Systemübergreifende Software-Architektur: Erfahrungen und Thesen

Johannes Siedersleben, Stephan Kurpjuweit

sd&m Research
Thomas-Dehler-Str. 27
D-81737 München
siedersleben@sdm.de
kurpjuweit@sdm.de

Abstract: Die Prinzipien der Software-Architektur sind wohlbekannt und akzeptiert: Trennung der Zuständigkeiten, Denken in Komponenten und Schnittstellen, durchgängige Behandlung von Fehlern und Ausnahmen und noch ein paar andere. Trotzdem wäre es vermessen zu behaupten, dass wir die Architektur eines einzelnen großen Systems beherrschen.

Dieser Vortrag befasst sich mit dem noch schwierigeren Problem der systemübergreifenden Software-Architektur: Wie kooperieren Systeme, die verschiedene Teams mit unterschiedlicher Technik unabhängig voneinander gebaut haben? Dabei sind u.a. folgende Fragen zu beantworten: Sind die klassischen Regeln der Software-Architektur weiterhin gültig? Sind sie ausreichend oder brauchen wir neue Richtlinien? Sind moderne Web-basierte Techniken hilfreich oder hinderlich?

1. Software-Architektur im Großen: ein großes System

Die Grundlagen der Software-Architektur (z.B. [P78]) sind auf jeder Ebene das Denken in Komponenten und Schnittstellen, die Trennung der Zuständigkeiten und die Berücksichtigung von Notfällen. Die Komponente (nicht die Klasse) ist die wesentliche Einheit des Entwurfs, der Planung und der Integration [BCK03]. Schnittstellen sind abstrakte Dienste, deren Syntax, Semantik und nichtfunktionalen Eigenschaften im Sinn eines Vertrags verbindlich festgelegt sind. Komponenten kommunizieren grundsätzlich über Schnittstellen, niemals direkt. So läuft jede Komponenten in einem durch Schnittstellen definierten Kontext; benachbarte Komponenten, die diese Schnittstellen implementieren, sind austauschbar.

Bei der Trennung der Zuständigkeiten hat sich in der Praxis die Trennung von Anwendungssoftware (A-Software) und Techniksoftware (T-Software) als besonders nützlich herausgestellt [S03].

Die Behandlung von Notfällen ist bis zum heutigen Tag ein Stiefkind der Software-Architektur. Wir empfehlen die strikte Trennung zwischen Fehlern der Anwendung und Notfällen. Notfälle sind selten (kommen aber vor) und erfordern völlig andere Maßnahmen [S03b].

2. Software-Architektur im ganz Großen: viele große Systeme

Obwohl wir kaum die Architektur eines einzelnen großen Systems beherrschen, entwerfen und betreiben wir Architekturen im ganz Großen: unternehmensweite oder unternehmensübergreifende Netze von Systemen, die verschiedene Teams mit verschiedenen Techniken zu verschiedenen Zeiten gebaut haben oder gerade bauen.

Die vielfach verwendete Analogie "Softwarelandschaft" zur Bezeichnung der Gesamtheit aller DV-Lösungen eines Unternehmens ist dabei irreführend. Während eine Landschaft sich im wesentlichen nicht ändert, stellen die system- und unternehmensübergreifenden DV-Lösungen permanente Grossbaustellen dar: Um mit den strategischen Unternehmensentscheidungen Schritt zu halten, werden ständig neue Systeme integriert, alte Systeme stillgelegt oder an neue Anforderungen angepasst. Der Begriff "Softwarestadt" ist daher treffender: Wie richtige Städte sind Softwarestädte historisch gewachsene, dynamische, extrem langlebige und heterogene Gebilde. Oft gibt es dabei nicht einmal eine zentrale Kontrollinstanz, die für den systematischen Architekturentwurf der Systemstadt – also für die Stadtplanung – zuständig ist. Die Prinzipien für den Entwurf von Einzelsystemen (siehe Abschnitt 1) gelten bei der Städteplanung uneingeschränkt weiter.

3. Nichtfunktionale Eigenschaften von Schnittstellen

Die Architektur eines einzelnen Systems[1] wird im Wesentlichen bestimmt durch zwei nichtfunktionale Eigenschaften, nämlich Performance und Wartbarkeit: Wie lange dauert ein Aufruf; wie viele Aufrufe können pro Zeiteinheit verarbeitet werden? Wie viel Aufwand erfordern zukünftige Änderungen? Bei systemübergreifenden Architekturen dominieren weitere Eigenschaften: Robustheit, Sicherheit, Kosten und Verfügbarkeit. Oft sind Systeme zu integrieren, die selten verfügbar sind, langsam oder gar nicht antworten, oder im schlimmsten Fall falsche Antworten liefern. Dadurch entstehen viele neue Fehlersituationen, die alle zu planen und zu behandeln sind. Neben der eingeschränkten Testbarkeit ist dies die wohl größte Herausforderung von Integrationsprojekten.

[1] zumindest eines einzelnen betrieblichen Informationssystems

4. Enge, lose und flexible Koppelung

Ein Generalfehler, der beim Architekturentwurf immer wieder gemacht wird, ist die fehlende Kontrolle der Abhängigkeiten zwischen den verschiedenen Systemteilen. Die Idee der flexiblen Entkoppelung war und ist daher enorm lehrreich für den Entwurf tragfähiger Softwarearchitekturen – dies gilt für einzelne große Systeme und erst recht für ganze Systemstädte.

Systemteile (oder Komponenten) können unterschiedlich gekoppelt sein. Gefährlich sind implizite Koppelungen über Annahmen, die die Komponenten übereinander machen und stillschweigend verwenden. Falsche Annahmen können gemacht werden über die Existenz einer Komponente, deren Verfügbarkeit, Korrektheit, und Zuverlässigkeit. Entkoppeln heißt, möglichst wenige Annahmen über die anderen Systemkomponenten zu treffen und auf alle Eventualitäten vorbereitet zu sein.

Ein extremer Ansatz zur Entkoppelung wird bei Microsoft zum Bau autonomer Geschäftsanwendungen mit Web Services entwickelt [SS03]: Eine Systemstadt besteht aus autonomen Software-Festungen (software-fiefdoms), die unabhängig voneinander agieren. Die Festungen machen keinerlei Annahmen übereinander und kommunizieren nicht einmal direkt. Die Kommunikation läuft über Emissäre, die von den Software-Festungen mit dem Auftrag entsandt werden, geeignete Partner zu suchen. Die Festung verlässt sich dabei auf gar nichts – nicht einmal, dass die Emissäre zurückkommen oder gültige Daten zurückliefern. Eine Festung macht keine Zusicherungen und begibt sich in keinerlei externen Abhängigkeiten - verteilte Transaktionen über mehrere Festungen sind undenkbar.

Was sind die Konsequenzen dieses Ansatzes? Die Festungen sind maximal voneinander entkoppelt und somit weitgehend unabhängig von einem bestimmten Kontext. Diesen Vorteil erkauft man durch extrem reduzierte Performance, höhere Komplexität und höheren Realisierungsaufwand, da Mechanismen zur Kompensation – etwa zum Auffinden alternativer Partner - implementiert und ausgeführt werden. Zwischen der maximalen Entkoppelung und der totalen Koppelung über Standardaufrufe gibt es ein ganzes Spektrum verschiedener Koppelungsabstufungen. Die Aufgabe des Software-Ingenieurs besteht darin, zwischen je zwei Komponenten die geeignete Nähe zu finden nach dem Prinzip der *flexiblen Koppelung*: Soweit entfernt wie möglich, so eng wie nötig, damit Realisierungsaufwand und Performance im Rahmen bleiben.

5. Technologien

Wir unterscheiden vier Kategorien von Standards bzw. Produkten, die uns bei Integrationsprojekten das Leben erleichtern oder manchmal auch schwer machen.

a) Objektorientierte Verbindungssoftware (z.B. CORBA, RMI, COM). Bei der systemübergreifenden Integration geht die Bedeutung dieser Produkte zurück, weil sie eine enge Koppelung der beteiligten Systeme unterstellen bzw. herbeiführen. Objektreferenzen oder Transaktionen über Systemgrenzen hinweg binden die Systeme auf Gedeih und Verderb; diese enge Koppelung ist in der Regel unerwünscht.

b) Dienstorientierte Verbindungssoftware (z.B. SOAP, XML-RPC). Diese Standards werden bei systemübergreifenden Architekturen immer mehr zur Selbstverständlichkeit und werden objektorientierte Verbindungssoftware weitgehend verdrängen. Man muss aber sehen, dass etwa SOAP erstens ziemlich langsam ist, und dass es zweitens kaum mehr Funktionen bietet als Marshalling und Unmarshalling (also wesentlich weniger als CORBA).

c) Nachrichtenorientierte Verbindungssoftware (z.B. MQSeries) unterstützt die asynchrone, sichere Kommunikation verschiedener Systeme.

d) EAI-Produkte (z.B. SeeBeyond, Tipco). Diese Produkte sind hilfreich bei der Verwaltung der unübersehbaren Punkt-zu-Punkt-Verbindungen, sind aber noch sehr teuer und nicht unproblematisch im Betrieb.

Diese Produkte und Standards haben selbstverständlich ihren Wert, aber sie lösen nur einen kleinen Teil der Probleme systemübergreifender Software-Architektur.

6. Zehn Regeln

Die folgenden zehn Regeln fassen unsere bisherigen Erfahrungen mit Integrationsprojekten zusammen:

1. Abstrahiere jedes beteiligte System durch eine geeignete Schnittstelle.

2. Definiere Stadtviertel von benachbarten Systemen, die gemeinsam betrieben werden.

3. Entkoppele die beteiligten Systeme flexibel, um nicht beeinflussbare Änderungen dieser Systeme abzufedern.

4. Bottom-Up ist mindestens so wichtig wie Top-Down.

5. Standardisiere mit Augenmaß. Je größer die Systemstadt, desto geringer die Chancen für erfolgreiche Standardisierung.

6. Plane langfristig – wie beim Städtebau.
7. Plane ausführliche Tests; trotzdem keine Einführung ohne Fallback-Lösung.
8. Plane alle Notfälle ein; definiere und teste die jeweiligen Notverfahren.
9. Technologie kann helfen, aber erwarte nicht zuviel. Vermeide Abhängigkeiten und verlasse dich auf nichts.
10. Behandle die beteiligten Parteien mit Sorgfalt; erkenne und berücksichtige deren Interessen.

Literaturverzeichnis

[BCK03] Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice, Second Edition. Addison-Wesley, 2003.

[P78] Parnas, D.L.: Some Software Engineering Principles, Infotech State of the Art Report on Structured Analysis and Design, Infotech Internationsl, 1978.

[S03a] Siedersleben, J.: Quasar: Die sd&m Standardarchitektur Teil 1. sd&m AG, München, 2003.

[S03b] Siedersleben, J.: Errors and Exceptions: Rights and Responsibilities. Proceedings of the ECOOP 2003 Workshop on Excepton Handling in Object-Oriented Systems https://wwws.cs.umn.edu/tech_reports/listing/list-report.cgi

[SS03] Sundblad S.; Sundblad, P.: Design patterns: Architecture of an Autonomous Application. MSDN Magazine, Juli, 2003, http://msdn.microsoft.com/msdnmag/issues/03/07/DesignPatterns/