Fallstudie zur Spezifikation von Fachkomponenten

Jörg Ackermann

Von-der-Tann-Str. 42, 69126 Heidelberg, Deutschland, E-Mail: joerg.ackermann.hd@t-online.de

Zusammenfassung. Die geeignete Spezifikation einer Softwarekomponente ist eine wesentliche Voraussetzung für ihre erfolgreiche Wiederverwendung. Dieser Beitrag beschreibt die Erkenntnisse aus einer Fallstudie, in der eine Beispielkomponente spezifiziert wurde. Der Spezifikation lag der momentan diskutierte Vorschlag der GI-Arbeitsgruppe 5.10.3 zu Grunde.

Schlüsselworte: Fachkomponente; Softwarekomponente; Spezifikation; Standardisierung; Fallstudie

Die präzise und geeignete Spezifikation von Fachkomponenten ist eine wesentliche Voraussetzung, damit sich der Bau von Anwendungssystemen aus am Markt gehandelten Softwarekomponenten etablieren kann. Die Arbeitsgruppe 5.10.3. der Gesellschaft für Informatik erstellt derzeit einen Vorschlag, wie Aufbau und Inhalt einer solchen Spezifikation aussehen sollten. Für weitere Details dazu siehe das Memorandum „Vorschlag zur Spezifikation von Fachkomponenten“ (Turowski et al. 2001).

In diesem Zusammenhang erstellte ich eine Fallstudie, welche eine Fachkomponente beispielhaft spezifiziert. Dieser Beitrag beschreibt das Vorgehen bei der Fallstudie, die wichtigsten Erkenntnisse und die aufgetretenen Probleme. Die komplette Fallstudie inklusive weiterer Erläuterungen findet sich im Anhang.

1 Allgemeine Vorgehensweise und Erfahrungen

Die Fallstudie spezifiziert eine Fachkomponente „Flugticketverkauf“ anhand der Vorgaben im o.g. Memorandum. Die dort definierten Beschreibungsebenen und verwendeten Notationssprachen bilden den Rahmen für die Beispielspezifikation.

Der Vorschlag zur Spezifikation ist derzeit in Arbeit und verändert sich auch, an einigen Stellen werden noch verschiedene Alternativen diskutiert. In der Fallstudie ist es nicht möglich, allen Vorschlägen gleichermaßen gerecht zu werden. Ich habe versucht, jeweils den Alternativen mit breitester Zustimmung (sofern erkennbar) den Vorrang zu geben. Teilweise habe ich verschiedene Alternativen einander gegenübergestellt, um einen Vergleich zu ermöglichen.

Der Hauptfokus der Fallstudie lag auf Syntax, Verhalten und Abhängigkeiten der Dienste (Abstimmungsebene) der Fachkomponente. Einerseits war die Diskussion zu diesen Punkten schon am weitesten fortgeschritten und ihre Ergebnisse am genauesten beschrieben. Andererseits bilden diese Aspekte einen wesentlichen Kern der Beschreibung einer Fachkomponente. Als Konsequenz befasst sich dieser Beitrag hauptsächlich mit den Erkenntnissen auf der Syntax-, der Verhaltens- und der Abstimmungsebene.

In der Fallstudie wird die Komponente „Flugticketverkauf“ auf folgenden Ebenen spezifiziert

(in dieser Reihenfolge): Funktionsebene, Syntaxebene, Verhaltensebene, Abstimmungsebene, Leistungsebene, Administrationsebene, Terminologieebene.

Meinem Vorschlag in (Turowski et al. 2001) folgend, habe ich (soweit sinnvoll) auf allen Ebenen formale Beschreibung durch Prosabeschreibung ergänzt.

Die Spezifikation einer Komponente muss deren Außensicht vollständig beschreiben. Neben den angebotenen Diensten müssen deshalb auch die von der Komponente benötigten Dienste anderer Komponenten spezifiziert werden. Dies ist vor allem auf der Syntax-, der Verhaltens- und der Abstimmungsebene zu berücksichtigen.

In der Spezifikation sollte deutlich zwischen angebotenen und benötigten Diensten unterschieden werden. Dazu wurde in der Fallstudie folgende Notation verwendet:

NameDerFachkomponente::AngebotenerDienst()
Extern::BenötigterDienst()

Beispiele aus der Fachkomponente „Flugticketverkauf“ sind:

Flugticketverkauf::Flugreise::Anlegen()
Extern::Reisebüro::PrüfeExistenz()

Diese Notation drückt aus, dass keinerlei Aussage darüber getroffen wird, welche andere Komponente oder Komponenten die benötigten Dienste zu erbringen hat. Für die Fachkomponente gibt es lediglich eine externe Welt, die diese Dienste bereitzustellen hat.

Die Spezifikation verwendet objektorientierte Modellierungskonstrukte, obwohl die Komponente nicht objektorientiert implementiert ist. Alle beschriebenen Dienste wurden in der Form *Entität::Methode* (Beispiel *Flugreise::Anlegen*) dargestellt. Die Vorteile dieses Vorgehens werden in (Ackermann 2001a) diskutiert.

2 Beschreibung der Beispielkomponente

In der Fallstudie wird eine (Pseudo-) Komponente „Flugticketverkauf“ spezifiziert. Die beschriebene Funktionalität ist (ab SAP Web Application Server-Release 6.10) Teil einer Schulungsanwendung, mit der SAP verschiedene Technologien anhand eines einfachen betriebswirtschaftlichen Beispiels demonstriert und vermittelt.

Die Funktionalität zum „Flugticketverkauf“ existiert seit Release 6.10, wurde jedoch nicht als Komponente implementiert (deshalb der obige Begriff „Pseudo“). Sie ist jedoch weitgehend gekapselt und kann deshalb als Fachkomponente betrachtet werden. Es handelt sich um ein real existierendes Beispiel. Wichtig ist dabei, dass die Komponente unabhängig von der Fallstudie entstanden ist und nicht extra für die Fallstudie erstellt wurde. Die Komponente hat eine für die Fallstudie geeignete Komplexität, ist aber insgesamt gesehen nicht umfangreich.

Aus der Verwendung eines Beispiels aus dem SAP-Umfeld ergibt sich keine Aussage über die zukünftige Komponentenstrategie der SAP. Es besagt auch nicht, dass SAP den Entwurf des GI-Arbeitskreises 5.10.3. unterstützt.

Die Fachkomponente „Flugticketverkauf“ stellt Dienste zur Verfügung, die ein Reisebüro zum Verkauf von Flugtickets benötigt. Dazu zählen Verwaltung und Auswahl von Flugverbindungen und Verwaltung und Verkauf von Flugreisen durch ein Reisebüro. Im einzelnen werden folgende betriebliche Aufgaben unterstützt:

- Angebotserstellung für Flugreisen

- Verkauf von Flugreisen
- Verwaltung von Flugreisen
- Verwaltung von Flugverbindungen.

Die Beispielkomponente ist nicht objektorientiert implementiert und erlaubt keine komponentenübergreifende Objektübergabe.

Als Grundlage für diese Komponente wird der SAP Web Application Server (Release 6.10) vorausgesetzt.

3 Syntaxebene

Die Syntaxebene spezifiziert die Syntax aller Dienste der Fachkomponente mit Hilfe der OMG IDL. In der Fallstudie wurden zwei OMG IDL-Module definiert: das Modul „Flugticketverkauf" steht für die Fachkomponente selbst und das Modul „Extern" beschreibt die von außerhalb der Fachkomponente benötigten Dienste. Das Modul „Flugticketverkauf" enthält zwei Interfaces, welche die beiden Entitäten Flugverbindung und Flugreise repräsentieren. Die Interfaces definieren Methoden und benötigten Datentypen der jeweiligen Entitäten. Analog wurde das Modul „Extern" definiert. Außerdem enthält die Syntaxebene die Liste aller Fehlermeldungen, die von den Diensten der Fachkomponente zurückgeliefert werden.

Durch die Verwendung der OMG IDL auf der Syntaxebene und der OCL auf der Verhaltensebene entsteht ein Methodenbruch. So verwenden z.B. OMG IDL und OCL unterschiedliche Konstrukte zur Modularisierung. Dies wurde in der Fallstudie entschärft, indem ein eindeutiger Bezug zwischen den Begriffen hergestellt und dokumentiert wurde. Die Tabelle 1 zeigt, welche Konstrukte in der Fallstudie einander entsprechen.

Fachkonzept	**Beispiel**	**OMG IDL**	**OCL / UML**
Fachkomponente	Flugticketverkauf	module	package
Entität	Flugreise	interface	class
Dienst	Anlegen	operation	method

Tabelle 1: Zuordnung von Notationskonstrukten

Die OMG IDL (Version 2.4.2.) bietet an einigen Stellen nur eingeschränkte Ausdrucksmöglichkeiten. So kann man nicht einzelne Parameter einer Methode als optional deklarieren. Es ist außerdem nur schwer oder gar nicht möglich, semantisch reichere Datentypen zu definieren. Beispiele dafür sind:

- OMG IDL kennt weder Datum noch Uhrzeit. Als Konsequenz habe ich z.B. ein Datum als *fixed<8,0>* abgebildet. Dadurch müssen an verschiedenen Stellen genauere Bedingungen angegeben werden, da nicht alle achtstelligen Zahlen ein gültiges Datum darstellen. Dieser Mehraufwand entfällt, wenn das Datum als eigener Datentyp vorliegt.
- In der ABAP-Implementierung werden an verschiedenen Stellen sogenannte „Numerical Character-Typen" verwendet. Dabei handelt es sich um Character-Typen einer vorgegebenen Länge, die nur Ziffern als Zeichen enthalten dürfen. (Beispiele sind die achtstellige

Buchungsnummer oder die vierstellige Flugnummer.) Solche Typen lassen sich mit der OMG IDL nicht abbilden.

Eine weitere Schwierigkeit ergab sich bei der Fehlerbehandlung. Die OMG IDL verwendet das Konstrukt von „Exceptions“ für Ausnahmen. Die Beispielkomponente „Flugticketverkauf“ kennt allerdings keine „Exceptions“ im programmiertechnischen Sinne. Stattdessen werden Fehler- und sonstige Nachrichten gesammelt in einem standardisierten Parameter *Return* zurückgegeben. (Welche Dienste welche Nachrichten zurückgeben, wird auf der Verhaltensebene beschrieben.) Wird die Komponente mit der OMG IDL spezifiziert, müsste sie formal gesehen „Exceptions“ verstehen und verarbeiten können.

Zusammengefasst lässt sich feststellen, dass die OMG IDL mit Einschränkungen für die Beschreibung der Syntaxebene geeignet ist. Aus meiner Sicht wäre es aber sinnvoll, noch einmal Alternativen zur OMG IDL zu diskutieren.

4 Verhaltensebene

Die Verhaltensebene enthält am Beginn ein UML-Modell, welches alle wesentlichen Entitäten und deren Beziehung zueinander darstellt. Dieses Modell bildet die Grundlage für die auf der Verhaltensebene formulierten OCL-Bedingungen. Das Modell enthält zwei Pakete: „Flugticketverkauf“ und „Extern“. Die Pakete enthalten jeweils die wichtigsten Entitäten als Klassen.

Danach wurden dienstübergreifende und dienstspezifische Bedingungen als OCL-Ausdrücke beschrieben. Die ersteren sind Invarianten, die letzteren Vor- und Nachbedingungen.

Des weiteren wurden Bedingungen an die externen Dienste formuliert. Diese beschreiben, welches Verhalten die Komponente „Flugticketverkauf“ von den Diensten erwartet, die von anderen Komponenten zur Verfügung gestellt werden. Diese Bedingungen wurden möglichst allgemein gehalten, um die externen Dienste nicht unnötig einzuschränken.

Auf der Verhaltensebene haben sich die folgenden Erkenntnisse ergeben:

1. Die Verwendung eines UML-Modells in der angegebenen Form hat mehrere Vorteile und ist meiner Meinung nach sogar unverzichtbar. Einerseits macht es die Bedingungen verständlicher und erhöht die Ausdruckskraft der Beziehungen deutlich. Andererseits wären einige der Bedingungen ohne Modell gar nicht oder nur mit großen Umwegen auszudrücken. Beispiele dafür sind:

- Bedingungen an die Existenz einzelner Entitäten (siehe z.B. Anhang, 3.3.1.): In vielen Bedingungen werden Ausdrücke der Art *Flugverbindung->exists* verwendet. Dazu muss aber die Entität „Flugverbindung“ als solche definiert sein, was nur im Modell möglich ist. (Falls eine Methode *Flugverbindung::Anlegen* existiert, kann die Bedingung alternativ auf der Abstimmungsebene beschrieben werden. Dies ist aber nicht in allen Fällen möglich).
- Übertragen von Invarianten auf Vor- und Nachbedingungen von Diensten (siehe z.B. Anhang, 3.2.5): Für dienstübergreifende Bedingungen an Entitäten können Invarianten definiert werden, die dann für alle Dienste dieser Entität gelten. Ohne Modell ist dies nicht möglich, so dass diese allgemeinen Bedingungen bei jedem Dienst wiederholt werden müssten.
- Bedingungen an Daten, die nicht in der Methoden-Schnittstelle vorhanden sind (siehe

z.B. Anhang, 3.2.7): Es gibt Fälle, in denen Bedingungen an Daten formuliert werden sollen, die nicht in der Schnittstelle des betrachteten Dienstes vorhanden sind. Solche Bedingungen können ohne Modell oft nicht ausgedrückt werden.

Man beachte, dass das Modell nur ein Spezifikationsartefakt ist. Es ist ein Hilfsmittel, um die Spezifikation verständlich und ausdrucksstark zu machen. Das Modell abstrahiert von der konkreten Implementierung. Die angegebenen Klassen müssen nicht tatsächlich existieren. Bei einer Nicht-OO-Implementierung existieren ja gar keine Klassen. (Deshalb verwende ich den Ausdruck „Entität" statt „Klasse"). Um den Spezifikationsgedanken zu unterstreichen, sind im Modell alle Eigenschaften als privat deklariert. Einzige Ausnahme bilden die tatsächlich vorkommenden Dienste. In das Modell wurden nur Daten/Eigenschaften aufgenommen, über die in der Spezifikation sowieso Aussagen getroffen werden. Damit verstößt meiner Meinung nach dieses Vorgehen nicht gegen den Black-Box-Gedanken!

2. Auch wenn Vorbedingungen formuliert werden, sollte die Komponente fehlertolerant erstellt sein und bei Verletzung einer Vorbedingung einen entsprechenden Fehler zurückgeben. Die Spezifikation sollte die Information enthalten, auf welche Situationen mit welchem Fehler reagiert wird. Dazu hat sich folgendes Vorgehen als sehr praktikabel erwiesen: Zu jeder Vorbedingung erscheint eine Nachbedingung, die den entsprechenden Fehler aufführt, der bei nicht erfüllter Vorbedingung zurückgegeben wird. Zur Vereinfachung wird der Vorbedingung ein Name gegeben. Im folgenden Beispiel enthält der Exportparameter *Status* die Liste aller Fehler und Meldungen:

```
Flugticketverkauf::Flugverbindung::LiefereListe(Rbnr,..., Status)

 pre ReisebüroExistiert:
      Extern::Reisebüro::PrüfeExistenz(Rbnr) = ´X´

 post: ReisebüroExistiert = false implies
      Status->exists(st | st.Typ = ´E´ and st.Nummer = ´151´)
```

OCL erlaubt, Bedingungen einen Namen zu geben. Ich konnte aber bisher nicht ermitteln, ob es auch erlaubt ist, auf diesen Namen in anderen Bedingungen Bezug zu nehmen. Dies könnte gegebenenfalls ein Erweiterungsvorschlag für die OCL sein.

3. Es hat sich herausgestellt, dass nicht alle Bedingungen sinnvoll in OCL auszudrücken sind, die man intuitiv der Verhaltensebene zuordnen würde. Beispiele dafür sind:

- Ein Parameter enthält die zwei Felder „Währungsbetrag" und „Währung". Es soll ausgedrückt werden, dass der Währungsbetrag sich auf die im Feld „Währung" stehende Währung bezieht.
- Es soll ausgedrückt werden, dass alle angegebenen Abflug- und Ankunftszeiten jeweils lokale Zeiten sind. Außerdem ermittelt sich das Attribut „Flugdauer" als Differenz zwischen Ankunfts- und Abflugzeit unter Berücksichtigung der Zeitverschiebung. Beide Aussagen sind in Prosaform intuitiv und unmissverständlich. Um diese in OCL auszudrücken, müsste das Konzept von Zeitzonen und Zeitverschiebungen explizit in das Modell aufgenommen und beschrieben werden.

Diese Bedingungen wurden in der Fallstudie stattdessen auf der Funktionsebene formuliert.

4. Die OCL (Version 1.3) enthält noch eine Einschränkung, die meiner Meinung nach auf eine fehlende Aktualisierung der OCL zurückzuführen ist: Obwohl in UML 1.3 bei Methoden

Exportparameter erlaubt sind, ist es in OCL nicht möglich, auf diese Bezug zu nehmen. Das ist jedoch unbedingt notwendig. In der Fallstudie wurde dafür eine geeignete Notation eingeführt.

Zusammengefasst lässt sich feststellen, dass eine formale Beschreibung auf der Verhaltensebene sehr aufwändig ist. Der Vorteil der formalen Korrektheit sollte deshalb noch einmal genau gegen die Nachteile des hohen Aufwands und der eingeschränkten Verständlichkeit abgewogen werden. Die OCL selbst war als formale Sprache gut geeignet. Sie ist verständlich und mit wenigen Einschränkungen ausdrucksstark genug. Für eine höhere Einheitlichkeit und bessere Verständlichkeit sind für die Komponentenspezifikation bestimmte Konventionen (im Sinne von Best Practices) wünschenswert.

5 Abstimmungsebene

Auf der Abstimmungsebene werden Konsistenz- und Reihenfolgebedingungen zwischen verschiedenen Diensten ausgedrückt. Bei der Komponente „Flugticketverkauf“ ergaben sich dafür zwei Beispiele:

- Alle Dienste können prinzipiell parallel ausgeführt werden. Bei *Flugreise::Anlegen* ist dies jedoch nur unter bestimmten Bedingungen möglich. Werden diese nicht beachtet, ergeben sich Sperrprobleme beim Buchen der Einzelflüge.
- Alle Dienste verwenden die Services von externen Diensten. Es wird jeweils formuliert, auf welche externen Dienste während der Abarbeitung eines Dienstes der Komponente zugegriffen wird. Diese Information ist für den Komponentenintegrator notwendig.

Auf der Abstimmungsebene wird als Notation die OCL unter Ergänzung der temporalen Operatoren verwendet. In der Fallstudie verwende ich eine präzisierte Syntax, die sich an (Saake 1993) anlehnt. Der Ausdruck *after(Methode(par))* ist ein boolscher Ausdruck. Dieser ist zu genau dem Zeitpunkt wahr, in welchem der Methodenaufruf (mit den spezifizierten Parametern *par*) erfolgreich beendet wurde. Ansonsten ist er falsch. Analog dazu ist *before(Methode(par))* genau zu dem Zeitpunkt wahr, in welchem der Methodenaufruf (mit den spezifizierten Parametern *par*) gestartet wird. Ansonsten ist er falsch. Mit dieser Notation ergeben sich z.B. folgende Bedingungen:

- Der Dienst *Flugverbindung::LiefereListe* ruft während seiner Abarbeitung den externen Dienst *Extern::Reisebüro::PrüfeExistenz*.

```
Flugticketverkauf::Flugverbindung::LiefereListe(Rbnr, ...)
 post: before(Extern::Reisebüro::PrüfeExistenz(Rbnr))
              sometime_since_last
         before(Flugverbindung::LiefereListe(Rbnr, ...))
   and after(Extern::Reisebüro::PrüfeExistenz(Rbnr))
              sometime_since_last
   before(Extern::Reisebüro::PrüfeExistenz(Rbnr))
```

- Die Methode *Flugreise::Anlegen* kann nur unter bestimmten Bedingungen aufgerufen werden, solange noch ein anderer Aufruf derselben Methode abgearbeitet wird.

```
Flugticketverkauf::Flugreise::Anlegen(Rd2, ...)
 pre:    not after(Flugreise::Anlegen(Rd1, ...))
                   sometime_since_last
              before(Flugreise::Anlegen(Rd1, ...))
         implies Rd1 <> Rd2
```

(Die Bedingung nach *implies* ist in der Fallstudie deutlich komplexer. Da an dieser Stelle nur der erste Teil relevant ist, wurde sie vereinfacht dargestellt. Für die ausführliche Version siehe Anhang, 4.5.)

Ohne diese erweiterte Syntax der temporalen Operatoren könnten die angegebenen Bedingungen nicht (Beispiel 1) oder nur syntaktisch unpräzise (Beispiel 2) ausgedrückt werden.

Es ist außerdem zu beachten, dass durch die Erweiterung um temporale Operatoren einige Grundkonzepte der OCL nicht mehr gültig sind. Deshalb sollte auf der Verhaltensebene die Original-OCL und nur auf der Abstimmungsebene die OCL mit temporalen Operatoren verwendet werden. Für eine Diskussion dazu siehe (Ackermann 2001a).

In manchen Fällen ist es möglich, eine Bedingung entweder auf der Verhaltensebene oder der Abstimmungsebene zu beschreiben. Beispiel: Eine Flugreise kann nur storniert werden, wenn sie zuvor angelegt wurde und noch nicht storniert wurde. Auf der Abstimmungsebene sieht diese Bedingung folgendermaßen aus:

```
Flugticketverkauf::Flugreise::Stornieren(Rbnr, Rnr, ...)
 pre: sometime_past
           after(Flugreise::Anlegen(Rd, Rbnr, Rnr, ...))
  and not sometime_past
           after(Flugreise::Stornieren(Rbnr, Rnr, ...))
```

Alternativ kann diese Bedingung auf der Verhaltensebene formuliert werden. Dabei wird das Statusattribut (B = gebucht, C = storniert) der Flugreise verwendet:

```
Flugticketverkauf::Flugreise::Stornieren(Rbnr, Rnr, ...)
 pre: Flugreise->exists(flr | flr.Reisebüronummer = Rbnr and
                              flr.Reisenummer     = Rnr  and
                              flr.Status          = ´B´ )
 post: Flugreise->exists(flr | flr.Reisebüronummer = Rbnr and
                              flr.Reisenummer     = Rnr  and
                              flr.Status          = ´C´ )
```

Es entsteht ein Interpretationsfreiraum, auf welcher Ebene bestimmte Bedingungen auszudrücken sind. Dies sollte vermieden werden. Meiner Meinung nach sind die Zustandsattribute für solche Fälle besser geeignet. Für eine detaillierte Diskussion der Vor- und Nachteile beider Varianten sei auf die Fallstudie (Anhang, 4.6) verwiesen. Ich schlage vor, dass eine Bedingung möglichst immer auf der Verhaltensebene mit Hilfe der OCL formuliert wird. Nur wenn eine Bedingung nicht mit Hilfe der OCL formulierbar ist, sollte diese mit Hilfe der temporalen Operatoren auf der Abstimmungsebene ausgedrückt werden.

Zusammengefasst lässt sich feststellen, dass in der Fallstudie nur wenige Bedingungen auf der Abstimmungsebene zu formulieren waren. Für diese ist allerdings eine Erweiterung der Syntax der temporalen Operatoren notwendig. Mit der Erweiterung konnten die Bedingungen adäquat ausgedrückt werden.

6 Weitere Ebenen

Die Fachkomponente „Flugticketverkauf“ wurde außerdem auf der Funktions-, der Leistungs-, der Administrations- und der Terminologieebene beschrieben. Die Diskussion zu diesen Ebenen ist noch nicht soweit fortgeschritten und die Vorgaben im Memorandum waren noch nicht so klar. Deshalb können hier keine allgemeingültigen Erkenntnisse präsentiert werden. Für einen Eindruck, wie die Spezifikation auf diesen Ebenen aussieht, sei auf die Fallstudie

im Anhang verwiesen.

Ich möchte an dieser Stelle aber kurz auf ein aufgetretenes Problem eingehen. Voraussetzung für Installation und Betrieb der Beispielkomponente ist das Komponenten-Framework des SAP Web Application Servers. Eine ganze Reihe von Aspekten in der Spezifikation werden allein oder vorwiegend durch das Framework bestimmt. Dabei handelt es sich um einige der technischen Merkmale auf der Administrationsebene (z.B. Systemarchitektur) und die meisten der Merkmale auf der Leistungsebene (z.B. Verfügbarkeit). Es handelt sich also weniger um Eigenschaften der Komponente, die ich spezifizieren musste. Für die spezielle Beispielkomponente sind die bisherigen Vorschläge daher nicht immer optimal geeignet.

7 Zusammenfassung und Ausblick

Die Fallstudie hat gezeigt, dass eine Fachkomponente anhand der Vorschläge im Memorandum zufriedenstellend spezifiziert werden kann. Als Konsequenz haben sich eine ganze Reihe von Erkenntnissen ergeben, die in diesem Beitrag vorgestellt wurden und jetzt in den Vorschlag zur Spezifikation einfließen sollten.

Ein Ergebnis der Fallstudie ist aber auch, dass der Aufwand relativ hoch war, die Spezifikation zu erstellen. Er betrug ein mehrfaches vom Aufwand, welcher für die Implementierung und Dokumentation benötigt wurde. Enthalten im Aufwand sind dabei sowohl die Einarbeitung in die verschiedenen Techniken als auch die Fehler und Irrwege, die man beim ersten Mal unvermeidlich geht. Bereinigt um diese Einmal-Effekte schätze ich den Aufwand für die Spezifikation trotzdem auf mindestens den Aufwand von Implementierung und Dokumentation. Dies sollte als kritischer Punkt vermerkt werden.

Parallel zum Aufwand für das Erstellen ergibt sich auch ein recht großer Umfang der Fallstudie selbst. Diese ist etwa 60 Seiten lang, obwohl die Beispielkomponente mit 4 Diensten nicht komplex ist. Daraus ergibt sich für einen Verwender ein entsprechender Aufwand beim Lesen und Verstehen der Spezifikation.

Für die Zukunft wünschenswert wäre der Versuch, die Beispielkomponente zu verwenden. Daraus ergeben sich Erkenntnisse, wie gut eine solche Spezifikation zu verstehen ist.

Außerdem wäre es aus meiner Sicht nützlich, wenn es neben dem Memorandum eine Liste von Konventionen gibt, wie bestimmte wiederkehrende Fragen behandelt werden sollten. Dies führt zu einer größeren Einheitlichkeit und damit besseren Verständlichkeit. Erste Vorschläge dazu wurden in diesem Beitrag unterbreitet.

Literatur

Ackermann, J.: Diskussionsbeitrag zur Vereinheitlichung der Spezifikation von Fachkomponenten. In: K. Turowski (Hrsg.): Modellierung und Spezifikation von Fachkomponenten: 2. Workshop, Bamberg, 2001a.

Conrad, S.; Turowski, K.: Vereinheitlichung der Spezifikation von Fachkomponenten auf der Basis eines Notationsstandards. In: Tagungsband Modellierung 2000. St. Goar, 2000.

Saake, G.: Objektorientierte Spezifikation von Informationssystemen. Teubner Verlag. Stuttgart, 1993.

Turowski, K., et al.: Vorschlag zur Vereinheitlichung der Spezifikation von Fachkomponenten. Diskussionsvorschlag des GI-Arbeitskreises 5.10.3, 2001. URL: http://www.fachkomponenten.de. Abruf am 2001-08-03.