Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Ausarbeitung
Seminar

Jan Weinschenker

Dokumentationsverfahren und Beschreibungssprachen für Software-Architekturen

Betreuender Prüfer: Prof. Dr. Kai v. Luck

Fakultät Technik und Informatik
Studiendepartment Informatik

Jan Weinschenker

Thema der Ausarbeitung
Seminar

Dokumentationsverfahren und Beschreibungssprachen für Softwarearchitekturen

Stichworte

Softwarearchitektur Dokumentation Analyse ADL Views-and-Beyond UML ACME

Kurzzusammenfassung

Dieser Artikel führt in die Bedeutung des Architekturbegriffs in der zeitgemäßen Softwareentwicklung ein. Weiterhin wird der Views-and-Beyond-Ansatz als ein Verfahren zur Dokumentation von Softwarearchitekturen vorgestellt. Anhand einiger Beispiele wird gezeigt, wie die Modellierungssprachen UML und ACME dabei als Hilfsmittel herangezogen werden können.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Am Entwurf und der Realisierung von großen Informationssystemen sind heutzutage nicht mehr nur Informatiker beteiligt. Die Liste der Stakeholder ist lang. Als Erstes fallen einem Projektleiter und das Projektteam ein. Diese sind direkt mit der Realisierung des Systems beschäftigt. Dann gibt es noch den Kunden beziehungsweise den Auftraggeber. Doch auch innerhalb dieser beteiligten Gruppen gibt es Personen, die ganz unterschiedliche Sichten und Verständnisse vom zu entwerfenden Produkt haben.

Um ein Projekt erfolgreich abschließen zu können, bedarf es viel Kommunikation zwischen allen beteiligten Stakeholdern. Es gab und gibt im Bereich der Informatik viele Bestrebungen, diese Kommunikation zu formalisieren, um Mehrdeutigkeiten und daraus resultierende Fehler

in der Kommunikation zu vermeiden. Damit sollte die Erfolgsquote der Projekte erhöht werden. Ein Nachteil dieser Bestrebungen war, dass die entworfenen Notationen sich in der Regel nur an den Bedürfnissen der Informatiker, also der Techniker orientierten. Nicht-Techniker haben oft weder die Vorbildung noch die Zeit oder die Motivation, sich mit allzu technischer Dokumentation auseinander zusetzen.

Diese spiegelt oft lediglich die rein technische Sicht des Projekts wider. Jeder Stakeholder hat jedoch seine eigene Sicht auf das Produkt und eigene Erwartungen an die Dokumentation. Dazu vier Beispiele:

- Besonders in verteilten Systemen ist der Aspekt der Interoperabilität von Bedeutung. Schon aus der Grundstruktur eines Informationssystems muss sich dieser Aspekt ergeben und allen Beteiligten erschließen.
- Der Kunde möchte wissen und nachlesen können, wie performant das Produkt später arbeitet und wie exakt das Produkt später seine Berechnungen durchführt.
- Das Marketing möchte die besonderen Vorteile des Produkts kennen lernen, um darauf aufbauend Strategien für Werbung und Verkaufsförderung entwickeln zu können.
- Der Kunde möchte sichergestellt wissen, dass seine Investition sich auch langfristig auszahlt und zukünftige Modifikationen am Produkt einfach möglich sind. Außerdem soll sich das Produkt einfach in die bestehende Techniklandschaft des Kunden integrieren lassen.

Die genannten Aspekte lassen sich eventuell bei trivialen Problemstellungen noch rein technisch beschreiben, so dass trotzdem viele Beteiligte daraus schlau werden. Werden die Aufgabenstellungen jedoch komplexer, so können oft nur Fachleute die technische Dokumentation lesen und verstehen. Dies ist jedoch in großen Projekten nicht akzeptabel, da dort die Techniker nur einen Teil des Projektteams stellen und die Nicht-Techniker ebenfalls am Entwicklungsprozess beteiligt werden müssen, um einen Erfolg herbeizuführen.

Also müssen Ideen, Konzepte und Anforderungen unter allen Beteiligten kommuniziert werden. Voraussetzung dafür ist eine einheitliche Basis bezüglich der Begrifflichkeiten und der groben Zusammenhänge des gemeinsamen Vorhabens. Ein Mittel, um so etwas zu schaffen, ist eine gut konzipierte Dokumentation, die sich an den Bedürfnissen und Vorkenntnissen *aller* Beteiligten orientiert.

Noch bevor konkrete Arbeiten am Projekt beginnen, werden normalerweise Design- und Architekturentscheidungen getroffen. Wenn bereits in dieser frühen Projektphase alle Stakeholder in den Entwicklungsprozess eingebunden werden können, ist die Wahrscheinlichkeit groß, dass Fehler früh erkannt und beseitigt werden können. Und je früher Fehler in einem Projekt entdeckt werden, desto billiger ist deren Korrektur.

Neben der lebhaften Kommunikation unter allen Beteiligten gilt es jedoch, auch zusätzlich eine Analyse der verwendeten Architektur vorzunehmen. Grundlage dafür sind die Architekturbeschreibungssprachen (*Architecture Definition Languages - ADL*). Auch eine damit erstellte Analyse gehört als ein wichtiger Bestandteil zu jeder umfassenden Dokumentation. Zwei aktuell relevante ADLs sind ACME und xADL (siehe Garlan u. a. (2000) bzw. Dashofy u. a. (2001)). ACME wird in einem kurzen Beispiel in Abschnitt 3.3 vorgestellt.

Das folgende Thesis Outline wird eine Methode zur Dokumentation softwarelastiger Architekturen, *Views and Beyond*, vorstellen (Clements u. a., 2003). In Anlehnung an (Clements u. a., 2003) wird diese Ausarbeitung drei Sichttypen erläutern, aus denen eine Softwarearchitektur betrachtet werden kann. Dabei handelt es sich um Sichten bezüglich statischer und dynamischer Zusammenhänge sowie der Interaktion der Architektur mit ihrer Umgebung. Der zweite bedeutende Ansatz zur Dokumentation von Architekturen stammt vom Institute of Electrical and Electronics Engineers (IEEE) und hat dort die Standard-Nummer 1471-2000. Letzterer ist nicht Gegenstand dieses Thesis Outlines, da sich die Verfahren nicht grundlegend voneinander unterscheiden. Für einen Vergleich dieser beiden Verfahren siehe (Clements, 2005), für die eigentliche Beschreibung des Standards, siehe (IEEE1471, 2000).

Dieses Thesis Outline entstand in einem Projekt des Master-Programms des Studiendepartments Informatik der Hochschule für Angewandte Wissenschaften Hamburg. Ziel des Projektes ist der Entwurf und die Realisierung einer verteilten IT-Infrastruktur für einen Ferienclub. Einige der in den nachfolgenden Erläuterungen enthaltenen Beispiele sind diesem Projekt entnommen worden oder waren von ihm inspiriert.

2 Grundlagen

Dieses Thesis Outline beschäftigt sich mit der Dokumentation und der Analyse von Softwarearchitekturen. Zu Beginn soll der hier verwendete Architekturbegriff und dessen heutige Bedeutung im Bereich der Softwareentwicklung erläutert werden. Anschließend wird auf die allgemeinen Vorteile der Dokumentationserstellung in Softwareprojekten eingegangen. Am Ende dieses Abschnitts gibt es eine kurze Beschreibung des Konzeptes der Architekturbeschreibungssprachen.

2.1 Softwarearchitekturen

Die nachfolgenden Erläuterungen beziehen sich auf Architekturen, deren Komponenten im wesentlichen aus Software bestehen und sich darüber definieren. Die zugehörigen Hardwarekomponenten, also Prozessoren, Bussysteme oder Datenspeicher werden nicht tiefer gehend betrachtet. Zwei gängige Definitionen des Begriffs Softwarearchitektur lauten:

> *Die Software Architektur eines Programms oder Informationssystems ist die Struktur oder sind die Strukturen des Systems, welche Softwareelemente, die extern sichtbaren Bestandteile dieser Elemente und die Beziehungen unter ihnen beinhalten. (Bass u. a., 2003)*

> *The fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution. (IEEE1471, 2000)*

Beiden Definitionen ist gemein, dass sie Softwarearchitektur als eine abstrakte Struktur bezeichnen, die sich aus mehreren Komponenten zusammen setzt, die Beziehungen untereinander haben. Jede Komponente nimmt dabei eine klar definierte Teilaufgabe im Gesamtzusammenhang wahr. Mehr als 40 weitere Definitionen sind unter (SEI Website, 2005) nachzulesen,

sie unterscheiden sich nicht grundlegend von den beiden hier exemplarisch Genannten, auch wenn sie teilweise andere Schwerpunkte setzen. Oft wird anstelle des Begriffs *Softwarearchitektur* auch die Bezeichnung *komponentenbasiertes System* verwendet. Eine Einführung in die Softwarearchitektur als Gegenstand der aktuellen Forschung bietet (Shaw, 2001).

Eine Architektur ist, zumindest dem klassischen Verständnis nach, mehr als die Summe ihrer Einzelteile. Nach Vitruv beruht sie auf den drei Prinzipien Stabilität (*Firmitas*), Nützlichkeit (*Utilitas*) und Schönheit (*Venustas*) (Pollio, 1996). Auch wenn Virtuv schon vor über zweitausend Jahren das zeitliche segnete, hat sich sein Verständnis von Architektur noch bis heute gehalten. Danach sollte der Architekt also nicht nur den Anspruch an sich haben, dass sein Werk dem bloßen Bedürfnis, oder in der Sprache der Informatik, den funktionalen Anforderungen und den geforderten Qualitätsmerkmalen genügt. Auch das Prinzip der Schönheit sollte in seinen Überlegungen berücksichtigt werden. Andere Altmeister, wie etwa Leon Battista Alberti, sprechen in diesem Zusammenhang von Harmonie und Einklang (Borsi, 1975). Diesen Anspruch an sich und ihre Arbeit erheben auch die Softwarearchitekten. Neben aller Formalität und Technik, die oft den Prozess der Softwareentwicklung maßgeblich bestimmen, sollte ein Architekt also immer versuchen, neben den geforderten Qualitätsmerkmalen auch an die Ästhetik seines Produkts zu denken.

2.2 Dokumentation in der Softwareentwicklung

Generell ist das erstellen von Dokumentation bei Softwareentwicklern eine unbeliebte Aufgabe. Es kostet Zeit, hält vom eigentlichen Entwickeln ab. Hinzu kommt, dass sie mit jeder Änderung am Produkt aktualisiert und fortgeschrieben werden muss. Dennoch ergeben sich bei nüchterner Betrachtung viele Vorteile, die gut gemachte Dokumentation mit sich bringt. Diese wiegen in vielen Fällen die zuvor genannten Nachteile auf.

Schon in der Einleitung wurde die Bedeutung der Kommunikation in Projekten angesprochen. Wichtige Aspekte, wie einheitliche Begrifflichkeiten und die Darstellung der groben Zusammenhänge eines Projekts sind dafür unerlässlich und waren schon immer die Kernelemente guter Dokumentation. Es ist in jedem Fall Aufgabe eines Projektleiters, dafür zu sorgen, dass bei allen Beteiligten Einigkeit in diesen Punkten besteht und dass sie nicht von widersprüchlichen Annahmen bei ihrer Arbeit ausgehen.

Man kann nun versuchen, bei jedem Projektmeeting durch Diskussion eine solche Einigkeit zu erzielen. Sinnvoller ist es jedoch, einmal erzielte Einigkeiten schriftlich festzuhalten und an zentraler Stelle für alle zugänglich zu hinterlegen. Bei zukünftigen Uneinigkeiten verweist man dann einfach auf die Dokumentation, anstatt weiter zu diskutieren. In diesem Fall kann die Dokumentation sogar dafür sorgen, dass Zeit anstatt für unnötige Diskussionen für sinnvollere Tätigkeiten genutzt werden kann.

Auf der anderen Seite soll Dokumentation auch Diskussionen in Gang bringen und fördern. Zu Beginn eines Projektes ist vielen Beteiligten eventuell die Bedeutung vieler Begriffe und Zusammenhänge unklar. Oft ist zu diesem Zeitpunkt auch die Dokumentation an vielen Stellen unklar und unvollständig. Dass dies bei den Beteiligten dann Fragen aufwirft, ist durchaus positiv, solange sie auch gestellt werden und dadurch produktive Diskussionen auslösen. In vielen Fällen hilft es dann, wenn jemand die entstehenden Diskussionen moderiert und die Ergebnisse schriftlich festgehalten (dokumentiert) werden.

Weitere Aspekte, die dokumentiert werden sollten, erscheinen auf den ersten Blick trivial.

Beispiele dafür sind unter anderem Zuständigkeiten, zu verwendende Werkzeuge und deren Versionen, Termine und Zielvereinbarungen. Oft sind diese Fragestellungen in Projekten nicht eindeutig geklärt, was zwangsläufig zu Problemen führt. Man erleichtert sich die Arbeit, indem man all dies zu Papier oder in elektronische Form bringt und dafür sorgt, dass alle Beteiligten Zugriff darauf erhalten.

2.3 Modellierung und Beschreibung von Architekturen

Die in der Softwareentwicklung gebräuchlichste Modellierungssprache ist derzeit wohl UML (OMG, 2005) und dort speziell der Diagrammtyp des Klassendiagramms. UML ist, auch in der aktuellen Version, primär für die Modellierung objektorientierter Zusammenhänge gedacht und entworfen. UML ist keine Architekturbeschreibungssprache. Die darin eingesetzten Entitäten sind in erster Linie Klassen, Objekte und unterschiedliche Arten von Relationen. Diese sind jedoch keine architektonischen Elemente, sondern unterstützen eher die reine Datenmodellierung. Bei den in Abschnitt 2.1 genannten Definitionen war jedoch vielmehr von Komponenten die Rede, die miteinander in Beziehung stehen.

Um architektonische Elemente abzubilden und deren Zusammenhänge modellieren zu können, benötigt man andere Sprachen und Werkzeuge. Aus diesem Grund gibt es Architekturbeschreibungssprachen, oder kurz: ADLs. Sie verfügen über die notwendigen sprachlichen Konstrukte. Für eine knappe Einführung in das Gebiet der ADLs im Allgemeinen und der Sprache ACME im Besonderen, siehe (Garlan u. a., 1997) und (Garlan u. a., 2000). Doch ADLs sind nicht nur Hilfsmittel zur Modellierung. Sie sind außerdem Mittel zur Analyse von Architekturen. Sie können helfen, diese zu Bewerten und ihre Schwachstellen aufzuzeigen.

Da die Entscheidung für den Entwurf oder die Verwendung einer bestimmten Architektur stets ganz zu Beginn eines Entwicklungsprozesses stehen sollte, ist es sinnvoll, die gewählte Architektur in einer ADL zu modellieren, um sie dann anschließend analysieren zu können (Svahnberg u. a., 2002). Auf diese Weise können grundsätzliche Designfehler und -schwachstellen relativ früh aufgedeckt und beseitigt werden. Verfahren zur Analyse und Evaluierung von Softwarearchitekturen sind nicht Gegenstand dieses Thesis Outlines. Tiefer gehende Informationen zu diesem Themenbereich liefern (Bass u. a., 2001) oder (Clements u. a., 2002).

3 Views and Beyond – Module Viewtype

Views and Beyond ist ein umfassender Ansatz zur Dokumentation von softwarelastigen Architekturen (Clements u. a., 2003). Er wird im nachfolgenden Text mit V&B abgekürzt. Seine Herangehensweise ist, drei Sichtweisen, genannt *Viewtypes*, auf eine Architektur zu definieren, die sich jeweils auf bestimmte Aspekte einer Architektur konzentrieren. In der Literatur werden jedoch auch Anleitungen dazu gegeben, sich eigene Viewtypes zu definieren, sollte dies notwendig werden.

In diesem Abschnitt wird der *Module Viewtype* vorgestellt. Bei dieser Sichtweise handelt es sich um die Darstellung einer Architektur in ihren statischen Aspekten. Verhalten und Interaktion werden an dieser Stelle also nicht betrachtet. Stattdessen konzentriert man sich auf die Zerlegung der Architektur in implementierbare Einheiten. Die Beziehungen unter ihnen

charakterisieren sich hauptsächlich über Formen wie: *x ist-Teil-von y*, *x verwendet y* oder *x ist-abhängig-von y*.
Der Module Viewtype kann gut dazu verwendet werden, den konzeptionellen Überblick wiederzugeben. Bei der rein grafischen Darstellung dieses Überblicks werden in der Regel alle wesentlichen Einzelkomponenten, die Grundbausteine der Architektur, aufgeführt. Dabei werden Details häufig erst einmal weggelassen, ihre Beziehungen untereinander jedoch mit dargestellt. Der konzeptionelle Überblick ist oft der erste Eindruck, den ein Neuling im Projekt oder der Kunde von dem zu schaffenden Produkt erhält. Das bewusste Weglassen von Detailinformationen schreckt nicht gleich im ersten Moment ab und wirft zusätzlich Fragen beim Betrachter auf. Dies ist in Projekten durchaus positiv, um die Kommunikation unter den Beteiligten in Gang zu bringen. Die weiteren Konzepte sollen in den nachfolgenden Unterabschnitten anhand mehrerer Beispiele erläutert werden.

3.1 UML-Notation

Es wurde bereits erwähnt, dass UML nicht immer geeignet ist, um architektonische Zusammenhänge zu modellieren. Wenn es jedoch darum geht einen groben Überblick zu schaffen, reichen die Mittel von UML völlig aus.

Wie eingangs erwähnt, sollen im Module Viewtype die einzelnen Module und ihre Beziehungen untereinander dargestellt werden. Ein Modul kapselt eine bestimmte Funktionalität und verbirgt seine innere Struktur. Ein UML-Datentyp, der sich dafür anbietet, ist das Package-Konstrukt. Packages können in UML miteinander in Relation stehen und weitere Packages enthalten. Sie können Schnittstellen anbieten oder nutzen. Abbildung 1 auf Seite 9 zeigt ein simples Beispiel, welches alle diese Aspekte anwendet.

Dargestellt wird darin exemplarisch die Art und Weise der Kommunikation von Services innerhalb des Ferienclub-Szenarios. Es existieren zwei Services innerhalb des Ferienclubs. Sie bieten jeweils eine Schnittstelle an und Nutzen je eine Schnittstelle des Enterprise Service Bus. Die Services können sich nicht gegenseitig aufrufen, sondern dies nur über den Service Bus erledigen.

Diese Funktionsbeschreibung ist unabhängig von konkreten Technologien und stellt gleichzeitig eine Designentscheidung dar. Es handelt sich um eine Architektur nach den Definitionen aus (Bass u. a., 2003) und (IEEE1471, 2000), welche die Kommunikation der Dienste über bestimmte Wege lenken will und die direkte Kommunikation unter ihnen verbietet.

3.2 Informale Notation

Da UML für einige Leser schon zu formal sein könnte und für sie die Lesbarkeit erschweren würde, wird oft zusätzlich eine informale Notation verwendet. Bestimmte Architekturen, wie etwa das Schichtenmodell oder Pipes-and-Filter-Systeme können in UML gar nicht ohne Weiteres wiedergegeben werden. Entsprechende Sprachelemente fehlen dort, so dass informale Notationen oft sogar semantisch passender sind als UML.

Insbesondere um einen ersten Eindruck vom Projekt zu schaffen, bietet sich diese Form der Darstellung durchaus an. Frei von einer fest definierten Syntax lassen sich auf einfache Weise optisch ansprechende Zeichnungen erstellen. Es lassen sich dort alle wichtigen Schlagworte

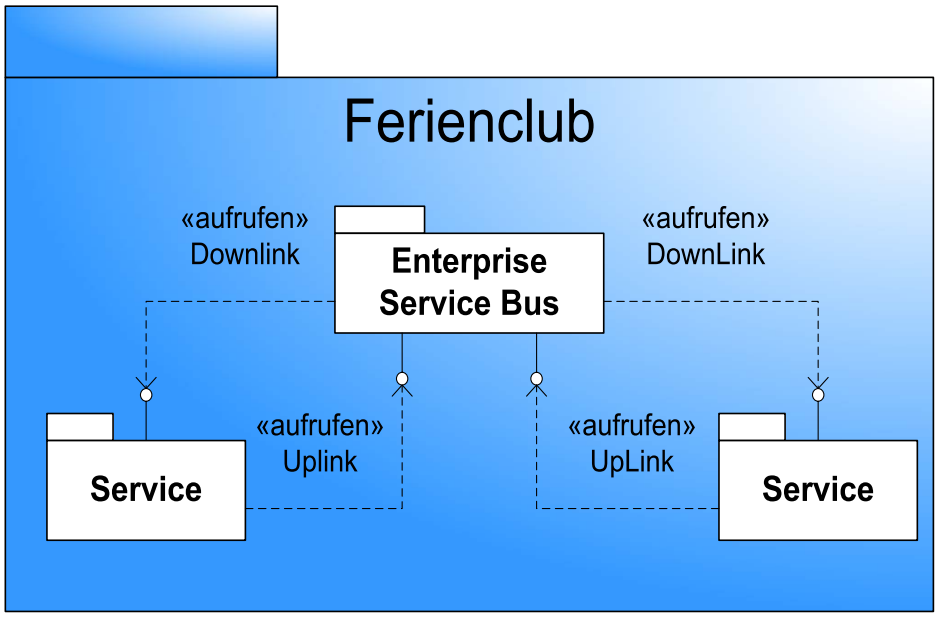

Abbildung 1: Module Viewtype nach UML-Notation

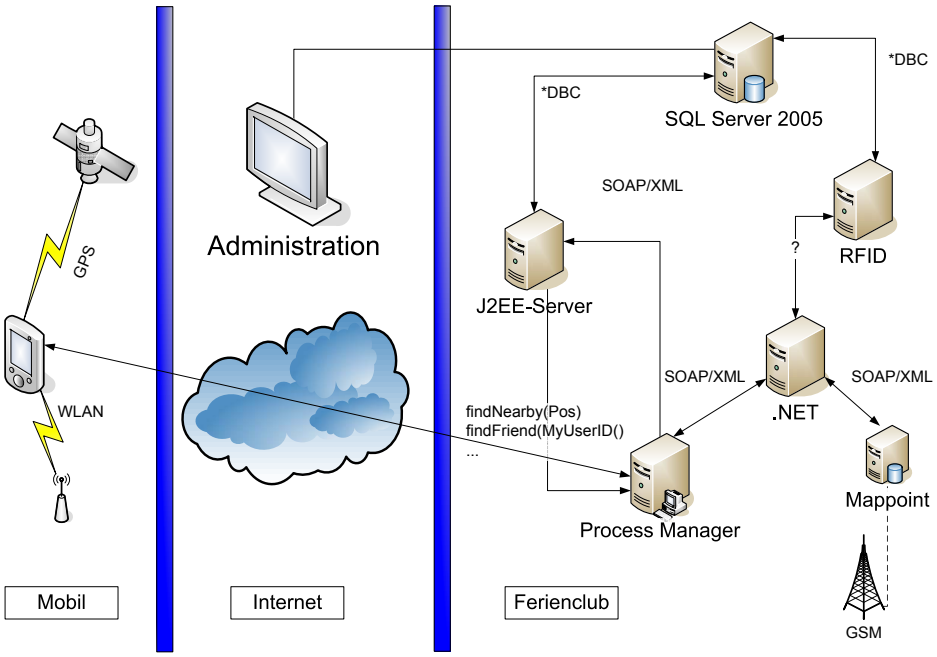

Abbildung 2: Module Viewtype in informaler Notation

oder auch „Buzzwords“ unterbringen. Mit Hilfe unterschiedlicher Symbole, zum Beispiel kleiner PC-Gehäuse, Monitore oder PDAs können sich damit auch Nicht-Techniker ein ungefähres Bild davon machen, welche Technik später zum Einsatz kommt. Auch wenn diese Darstellungen oft nicht die ganze Wirklichkeit wiedergeben, unterstützen sie den Einstieg in das Thema und können eine gemeinsame Wissensgrundlage für alle Beteiligten bieten.

Mit der Zeit sollte das Wissen über das Projekt bei den Beteiligten wachsen und sich konkretisieren. Idealerweise wird in diesem Zuge auch die Projektdokumentation weiter gepflegt und dem tatsächlichen Projektstand angepasst.

Abbildung 2 auf Seite 9 zeigt eine solche Darstellung der Architektur des Ferienclubs. Darin befinden sich drei Zonen: Mobil, Internet und der Ferienclub an sich. In der Ferienclub-Zone sehen wir diverse Komponenten, die über bestimmte Protokolle miteinander kommunizieren können. Darunter ein J2EE-Server, ein Microsoft SQL-Server sowie Dienste, die auf Microsoft .NET basieren. Generell ist zweiseitige Kommunikation möglich, angedeutet durch die Pfeile, die in beide Richtungen zeigen. Die Administration erfolgt internetbasiert. Zugriff auf die Ferienclub-Infrastruktur ist auch von einem mobilen Gerät aus möglich. Für Letzteres besteht außerdem die Möglichkeit, eine Positionsbestimmung mittels GPS oder eines WLAN-Dienstes vorzunehmen. In Kombination mit einem Mappoint-Server im Ferienclub können damit ortsbezogene Dienste genutzt werden.

Diese Beschreibung in Verbindung mit Abbildung 2 soll den konzeptionellen Überblick des Ferienclubs darstellen. Alle relevanten Technologien und „Buzzwords“ werden zumindest einmal genannt. Dass damit noch nicht alle Fragen geklärt sind, sondern sich im Gegenteil sogar erst ergeben, ist ein gewolltes Ziel. Der Betrachter soll angeregt werden, viele Fragen zu stellen, um die Kommunikation im Projekt zu fördern.

3.3 ACME-Notation

ACME zählt zu den Architekturbeschreibungssprachen. Sprachdefinition und weitere Erläuterungen finden sich in (Garlan u. a., 1997) sowie (Garlan u. a., 2000). Listing 1 zeigt eine einfache Beschreibung eines Client-Server-Systems.

Es gibt eine Server- und eine Client-Komponente. Beide verfügen über jeweils zwei Ports. Port ist im architektonischen Kontext die Bezeichnung für eine Schnittstelle. Dieser andere Begriff wird verwendet, da Schnittstellen auf architektonischer Ebene eine andere Bedeutung haben, als auf der rein Programmiertechnischen (Clements u. a., 2003).

Listing 1: ACME-Quellcode

```
System Ferienclub = {
    Component server   = { Port outBound; Port inBound; };
    Component client   = { Port outBound; Port inBound; };
    Connector upLink   = { Role caller; Role callee; };
    Connector downLink = { Role caller; Role callee; };

    Attachment server.outBound to downLink.caller;
    Attachment client.inBound  to downLink.callee;
    Attachment server.inBound  to upLink.callee;
    Attachment client.outBound to upLink.caller;
```

};

Ports sind in einer ADL First-Level-Elemente, ebenso wie die Komponenten. Von der Bedeutung her haben sie also das selbe Gewicht. Dies ist ein Unterschied zu UML. Dort gibt es zwar im Diagrammtyp *Composite Structure* ein Port-Element, dieses ist jedoch ein Attribut einer Klasse oder eines Objekts (OMG, 2005). Der First-Level-Element-Aspekt wird in einer grafischen Darstellung also oft nicht deutlich. In Fragen der Kommunikation und der Interaktion sind Ports jedoch von großer Komplexität und Bedeutung, so dass diese Klassifikation notwendig ist (Clements u. a., 2003, 2002).

Mit Hilfe weiterer Sprachelemente ist es jetzt beispielsweise möglich, Komponenten und Konnektoren mit Kapazitäts- und möglichen Belastungswerten zu belegen. Im Rahmen einer Laufzeitsimulation können auf diese Weise unter Anderem Schwachstellen oder Engpässe ermittelt werden. Dies ist insbesondere im Component-and-Connector-Viewtype von Bedeutung, der im folgenden Abschnitt erläutert wird.

4 Views and Beyond – Component-and-Connector Viewtype

Während im Module Viewtype die statischen Aspekte einer Architektur dargestellt werden, widmet sich der Component-and-Connector Viewtype der Dynamik. Hier liegt der Schwerpunkt der Betrachtung im Laufzeitverhalten einer Architektur. Die wichtigsten Elemente dabei sind Komponenten und Konnektoren, die auch Namens gebend für diesen Viewtype waren.

Als Komponenten werden alle Elemente einer Architektur verstanden, die entweder für die Applikationslogik oder die Datenspeicherung zuständig sind. Relevant sind alle Komponenten, die zur Laufzeit existieren. Konnektoren stehen für Interaktion und sind eine spezifische Form der Kommunikation, wie beispielsweise eine Transaktion, ein Remote-Procedure-Call oder der Aufruf eines Webservice (Clements u. a., 2003).

4.1 Laufzeitverhalten

Wichtige Eigenschaften jeglicher Software sind unter anderem Performance, Ausfallsicherheit und Verfügbarkeit. Ob sie den geforderten Qualitäten entsprechen, ist oft nur zur Laufzeit messbar. Einflussfaktoren sind in der Praxis unter anderem die Anzahl der Zugriffe auf bestimmte Ressourcen, der Bedarf an Speicher und Bandbreite oder die Anzahl der gleichzeitigen Nutzer.

Um dem schon zur Designzeit Rechnung zu tragen, benötigen Architekten eine Laufzeitsicht auf die Architektur. Um eine Analogie zur objektorientierten Modellierung herzustellen: man modelliert nicht mehr mit Klassendiagrammen, wie es im Module Viewtype der Fall war (siehe Abschnitt 3). Stattdessen verwendet man Modelle analog zu Objektdiagrammen. Die Komponenten werden als konkrete Instanzen dargestellt. Von einem Komponententyp *Server* können in einem Component-and-Connector View also mehrere Instanzen auftreten. Das Selbe gilt für Clients oder die Verbindungen unter den Komponenten. Existiert also zwischen Client und Server vom Konzept her nur eine logische Verbindung, so kann dies im laufenden Betrieb durchaus anders sein. Um Ausfallsicherheit und/oder bessere Performance zu gewährleisten, können zur Laufzeit mehrere redundante Verbindungen im Gebrauch sein. Dieser Sachverhalt soll im Component-and-Connector Viewtype deutlich werden.

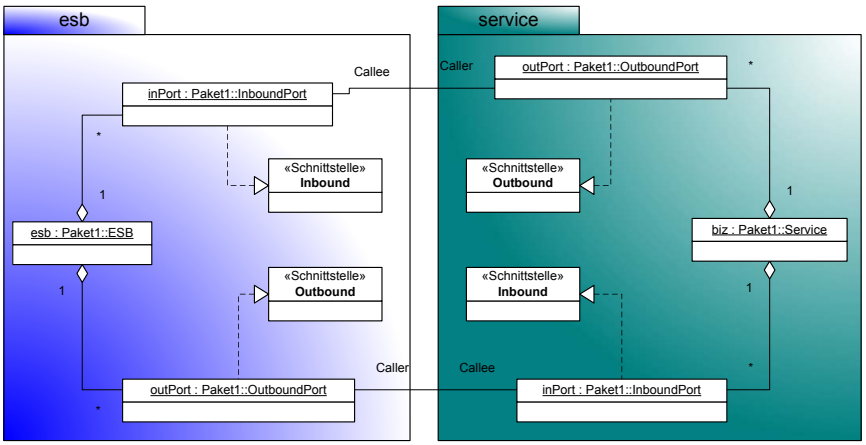

Abbildung 3: Component-and-Connector Viewtype nach UML-Notation

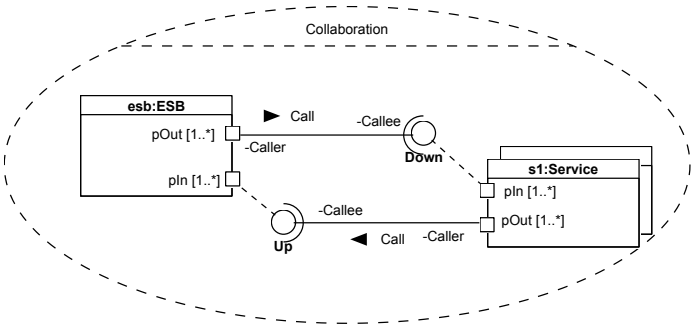

Abbildung 4: Component-and-Connector Viewtype nach UML2.0-Notation

4.2 Schnittstellen

Schnittstellen sind ein Treffpunkt zweier unabhängiger Entitäten, an dem diese Entitäten miteinander kommunizieren (Clements u. a., 2003). Wichtig ist, dass Schnittstellen auf architektonischer Ebene nicht die selbe Bedeutung haben, wie auf der programmiertechnischen Ebene. Vom UML-Verständnis her kann eine Schnittstelle nicht instanziiert werden. Es gibt dort Klassen und Objekte, die eine Schnittstelle realisieren können. Normalerweise realisiert eine Klasse ein Interface auch nur einmal und nicht mehrfach. Verwendet man UML zur Architekturbeschreibung, dann sind einige Sachverhalte, zumindest mit der UML Version 1.4, nur sehr umständlich darstellbar.

Beispielsweise muss eine Komponente, wie etwa ein Datenbankserver, eine Client-Schnittstelle mehrfach anbieten, damit nicht nur ein Benutzer gleichzeitig auf die Datenbank zugreifen kann. Will man jetzt diesen Sachverhalt mit UML modellieren, so entstehen dabei recht umfangreiche Modelle, wie etwa Abbildung 3.

Zwei miteinander kommunizierende Komponenten, ein Service und ein Enterprise-Service-Bus, verfügen über mehrere Ports, über die sie mit der Gegenstelle kommunizieren. Die Kom-

ponenten können jeweils mehr als einen Port haben, entsprechend sind die Kardinalitäten an den Aggregations-Relationen gesetzt. Dazu ist es notwendig geworden, die Ports als UML-Objekte zu modellieren, ebenso wie die Komponenten selbst. Das Problem ist nun, dass für Komponenten, deren Semantik eigentlich unterschiedlich ist, die selben Symbole verwendet werden.

Mit der UML-Version 2.0 wurde jedoch auch das Konzept des Ports in den Standard aufgenommen. Abbildung 4 auf Seite 12 zeigt ein entsprechendes Diagramm. Der semantische Unterschied zwischen Port und Objekt ist deutlich sichtbar. Nachteilig ist immer noch, dass die Ports nicht mehr als First-Level-Elemente erscheinen, da sie erheblich kleiner als ein Objekt sind. Je nachdem, wie relevant die Ports im konkreten Fall sind, bietet sich also entweder die Form aus Abbildung 3 oder diejenige aus Abbildung 4 an.

5 Views and Beyond – Allocation Viewtype

Dieser Viewtype spielt im Rahmen des Ferienclub-Projekts keine entscheidende Rolle und wird deswegen an dieser Stelle nur kurz vorgestellt. Für ausführlichere Erläuterungen siehe (Clements u. a., 2003).

Der Allocation Viewtype dient dazu, die Verbindungen einer Softwarearchitektur zu den Nichtsoftware-Komponenten zu dokumentieren. Außerdem sollen mit ihm die Zuständigkeiten der Beteiligten Personen für die Teilkomponenten festgehalten werden. Beispielsweise kann aufgrund des konzeptionellen Überblicks oder anderer Darstellungen des Module Viewtypes (vgl. Abschnitt 3) ein Mapping von Mitarbeitern auf die einzelnen Module vorgenommen werden. Weiterhin wird dokumentiert, welche Software-Komponenten auf welchen Hardwarekomponenten oder sonstigen Laufzeitumgebungen installiert werden.

6 Zusammenfassung und Ausblick auf die Master's Thesis

Die Kernidee des Views-and-Beyond-Ansatz zur Dokumentation von Softwarearchitekturen ist, verschiedene Sichten zu definieren, welche alle relevanten Aspekte der Architektur beleuchten. V&B definiert dazu drei Sichten.

Zum Einen soll die statische Sicht der Architektur dokumentiert werden. Diese umfasst die Zerlegung Letzterer in einzelne Module und deren Beziehungen untereinander. Das Ziel dabei ist, den Projektbeteiligten einen konzeptionellen Überblick des Projekts zu geben. Außerdem bietet die Festlegung der einzelnen Module eine Grundlage, die Arbeitsteilung im Projekt zu organisieren. Sind die Module erst entworfen, können daran die Zuständigkeiten der Entwickler festgemacht werden.

Die zweite definierte Sichtweise zielt auf die Betrachtung und Dokumentation des Laufzeitverhaltens der Architektur ab. Von besonderer Bedeutung ist hier der Performance-Aspekt, der Verbrauch von Ressourcen sowie die Interaktion der Komponenten untereinander.

Die dritte und letzte Sicht beschäftigt sich mit der Einbettung der Softwarearchitektur in ihre Nicht-Software-Umgebung, also der Installation auf Servern und Dateisystemen. Weiterhin wird hier die Zuordnung der Zuständigkeiten von Personen auf die Komponenten der Architektur festgehalten.

Beim Verfassen der Dokumentation ist weiterhin die Wahl der Notation von Bedeutung. In Abschnitt 3 wurden sowohl die UML-Notation, die Notation in einer ADL als auch die informale Notation erläutert. Alle drei haben ihre Vor- und Nachteile. Während die Sachverhalte in UML und insbesondere in ACME sehr exakt modelliert werden können, so sind sie doch für einen Nicht-Techniker oft nur schwer lesbar. Die informale Notation hat den eindeutigen Nachteil, dass sie oft Ungenauigkeiten enthält und in einigen Fällen sogar nicht hundertprozentig korrekt ist. Bei sorgfältiger Anwendung hat sie jedoch durchaus ihre Berechtigung. Frei von Formalien und Details kann sie so gestaltet werden, dass komplizierte Sachverhalte auch für Laien verständlich gemacht werden können.

In einer Master's Thesis im kommenden Semester sollen die hier vorgestellten Methoden dazu benutzt werden, ein Framework zum Vergleich von Architekturbeschreibungssprachen zu entwerfen.

Anhand einer zeitgemäßen, verteilten Softwarearchitektur, wie etwa Java J2EE, sollen zwei ADLs miteinander verglichen werden. Herangezogen werden voraussichtlich ACME und xADL (siehe (Garlan u. a., 2000) bzw. (Dashofy u. a., 2001)). Er wird untersucht, wie die einzelnen Aspekte der Architektur mit der jeweiligen ADL erfasst werden können. Um möglichst alle relevanten Architekturaspekte zu berücksichtigen, soll dabei V&B als Leitfaden dienen und die unterschiedlichen Sichten auf die Architektur vorgeben.

Die daraus gewonnenen Erkenntnisse sollen anschließend genutzt werden, um eine allgemeine Vorgehensweise zum Vergleich von ADLs zu entwickeln.

Literatur

[Bass u. a. 2003] BASS, Len ; CLEMENTS, Paul ; KAZMAN, Rick: *Software Architecture in Practice*. 2. Addison Wesley, 2003 (Series in Software Engineering). – 1–529 S

[Bass u. a. 2001] BASS, Len ; KLEIN, Mark ; MORENO, Gabriel: Applicability of General Scenarios to the Architecture Tradeoff Analysis Method / School of Computer Science – Carnegie Mellon University. 2001 (CMU/SEI-2001-TR-014). – Forschungsbericht

[Borsi 1975] BORSI, Francesco: *Leon Battista Alberti. Das Gesamtwerk*. Milano, 1975

[Clements 2005] CLEMENTS, Paul: Comparing the SEI's Views and Beyond Approach for Documenting Software Architectures with ANSI-IEEE 1471-2000 / School of Computer Science – Carnegie Mellon University. Juli 2005 (CMU/SEI-2005-TN-017). – Forschungsbericht

[Clements u. a. 2003] CLEMENTS, Paul ; BACHMANN, Felix ; BASS, Len u. a.: *Documenting Software Architectures*. 1. Addison Wesley, 2003 (Series in Software Engineering). – 1–529 S

[Clements u. a. 2002] CLEMENTS, Paul ; KAZMAN, Rick ; KLEIN, Mark: *Evaluating Software Architectures*. 1. Addison Wesley, 2002 (Series in Software Engineering)

[Dashofy u. a. 2001] DASHOFY, Eric M. ; HOEK, André van der ; TAYLOR, Richard N.: A Highly-Extensible, XML-Based Architecture Description Language. In: *Proceedings of the*

Working IEEE/IFIP Conference on Software Architectures (WICSA), Amsterdam, Netherlands IEEE (Veranst.), 2001, S. 10

[Garlan u. a. 1997] GARLAN, David ; MONROE, Robert T. ; WILE, David: Acme: an architecture description interchange language. In: *CASCON '97: Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research* ACM (Veranst.), IBM Press, 1997, S. 7

[Garlan u. a. 2000] GARLAN, David ; MONROE, Robert T. ; WILE, David: Acme: Architectural Description of Component-Based Systems. In: LEAVENS, Gary T. (Hrsg.) ; SITARAMAN, Murali (Hrsg.): *Foundations of Component-Based Systems*, Cambridge University Press, 2000, S. 47

[IEEE1471 2000] IEEE (Veranst.): *IEEE Std 1471 Recommended Practice for Architectural Description*. 2000

[OMG 2005] Object Management Group (Veranst.): *Unified Modelling Language: Superstructure version 2.0*. August 2005

[Pollio 1996] POLLIO, Marcus V.: *Zehn Bücher über Architektur*. 5. Primus Verlag, 1996

[SEI Website 2005] SEI: Website des Carnegie Mellon Software Engineering Institute (Veranst.): *Software Architecture for Software-Intensive Systems*. Oktober 2005

[Shaw 2001] SHAW, Mary: The coming-of-age of software architecture research. In: *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*. Washington, DC, USA : IEEE Computer Society, 2001, S. 656. – ISBN 0-7695-1050-7

[Svahnberg u. a. 2002] SVAHNBERG, Mikael ; WOHLIN, Claes ; LUNDBERG, Lars ; MATTSSON, Michael: A method for understanding quality attributes in software architecture structures. In: *SEKE '02: Proceedings of the 14th international conference on Software engineering and knowledge engineering* ACM (Veranst.), ACM Press, 2002, S. 819–826. – ISBN 1-58113-556-4

Glossar

ACME: Eine generische → ADL. Maßgeblich wird ihre Entwicklung vom Software Engineering Institute der Carnegie Mellon University vorangetrieben. Siehe (SEI Website, 2005). In der amerikanischen Umgangssprache steht die Abkürzung ACME normalerweise für *A company manufacturing everything*. Ob ACME im Kontext der ADLs ebenfalls eine Abkürzung ist, konnte der Autor nicht in Erfahrung bringen. Es wird vermutet, dass die genannte Bedeutung eine Anspielung auf den generischen Charakter von ACME als ADL sein soll.

ADL: Architecture Definition Languages – Architekturbeschreibungssprache. Siehe (Clements u. a., 2003)

HAW: Hochschule für Angewandte Wissenschaften Hamburg

IEEE: Institute of Electrical and Electronics Engineers

PDA: Persönlicher Digitaler Assistent. Ein mobiler Computer, oft im Westentaschenformat.

Stakeholder: Eine Person, die ein Interesse an dem Projekt hat.

UML: Unified Modelling Language. Siehe (OMG, 2005)

V&B: Views and Beyond. Siehe (Clements u. a., 2003)