedlichen Technologien, womit die Chance auf portable, universell einsetzbare Komponenten drastisch sinkt. Selbst wenn im Rahmen von internationalen Standardisierungsmaßnahmen verschiedene technologische Ansätze unter einen Hut gebracht werden, z.B. durch das neue CORBA Component Model (CCM) der OMG, kann man davon ausgehen, dass irgendjemand aus nachvollziehbaren marktpolitischen Gründen ausschert, so wie jüngst Microsoft mit der Ankündigung der .NET- Strategie. Wir müssen also Szyperskis schöne Formulierung um einen hässlichen aber realistischen Zusatz ergänzen:

*Components are for composition**	**in a given infrastructure*

Welche Rolle können dann Komponenten bei der Anwendungs-Integration überhaupt spielen? Komponenten können dank ihrer Grundeigenschaften leichter verteilt, ausgetauscht und weiterentwickelt werden. Daher sind sie in einer gegebenen Infrastruktur bezüglich ihrer Integrierbarkeit allen anderen Ansätzen deutlich überlegen. Die zunehmende Verwendung der Komponententechnologie bewirkt, dass sich die Integrationsaufgaben im Laufe der Zeit verändern werden. Ein komponentenbasiertes Anwendungssystem hat klare Vorteile bei der Integration mit anderen ebenfalls komponentenbasierten Anwendungen. Es bleibt das Problem der Infrastrukturabhängigkeit zu lösen. Geschickte Verwendung von Isolation Layers (siehe Anti-Patterns) können diese weitgehend beseitigen.

Über den Weg des Componentizing ist es möglich, vorhandene Anwendungs-Systeme und –Teilsysteme nachträglich in Komponenten umzuwandeln und so die Integration auf indirektem Wege zu bewerkstelligen, mit dem zusätzlichen Vorteil, dass der nachfolgend erläuterte Konflikt auf der Infrastrukturebene von vornherein ausgeschlossen ist. Natürlich braucht man leistungsfähige Werkzeuge, um diesen Prozess zu automatisieren. Die methodischen Grundlagen dazu können u.a. die weiter unten erläuterten Design Patterns liefern.

2.2 Schichten und Aspekte

Bereits 1976 hat Dijkstra die Forderung *Separation of Concerns* formuliert. Die strikte Verfolgung dieses Gedankens ist ein wesentlicher Schlüssel für eine robuste Integration von Anwendungen. Innerhalb des *Generated Adaptive Frameworks* Konzepts geschieht dieses auf zweierlei Art:

1. Durch die Einteilung in Schichten, mit genauer Zuordnung von Aufgaben. Diese Schichten können als Ganzes auch zur Laufzeit ausgetauscht werden.
2. Durch die isolierte Beschreibung technologischer und applikatorischer Aspekte, die erst zum Zeitpunkt der Generierung automatisch zusammenfügt werden (*Code Weaving*).

Die erste Schicht, die ein Anwendungsmodul umhüllt ist ein *Application Adapter*, "Component Frame" genannt. Der Component Frame macht die Operationen nach außen sichtbar. Zudem implementiert er die in der Komponentenspezifikation als *„Mapping“* bezeichnete *Abbildung* zwischen den Operationen und den Methoden des Funktionskerns. Wenn ein Client eine Operation der Komponente aufruft, wird das Mapping ausgeführt. Zu diesem gehört auch der Methodenaufruf des Funktionskerns.

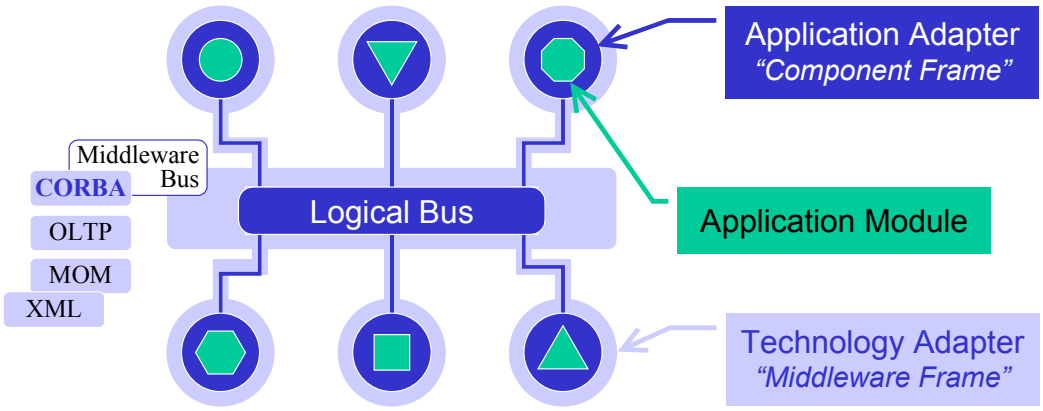

Abbildung 5: Layer Architecture

Der *Component Frame* bietet „neutrale“ Operationsschnittstellen an, wobei der "Logical Bus" dem *CPR* im Abschnitt "Integration mit Referenzmodell" entspricht.

Die nächste Schicht ist ein *Technology Adapter*, "Middleware Frame" genannt. Die Verbindung zwischen *Component Frame* und der Middleware, welche die Umgebung der Komponente bildet, stellt der *Middleware Frame* her. Alle Middleware-Aufrufe sind im Middleware Frame verborgen und damit nach dem Konzept der Isolation Layers vom Rest der Komponente getrennt.

Wenn eine Komponente von einer Middleware-Umgebung zur anderen migriert werden soll, muss nichts von Hand geändert werden. Lediglich die *Middleware Frames* müssen für die neue Umgebung neu generiert werden.

Genau genommen kann der *Middleware Frame* aus mehreren Schichten bestehen, wenn z.B. die im Abschnitt "Lose vs. enge Kopplung" genannten *2-Level Interfaces* realisiert werden sollen.

Nicht alle Funktionen lassen sich sinnvoll und vor allem effizient in Schichten darstellen, es gibt immer wieder Aufgaben die "quer" zu den übrigen Funktionen implementiert werden müssen, das gilt z.B. für Transaktionslogik oder Laufzeitoptimierung. Eine relativ neue Technik zur Bewältigung derartiger Probleme wird *Aspect Oriented Programming (AOP)* genannt. Dabei werden die einzelnen Aufgaben (Aspekte) unabhängig voneinander beschrieben, und ein Generator übernimmt die Arbeit den Code "abzumischen". Dieser Vorgang wird *Weaving* genannt.

2.3 Proxies überwinden die Sprachgrenzen

Proxies helfen, die Kluft zwischen der objektorientierten und der nicht-objektorientierten Welt zu überbrücken. Ein Proxy vertritt die in der nicht-objektorientierten Welt angesiedelte Server-Komponente bei der Client-Komponente in der Objektwelt. Proxies sind also Dolmetscher, welche die Kommunikation zwischen den beiden Welten ermöglichen: jede Seite spricht in ihrer Sprache und wird doch von der anderen verstanden. Auch die Proxies werden aus den Komponentenspezifikationen generiert – in der gewünschten objektorientierten Sprache Java, C++ oder Visual Basic.

Das Proxy ist ein *genaues Abbild der Operationsschnittstelle* der Server-Komponente. Es enthält aber keine Anwendungsfunktionen: diese sind im Funktionskern der Komponente implementiert, deren Stellvertreter das Proxy ist. Dieses erlaubt dem OO-Programmierer, die Komponente so anzusprechen, als wäre sie eine in seiner Sprache geschriebene Klasse mit zugehörigen Methoden. OO-Programmierer und Komponentenbauer müssen sich nicht mehr über die technischen Details der Implementation unterhalten, sondern nur noch über den Inhalt der Operationen bzw. Methoden.

Werden die Proxies generiert, ist sichergestellt, dass sie immer nach den gleichen Prinzipien und Regeln erzeugt werden.

3 SCORE/Integration Suite

Im vergangenen Jahr hat die **Delta Software Technology** das Produkt SCORE/Integration Suite eingeführt. Die wichtigsten Elemente sind das Component Repository, der Component Manager und spezielle Generatoren.

Sämtliche Informationen über Anwendungen und Anwendungskomponenten sowie deren Schnittstellen, die mit **SCORE/Integration Suite** verwaltet werden, sind im **Component Repository** im XML-Format abgelegt. Das Component Repository wird dabei nicht nur für interne Zwecke verwendet (als Basis für die Generierung der Frames), sondern steht darüber hinaus für administrative Aufgaben, Auswertungen und andere Zugriffe zur Verfügung.

Der integrierte **Component Manager** greift direkt auf das Repository zu. Alle wichtigen Informationen über Komponenten, Schnittstellen, Methoden, Klassen oder zur Versionierung werden zur interaktiven Bearbeitung vorgehalten. Außerdem steuert er alle Generierungsabläufe.

SCORE/Integration Suite verwendet eine von der Delta Software Technology Gruppe neu entwickelte Generatortechnologie. Diese erlaubt die effiziente Generierung der Frames. Sie ist überaus flexibel und für beliebige Plattformen und Zielumgebungen einsetzbar. Die Generierung kann vom Anwender auf seine individuellen Bedürfnisse

selbst zugeschnitten und optimiert werden. Dabei werden nicht nur bekannte Entwurfsmuster unterstützt, es können auch eigene hinzugefügt werden.

Die bisherigen Versionen erlauben die client-seitige Integration von Java-, C++- und VisualBasic-Anwendungen. Server-seitig werden C++, C- und COBOL-Anwendungen unterstützt. Die Middleware-Unterstützung beginnt bei OLTPs (z.B. TUXEDO und TX-Series) und umfasst mittlerweile ORBs (z.B. WebLogic oder ORBIX). Außerdem wird eine XML/SOAP-Anbindung angeboten.

4 Entwurfsmuster und Anwendungsintegration

4.1 Was sind Entwurfsmuster? - Eine erste Definition

Ab etwa 1995 hat sich eine zunehmende Anzahl von Autoren mit dem Thema Entwurfsmuster beschäftigt, wobei insbesondere der Aspekt des Design Reuse betont wird.

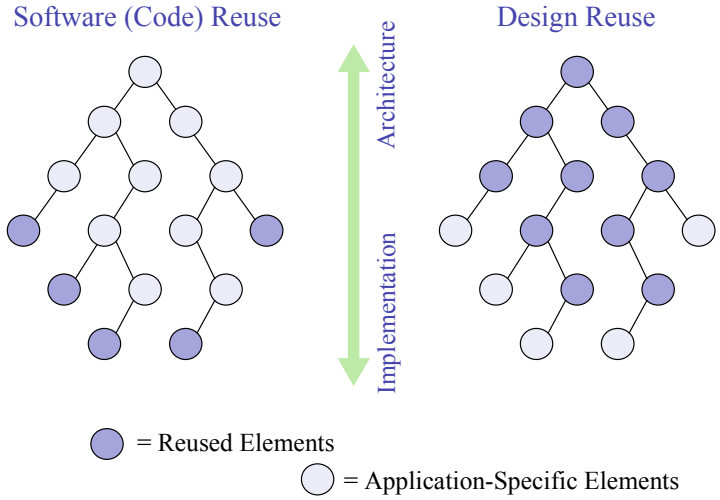

Abbildung 6: Design Pattern – Reusable Design

Die sicher bekannteste Veröffentlichung dazu ist das Buch *Design Patterns* aus dem Jahr 1994 (dt. Übersetzung [Gamma 1995]).

Software-Entwurfsmuster stellen ein derart mächtiges Instrument dar, dass sie natürlich auch bei der Anwendungs-Integration eine große Rolle spielen (müssen).

4.2 Anti-Patterns

1998 erschien das Buch *Anti-Patterns* [Brown 1998]. Es hat sich zum Ziel gemacht, die Erscheinungsbilder und Ursachen von Mustern problematischer oder fehlerhafter Software zu beschreiben, seien diese so entworfen worden oder "einfach nur entstanden". Interessant wird das Ganze dadurch, dass auch die zugehörigen Reparatur-Lösungen sog. *Refactored Solutions* dargestellt werden. Genau um diese Lösungen geht es im Zusammenhang mit der Integration. Mindestens für die im Folgenden aufgeführten Anti-Patterns gilt, dass die Konzepte zur Behebung der Probleme auch dazu verwendet werden können, sie gleich (evtl. von vorneherein) zu vermeiden.

4.2.1 Anti-Pattern: Stovepipe System

Stovepipe-Systeme entstehen durch die "ad hoc"- und Punkt-zu-Punkt-Integration von Anwendungen und Subsystemen ohne eine ausreichende Abstraktion der gemeinsamen Schnittstellen und Funktionen.

Die Lösung bildet eine Komponentenarchitektur, die einen flexiblen Austausch von Software-Modulen erlaubt. Dazu wird ein abstraktes Subsystem modelliert, dass weniger und einfachere Schnittstellen hat als die einzelnen Subsysteme. Als Ergebnis gibt es zwei Kategorien von Schnittstellen – vertikale und horizontale. Als vertikale Schnittstellen werden die Schnittstellen innerhalb der Subsysteme bezeichnet, die auch für die weitere Entwicklung der Subsysteme benutzt werden. Als horizontal werden die generalisierten Schnittstellen zwischen den Anwendungen bezeichnet.

Implementiert wird dieses Konzept durch Adapter, Bridges und Fascades, siehe auch 4.3 "Design Patterns (nach Gamma....)“.

4.2.2 Anti-Pattern: Vendor Lock-In

Es geht bei diesem Anti-Pattern im Prinzip darum, dass ein Software-Projekt oder ein ganzes IT-System abhängig ist von einem Hersteller, einem Produkt oder einer speziellen Technologie, die nicht einem allgemeinen Standard entspricht.

Mit Integrationsprojekten hat dieses Anti-Pattern auf zweierlei Art zu tun.

In dem nach wie vor jungen Markt der EAI-Lösungen ist die Gefahr groß, von neuen Technologien (und den entsprechenden Herstellern) abhängig zu werden. Das ist in diesem Fall besonders kritisch, weil die Integration nicht irgendein Teilsystem, sondern am Ende große Teile des gesamten IT-Systems und für das Unternehmen lebenswichtige Bereiche betrifft.

Vorhandene Anwendungen, die auf unterschiedlichen Infrastruktur-Systemen basieren (z.B. Datenbank- oder Middleware-Software), können eine Integration erheblich behindern. Ebenso kann es vorkommen, dass z.B. die Kommunikations-Middleware der bestehenden Anwendungen nicht mit dem Integrationssystem kompatibel ist.

Die Lösung dazu heißt Isolation Layer und ist der Lösung des Stovepipe Anti-Pattern sehr ähnlich. Es geht auch in diesem Fall darum, eine Schnittstellen-Abstraktion zu finden. Während aber im vorangehenden Fall quasi eine Schicht oberhalb der Anwendungen benötigt wird, stellen Isolation Layer eine Schicht unterhalb, zwischen Anwendung und Infrastruktur dar.

4.3 Design Patterns (nach Gamma ...)

Zwar sind die Entwurfsmuster im gleichnamigen Buch in erster Linie für den Entwurf neuer Applikationen gedacht, dennoch sind einige davon auch für die Anwendungs-Integration von erheblicher Bedeutung. Darüber hinaus ist es leicht möglich, weitere Integrationsmuster (Entwurfsmuster für die Integration) nach dem gleichen Schema zu definieren.

Wirklich interessant wird dieses Konzept der Entwurfsmuster, wenn es durch ein entsprechendes Werkzeug unterstützt wird. So kann ein ganzer Katalog von Lösungs-mustern für Integrationsaufgaben entstehen, und das Werkzeug sorgt dafür, dass diese Muster immer wieder gleichförmig, mit gleicher Qualität eingesetzt und angewandt werden.

4.3.1 Adapter

Das Adaptermuster dient dazu, voneinander unabhängige Module (Komponenten, Objekte, Klassen) zusammenarbeiten zu lassen.

Im Rahmen der Anwendungs-Integration finden sich zwei Arten von Adaptern: Technologie-Adapter und Anwendungs-Adapter, die auch Wrapper genannt werden.

Technologie-Adapter dienen zur Anbindung von Anwendungs-Modulen an die Infrastruktur, z.B. Middleware oder Integrationssysteme.

Anwendungs-Adapter werden benötigt, um die Schnittstellen der Anwendungen zu kapseln und über eine Middleware allgemein verfügbar zu machen.

4.3.2 Façade

Eine Fassade bietet eine einheitliche Schnittstelle zu einer Menge von Schnittstellen eines Subsystems (einer Anwendung). Sie definiert eine generalisierte Schnittstelle, welche die Benutzung des Subsystems vereinfacht. Das Fassademuster ist besonders geeignet zur Realisation des Konzepts der horizontalen Interfaces zur Auflösung des Stovepipe System Anti-Pattern.

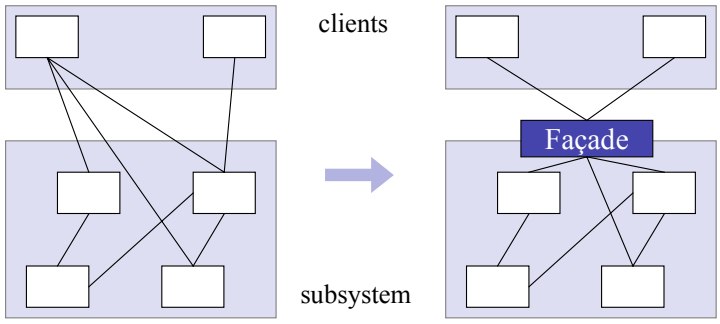

Abbildung 7: Design Pattern „Façade“

4.3.3 Decorator

Die Aufgabe des Dekorierermusters ist es, einem Modul (Komponente, Objekt) neue Funktionen und Eigenschaften hinzuzufügen, ohne das Modul selbst zu ändern. Es stellt damit auch eine Art Wrapper dar.

Im Rahmen der Integration werden Dekorierermuster benötigt, um vorhandene Anwendungen funktional zu erweitern. Der Dekorierer leitet Operationen an die Anwendung weiter und führt möglicherweise vor oder nach dem Weiterleiten zusätzliche Operationen aus, z.B. die Initialisierung von globalen Daten vorher oder das Speicher der Daten nachher.

4.3.4 Proxy

Das Proxy-Entwurfsmuster kontrolliert den Zugriff auf eine Komponente mit Hilfe eines vorgelagerten Stellvertreterobjekts.

Es gibt eine Reihe verschiedener Typen von Proxies, so stellt ein **Remote-Proxy** einen lokalen Stellvertreter für eine Komponente in einem anderen Adressraum dar; ORBs verwenden dieses Prinzip. Ein **virtuelles Proxy** erzeugt teure Objekte auf Verlangen, es werden z.B. einer Client-Anwendung zunächst nur Kopfinformationen zur Verfügung gestellt und die Details erst auf Verlangen nachgeliefert. Ein **Sprach-Proxy** dient zur Überbrückung von Sprachbarrieren; ein Problem, das sich regelmäßig beim Übergang von Java zu anderen Programmiersprachen ergibt.

Alle drei Proxy-Arten spielen bei der Anwendungs-Integration eine Rolle, wobei natürlich auch Kombinationen vorkommen. So kann bei der Verbindung eines Java-

Clients via CORBA zu einem Cobol-Daten-Server ein kombiniertes Sprach- und Remote-Proxy sehr nützlich sein.

4.3.5 Andere Muster

Auch andere Entwurfsmuster z.B. aus der Kategorie der Verhaltensmuster können für die Integration relevant sein.

5 Zusammenfassung

> *"By 2004, more than 70 per-cent of e-business applications not integrated with back-office systems will fail in meeting business expectations"*
>
> Gartner Group, Sept. 2000

Bedenkt man, dass diese Vorhersage immerhin vier Jahre in die Zukunft reicht – im Internet-Zeitalter eine Ewigkeit – bedeutet das nicht nur, dass Erfolg oder Nicht-Erfolg von E-Business Projekten entscheidend von der richtigen Integration der Kernanwendung abhängt, sondern auch davon, dass die Anwendungsintegration langfristig und nachhaltig angelegt sein muss.

Genau dieses Ziel verfolgt der "Generated Adaptive Frameworks"-Ansatz:

Ein methodisch fundiertes Framework bildet den Ausgangspunkt. Das zentrale Element dieses Frameworks ist eine Schichtenarchitektur, die eine klare Trennung vollzieht zwischen den notwendigen semantischen und syntaktischen Abbildungen einerseits und der Anbindung an Middleware, System- und Sprachplattformen andererseits. Diese Architektur erlaubt die leichte Adaption neuer Anwendungen und Funktionen, und selbst ein Wechsel der Middleware-Technik ist ohne nennenswerte Eingriffe möglich. Die Aufteilung in die verschiedenen Aspekte technologischer und applikatorischer Natur ist nicht nur die Grundlage einer reibungslosen Integration unterschiedlicher Anwendungen, sondern auch die Grundlage modernen Anwendungs-Designs per se.

Neben der Schichtenarchitektur und aspektorientierten Ansätzen stellen Entwurfsmuster ein weiteres modernes Element der Softwareentwicklung dar, das ebenfalls auch zur Softwareintegration herangezogen werden kann. Den Ausgangspunkt bilden dabei *klassische* OO-Pattern wie Adapter, Proxies und Fassaden, die auch auf komponentenbasierte Modelle übertragen werden können. Spezielle, eventuell projektspezifische, Muster kommen hinzu.

All dieses lässt sich natürlich auch manuell in die Praxis umsetzten - wirklich attraktiv sind diese Konzepte aber nur zusammen mit geeigneten, generierenden Werkzeugen, die in der Lage sind, die Umsetzung weitgehend zu automatisieren. Der scheinbare Gegensatz zwischen Flexibilität und einfacher Handhabung einerseits und hoher Performance und Stabilität andererseits wird ebenfalls durch den Einsatz moderner Generierungstechniken auf ideale Weise aufgelöst.

Diese Kombination von Schichtenarchitektur, aspektorientierten Ansätzen, Entwurfsmustern und entsprechenden Generatoren, führt zu adaptiven – d.h. sich selbst anpas-

senden – Lösungen, im Unterschied zu lediglich adaptierbaren – d.h. passiv anpassbaren – Lösungen.

Mit SCORE/Integration Suite stellte die Delta Software Technology Gruppe im abgelaufenen Jahr 2000 erstmals ein Produkt vor, das konsequent und erfolgreich diese Konzepte umsetzt.

Literatur

[Brown 1998] Brown William J. et al, Anti Patterns. Wiley Computer Publishing, 1998, Software Architecture AntiPatterns

[Gamma 1994] Gamma Erich et al, Design Patterns, Addison-Wesley, 1994

[Herzum 2000] Herzum Peter, Business Component Factory, Wiley Computer Publishing, 2000

[Szyperski 1998] Szyperski C., Component Software Beyond Object-Oriented Programming, Addison-Wesley, 1998