CARL VON OSSIETZKY universität OLDENBURG

Fakultät II - Department für Informatik
Abteilung Informationssysteme

OFFIS

OFFIS - Institut für Informatik
Bereich Betriebliches Informationsmanagement

Ausarbeitung zum Seminarthema: Modulare Software-Entwicklung

Projektgruppe Modulares EAM System

Jörn Trefke

Oldenburg, 26. Oktober 2007

Inhaltsverzeichnis

Abstrakt: Die Entwicklung modularer Software insbesondere basierend auf Komponenten ist ein viel verwandtes Konzept, welches heutzutage besonders im Bereich der E-Commerce Anwendungen eingesetzt wird. Diese Arbeit motiviert zunächst den Einsatz Modularer Software und klärt Grundbegriffe der modularen/komponentenorientierten Entwicklung.

1 Motivation - Modulare Software

In der heutigen Gesellschaft spielen Softwaresysteme eine immer wichtigere Rolle. Praktisch in jedem Lebensbereich haben diese Systeme Einzug gehalten: Smart cards für persönliche Identifikation, Online-Banking, E-Learning, E-Commerce, softwarekontrollierte medizinische Geräte, Airbags in den Autos und Autopiloten für Flugzeuge sind nur einige Beispiele, die zeigen, wie das tägliche Leben von dem korrekten Verhalten von Software abhängen.
Die damit einhergehende Komplexität moderner Software kann durch geschickte Untergliederung des Gesamtsystems reduziert werden und die einzelnen Subsysteme werden dadurch besser handhabbar.
Durch die modulare Entwicklung entstehen verschiedene Vorteile für Softwareentwickler und Softwarenutzer:
Zum Einen kann eine verkürzte Entwicklungszeit erreicht werden, da verschiedene Gruppen ohne großen Kommunikationsaufwand an einem Modul der Software arbeiten können. Diese Module sind genau spezifiziert und bereits früher verwendete Module, die auf der selben Spezifikation beruhen können möglicherweise wieder eingesetzt werden.
Weiterhin erhöht sich durch die mit der modulare Entwicklung verbundene, genaue Spezifikation der Module, die Flexibilität der Software, da es ohne weiteres möglich ist ein Modul zu verändern ohne andere Module anpassen zu müssen. Wenn Kernsysteme mit Grundfunktionalitäten angeboten werden und eine Möglichkeit besteht diese durch Module zu erweitern, so können diese Kernsysteme den Wünschen der Nutzer besser angepasst werden und in wirtschaftlicher Hinsicht, muss auch nur für die Funktionalität bezahlt werden, die auch tatsächlich genutzt wird.
Zudem erhöht sich die Verständlichkeit der Software, da es möglich sein sollte, sich mit einem Modul zur Zeit auseinanderzusetzen ohne konkret das Gesamtsystem dahinter verstehen zu müssen. Das gesamte System kann dadurch sauberer entworfen werden, da die einzelnen Teile und somit das Gesamtsystem besser verstanden werden können.
Modularität ist für Wiederverwendbarkeit und das Konzept der komponentenbasierten Entwicklung von großer Wichtigkeit. Durch die Wiederverwendung einer bewährten Komponente kann der Testaufwand und die Integration minimiert werden und die Qualität kann erhalten werden. Unter Umständen wird auch die Verlässlichkeit des Systems erhöht, da eine Komponente bei festgestellter Fehlerhaftigkeit oder Ausfall durch eine andere Komponente ersetzt werden kann um den weiteren Betrieb

sicherzustellen.

Die Modularität einer Software wird auf mehreren Ebenen festgelegt – angefangen beim Entwurf oder der Auswahl einer geeigneten Architektur, der Identifikation von Modulen und Schnittstellen bis hin zur Auswahl oder Implementierung der „richtigen“ Komponente. Dabei sind vielschichtige Entscheidungen zu treffen die Einfluss auf die Modularität haben.

Im Folgenden wird zunächst auf den Modulbegriff eingegangen und die Identifikation von Modulen im Wesentlichen beschrieben. Danach wird das Konzept der komponentenbasierten Softwareentwicklung vorgestellt welches ein Ansatz für die Entwicklung modularer Software ist. Anschließend wird kurz auf die Unterstützung bei der Implementierung modularer Software eingegangen.

2 Module und Komponenten

Der folgende Abschnitt zeigt zunächst die Definition von Modulen auf und identifiziert Kriterien für die Aufteilung der Module.

Die Festlegung der Modularisierung einer Software kann im Bereich der Software-Architektur eingeordnet werden und ist Teil des Entwurfsprozesses.

> „The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.“

So betonen Bass et al. [BCK03] in der vorhergehenden Definition zur Software-Architektur, dass Module („elements“) und die damit verbundenen Schnittstellen („visible properties“) und deren Beziehung („relationships“) zueinander bereits Teil der Software-Architektur sind. Die Dekomposition eines Softwaresystems in Module dient in dem Fall dazu, dass die Komplexität besser beherrschbar wird.

Was genau ein Modul ist soll im Folgenden geklärt werden. Kritieren für die Findung von Modulen sind in 2.2 zu finden.

2.1 Modulbegriff

Die modulare Programmierung ist seit fast 30 Jahren Forschungsgegenstand des Software Engineering.

In der Literatur ist kein einheitlicher Modulbegriff zu finden, so sei an dieser Stelle Parnas zitiert, der als einer der Erfinder des „information hiding“ gilt, welches eines wesentliche Voraussetzung für modulare Software ist. Er macht in [PCW84] folgende Aussagen zu einem Modul (übersetzt aus dem Englischen):

> „Ein Modul ist ein Arbeitsauftrag für einen Programmierer oder ein Programmierteam. Jedes Modul besteht aus einer Gruppe von eng miteinander verwandten Programmen. Diese Modulstruktur ist die Dekomposition des Programms in Module und die Annahme, dass das Team für jedes Modul verantwortlich ist, kann auch über andere Module gemacht werden.“

Module sind also statische Zuordnungen, die spezielle Funktionalitäten oder andersartige zusammengehörige Elemente zusammenfassen, welche zur Kompilierzeit feststehen.

Müller [Mül02] fasst einige der in Parnas' Darstellung bereits enthaltenen Ansätze auf und expliziert weitere Merkmale als Kriterien für Modularisierung:

- Klare Strukturierung
- getrennte Entwicklung
- „information hiding“
- Wiederverwandbarkeit

Module dienen demnach also der Strukturierung des Softwaresystems und können getrennt entwickelt werden. Sie verfolgen das Ziel des „information hiding“, d.h. dass die Information in dem Modul gekapselt ist und von anderen Modulen nicht direkt zugegriffen werden kann. Nicht zuletzt daraus folgt, dass Module als Einheiten für Wiederverwendbarkeit gesehen können.

2.2 Identifikation von Modulen - Kriterien für die Aufteilung

Das Primärziel der Dekomposition eines Systems in Module ist nach Parnas [PCW84] die Reduktion der Kosten durch die Wiederverwendbarkeit der Module, die unabhängig entworfen, entwickelt und getestet werden können - somit können auch Änderungen am System effizienter durchgeführt werden. Da das Innere eines Moduls normalerweise komplizierter ist als die Schnittstelle, wird durch Modulbildung auch die Systemkomplexität reduziert.

Parnas [Par72] betont als weiteres wichtige Dekompositionskriterium die Definition der Schnittstellen.
Sie sollten so entworfen werden, dass eine wahrscheinliche Änderung im Entwurf des Gesamtsystems nicht notwendigerweise auch Änderung an diesen erfordert. Weniger wahrscheinliche Änderungen können Schnittstellenänderungen erfordern, aber nur für kleine und nicht häufig verwendete Module. Nur sehr unwahrscheinlche Änderungen sollten Änderungen der Schnittstellen von häufig genutzten Modulen erfordern. Idealerweise sollten also sich häufig ändernde Anforderungen in Modulen gekapselt versteckt werden, damit das Design unabhängig von den Änderungen werden kann.

Die in 2.1 von erwähnten Merkmale von Modulen lassen sich als Kritieren für die Modularisierung nehmen (vgl. Müller [Mül02]). Im Folgenden werden diese Punkte im Bezug auf die Dekomposition näher beschrieben.

2.2.1 Klare Strukturierung

Große Softwaresysteme werden anhand von Modulen zerlegt um sie handhabbar zu machen. Programme bestehen normalerweise aus einer Vielzahl von Programmelementen (wie Klassen, Methoden, Variablen etc.). Indem Elemente die eng miteinander zusammenhängen in einem Modul zusammengefasst werden, können die Strukturen dieser Programme verdeutlicht werden. Parnas [Par72] nennt als Zielsetzung eine Modulstruktur, die einfach genug ist um sie noch vollständig zu verstehen.

Die Abhängigkeiten zwischen den Modulen (Kopplung) sollten aber so gering wie möglich gehalten werden (lose gekoppelt, „loosely coupled“) um die Komplexität zu reduzieren und problemloseren Austausch von Modulen zu ermöglichen - implizite Abhängigkeiten hingegen könnten bei Änderungen schnell zu Fehlern führen.

Müller [Mül02] hebt hervor, dass Module in guten Entwürfen und Implementierungen, so zugeschnitten werden sollten, dass die Elemente jedes Moduls sehr eng miteinander verbunden sind (hohe Kohäsion). Durch den engen Zusammenhang der Elemente, die normalerweise Zugriff auf dieselben Daten haben, findet der Informationsaustausch i.d.R. über eine gemeinsame Abstraktionsebene statt, wie zum Beispiel einen abstrakten Datentypen um auch diese Daten konsequent zu kapseln.

In den meisten aktuellen Programmiersprachen sind Modulkonzepte vorhanden, mit denen Programme strukuriert werden können. In Sprachen wie C findet die Modularisierung durch Anwendung von Konventionen statt: Module sind mit Dateien verknüpft und Schnittstellen werden durch Header Dateien ausgedrückt. In objektorientierten Sprachen dienen Klassen als Einheiten für die Modularisierung.
Daneben bieten einige objektorientierte Sprachen auch weitere Möglichkeiten zur Gruppierung zusammengehöriger Klassen an (wie z.B. Packages in Java). Viele Sprachen erlauben nur eine flache Modulstruktur - Java beispielsweise unterstützt aber auch eine hierarchische Modularisierung durch innere Klassen.

2.2.2 getrennte Entwicklung

Neben der Komplexitätsreduktion nennt Müller [Mül02] die getrennte Entwicklung der Module als eine der bedeutendsten Motivationen für die Einführung von Modulen. Module interagieren miteinander nur über klar definierte Schnittstellen, daher können sie unabhängig implementiert, kompiliert und getestet werden.
Durch die getrennte Entwicklung ist eine kostengünstigere Softwareerstellung möglich: Mehrere Entwickler können parallel an der Software entwickeln und der Aufwand für Kompilierung und Tests reduziert sich u.U. nur auf die neuen oder modifizierten

Module. Getrennt entwickelte Module sind weiterhin Voraussetzung für die Wiederverwendbarkeit.

2.2.3 „information hiding“

Parnas [Par72] verlangt, dass es möglich ist, die Implementierung eines Moduls zu verändern, ohne über das Wissen der Implementierung anderer Module zu verfügen und ohne das Verhalten anderer Module zu beeinflussen.
Das Verbergen von Information („information hiding“) ist eine wesentliche Technik um die gegenseitigen Abhängigkeiten von getrennt entwickelten Modulen zu minimieren.
Wenn ein bestehendes Modul modifiziert oder ein neues Modul implementiert wird, so können andere Systemteile, die sich auf die Schnittstelle des Moduls verlassen dieses verwenden ohne die Implementierung zu kennen und der Rest des Systems kann dadurch unverändert bleiben.

In vielen objektorientierten Sprachen wird „information hiding“ mit Kapselung kombiniert. Mit der Technik der Kapselung kann verhindert werden, dass auf interne Daten von Datenstrukturen zugegriffen wird ohne deren externe Schnittstellen zu benutzen. Diese Zugriffsbeschränkung auf Daten erlaubt es den Modulen die Konsistenz der Daten sicherzustellen und verhindert, dass vertrauliche Informationen (z.B. Passwörter) zugreifbar sind.

Die meisten existierenden Programmiersprachen unterstützen „information hiding“. Implementierende Module beschreiben ihre Schnittstellen über die Angabe der Elemente die durch andere Module genutzt werden können in bezeichnenden Teilen der Moduldeklaration z.B. durch markieren der Elemente durch Zugriffsmodi (beispielsweise private, protected oder public in Java).
Der Compiler garantiert, dass extern nur die erlaubten Elemente der Module aufgerufen werden können. Die meisten objektorientierten Sprachen erlauben es mehrere Schnittstellen für ein Modul/Klasse zu definieren. Typischerweise gibt es verschiedene Schnittstellen für Benutzer, implementierende Unterklassen und so genannten befreundeten Klassen.

2.2.4 Wiederverwendbarkeit

Da die sich in einem Modul befindlichen Programmelemente eng zusammenarbeiten, normalerweise eine gemeinsamem Abstraktion implementieren und nach außen hin nur über ihre Schnittstelle zugreifbar sind, können sie als Einheiten für Wiederverwendbarkeit gelten.

Sommerville [Som04] beschreibt weiterhin zwei verschiedene Ansätze für die Dekomposition eines (Sub-)Systems.

Zum einen die Möglichkeit der objektorientierten Dekomposition mit der versucht wird zusammengehörige Dinge mit den dazugehörigen Operationen zusammenzufassen und zum Anderen die funktionsorientierte Befehlsverknüpfung (pipelining). Bei der zweiten Variante werden im Wesentlichen zusammengehörige Funktionen zusammengefasst, die dann in einer bestimmten Reihenfolge ausgeführt werden.

3 Komponentenbasierte Softwareentwicklung

Nachdem im vorigen Abschnitt Module angesprochen worden, wird im Folgenden das Komponentenbasierte Software-Engineering (component-based software engineering (CBSE)) näher beschrieben.

CBSE ist in den späten 90er Jahren als ein Ansatz für wiederverwendbare Softwaresystementwicklung aufgetaucht.
Im CBSE Prozess werden unabhängige Komponenten definiert, implementiert und zu lose gekoppelten (loosely coupled) Systemen integriert oder zusammengesetzt („compose“). Die Entwicklung wiederverwendbarer Software(komponenten) ist ein guter Weg um die Komplexität meistern zu können und auf die Anforderungen schnellerer Auslieferungen reagieren zu können.

Sommerville [Som04] fasst als vier wesentliche Merkmale komponentenbasierter Softwareentwicklung unabhängige Komponenten, Komponentenstandards, Middleware und einen angepassten Entwicklungsprozess zusammen:

Unabhängige Komponenten

Diese sind durch ihre Schnittstellen komplett spezifiziert. Es sollte dabei eine klare Trennung zwischen der Komponentenschnittstelle und deren Implementierung geben, so dass eine Komponente unproblematisch durch eine andere ersetzt werden kann.

Komponentenstandards

Die Integration der Komponenten wird durch Komponentenstandards erleichtert. Diese Standards werden in einem Komponentenmodell auf einem sehr niedrigen Niveau festgelegt und umgesetzt, um zu beschreiben wie Komponentenschnittstellen spezifiziert werden sollten und wie sie miteinander kommunizieren. Einige Modelle definieren Schnittstellen, die von allen zum Standard konformen Komponenten implementiert werden sollten. Wenn Komponenten einem Standard entsprechen, dann ist deren Betrieb unabhängig von der Programmiersprache. Komponenten, die in verschiedenen Programmiersprachen geschrieben wurden, können in das selbe System integriert werden.

Middleware

Middleware stellt die Softwareunterstützung für die Komponentenintegration zur Verfügung. Damit unabhängige, verteilte Komponenten zusammenarbeiten können, muss die Middleware die Komponentenkommunikation unterstützen. Middleware wie CORBA behandelt low-level Probleme effizient und erlaubt es sich auf Applikationsprobleme zu konzentrieren. Zusätzlich zum Komponentenmodell kann die Middleware Unterstützung für z.B. Transaktionsmanagement, Sicherheit oder Nebenläufigkeit bieten.

Entwicklungsprozess

Ein Entwicklungsprozess der auf komponentenbasiertes Software-Engineering abgestimmt ist, stellt ein wesentliches Merkmal dar. Setzt man Entwicklungsprozesse ein, die auf die originäre Softwareproduktion ausgelegt sind, stellt man fest, dass der Prozess das Potential von CBSE limitiert.

Komponenten sind unabhängig, so dass sie sich nicht gegenseitig im Betrieb beeinflussen. Die Implementierungsdetails sind verborgen, so dass die Implementierung verändert werden kann, ohne den Rest des Systems zu beeinflussen. Da die Komponenten über wohldefinierte Schnittstellen kommunizieren, kann bei Wartung dieser Schnittstellen, eine Komponente durch eine andere Komponente ausgetauscht werden, die zusätzliche oder erweitere Funktionalität zur Verfügung stellt. Zusätzlich bieten Komponenteninfrastukturen high-level Plattformen, welche die Kosten der Applikationsentwicklung reduzieren.

Aus der Wiederverwendbarkeit von Komponenten entstehen schließlich weitere Fragen, wie z.B. Vertrauenswürdigkeit oder Zertifizierung von Komponenten oder Fragen über das sich ergebende (Laufzeit)Verhalten, wenn mehrere Komponenten zusammenarbeiten.

3.1 Begriff der Komponente

Es gibt keine klare Definition für eine Komponente, Sommerville [Som04] nennt aber als generell anerkannt, dass eine Komponente eine unabhängige Softwareeinheit ist, die mit anderen Komponenten zusammengesetzt werden kann um ein Softwaresystem zu erstellen.

In Heinemann et al. [HC01] wird einen Komponente als „ein Softwareelement, dass konform zu einem Komponentenmodell ist und unabhängig genutzt und zusammengesetzt („composed“) werden kann ohne eine Modifikation nach dem „composition standard“ vorzunehmen“ beschrieben.

Szyperski [Szy02] hingegen betont in seiner Definition die „Einheit“ und die „Schnittstellen“ als wesentliche Charakteristiken einer Komponente (übersetzt aus dem Englischen):

> Eine Softwarekomponente ist eine zusammengesetzte Einheit mit vertraglich spezifizierten Schnittstellen und nur expliziten Kontextabhängigkeiten. Eine Softwarekomponente kann unabhängig eingesetzt („deployed“) werden und ist Subjekt von Kompositionen von Dritten.

Komponenten sind aber nicht gleichzusetzen mit Klassen oder Objekten. Szyperksi [PS98] schreibt dazu:

> A component is defined as a collection of cooperating objects, with a clearly defined boundary to other objects or components. Objects inside of a component typically are intertwined tightly, while interaction across the component boundary is relatively weak.

Komponenten unterscheiden sich insofern, dass sie einsetzbare Einheiten sind die standardisiert sind und einem Komponentenmodell genügen, welches ihre Implementierung einschränkt. Aufgrund der klar definierten Grenze der Komponente sind sie letztendlich nur über Schnittstellen definiert und die Implementierung ist vollkommen transparent für den Nutzer. Abschließend hängen auch nicht von einer konkreten Programmiersprache ab und könnten prinzipiell auch in nicht objektorientierten Sprachen implementiert werden.

3.1.1 Charakteristika einer Komponente

Nähere Charakteristiken einer Komponente wie sie im CBSE genutzt wird, hat Sommerville [Som04] zusammengestellt. Sie werden als notwendige Eigenschaften einer Komponente beschrieben:

Standardisiert

Komponentenstandardisierung bedeutet, dass einen Komponente die im CBSE Prozess genutzt wird, konform zu einem standardisierten Komponentenmodell sein muss. Dieses Modell kann Komponentenschnittstellen, Komponentenmetadaten, Dokumentation, Komposition und Deployment festlegen.

Unabhängig

Eine Komponente sollte unabhängig sein – es sollte möglich sein sie zusammenzusetzen und zu deployen ohne spezifische andere Komponenten zu nutzen.

Zusammensetzbar

Damit eine Komponente zusammensetzbar („composeable“) ist, müssen alle externen Aufrufe durch eine als öffentlich deklarierte Schnittstelle erfolgen. Zusätzlich muss sie externen Zugriff ermöglichen um Informationen über sich selbst, wie Methoden und Attribute preiszugeben.

Einsetzbarkeit

Um einsetzbar („deployable“) zu sein, muss eine Komponente in sich abgeschlossen sein und in der Lage sein, als alleinstehende Einheit auf einer Plattform zu laufen, die das Komponentenmodell implementiert.

Dokumentiert

Komponenten müssen vollkommen dokumentiert sein, so dass potentielle Nutzer entscheiden können, ob die Komponente ihren Bedürfnissen genügt oder nicht. Die Syntax und idealerweise die Semantik aller Komponentenschnittstellen sollte spezifiziert sein.

3.2 Komponentenschnittstellen

Ein Komponente besitzt eine Schnittstelle über die es für die anderen Softwareelemente zugreifbar ist. Somit expliziert die Schnittstelle alle extern sichtbaren Eigenschaften einer Komponente.
Hinter der Schnittstelle an sich, steht keine Implementierung ihrer Funktionen. Anstelle dessen, benennt die Schnittstelle lediglich die Namen der zur Verfügung gestellten Operationen und bietet stellt nur Beschreibungen und die Protokolle dieser Operationen zur Verfügung.

Die klare Trennung von Schnittstelle und Implementierung ermöglichtauf der einen Seite, dass die Implementierung ersetzt werden kann ohne die Schnittstelle zu ändern und auf diesem Wege kann z.B. die Systemleistung durch Einsatz einer performanteren Komponente verbessert werden ohne das System neu erstellen und aufsetzen zu müssen. Auf der anderen Seite erlaubt diese Trennung neue Schnittstellen (und Implementierungen) hinzuzufügen ohne die existierende Implementierung zu verändern und damit das Systems anzupassen. Verwender der Komponente können ihre Komponenten aufgrund der Schnittstellen an das System anpassen, da diese der einzig sichtbare Teil der Komponente sind.
Idealerweise ist nach Crnkovic [CL02] in einer Schnittstelle die Semantik jeder einzelnen zur Verfügung stehenden Operation beschrieben, da dies wichtig für Entwickler und Nutzer der Schnittstelle ist um die Komponente entsprechend nutzen zu können, was in der Praxis ist häufig nicht der Fall ist. In den meisten existierenden Komponentenmodellen legt die Schnittstelle nur die Syntax (z.B. Typen der Eingaben oder Ausgaben) fest und gibt nur sehr wenige Informationen über das, was die Komponente tut.
Schnittstellen die in Standard Komponententechnologien definiert sind, können meist nur funktionale Eigenschaften ausdrücken. Diese beeinhalten einen Signaturteil in dem die von der Komponente angebotenen Operationen beschrieben sind und einen Verhaltensteil, in dem das Verhalten einer Komponente beschrieben ist.
Viele der Beschreibungstechniken für Schnittstellen wie die Interface Definition Language (IDL) beschäftigen sich nur mit dem Signaturteil. Solche Schnittstellenbeschreibungstechniken sind nicht ausreichend um nicht-funktionale Eigenschaften wie beispielsweise Verfügbarkeit, Verzögerungszeiten oder Sicherheitsaspekte auszudrücken.

Es lassen sich zwei Arten von Schnittstellen unterscheiden: Zum Einen Exportschnittstellen und zum Anderen Importschnittstellen. Eine Importschnittstelle spezifiziert die Dienste, die von einer Umgebung der Komponente erforderlich sind, wohingegen eine Exportschnittstelle die von der Komponente bereitgestellten Dienste

beschreibt.

In Crnkovic [CL02] heißt es des weiteren, dass Schnittstellen gewöhnlich nur syntaktisch beschrieben werden – Probleme im Zusammenhang mit Kontextabhängigkeiten (z.B. Spezifikation der Deployment- oder Laufzeitumgebung) und Interaktionsverhalten hat die Notwendigkeit eines Vertrages (contract) hervorgebracht, der das Komponentenverhalten deutlich beschreibt.

3.2.1 Verträge (contracts)

Die meisten Techniken um Schnittstellen zu beschreiben, wie die IDL setzen sich nur mit dem Signaturteil auseinander in dem die Operationen der beschriebenen Komponente zur Verfügung gestellt werden und können nicht das allgemeine Verhalten der Komponente beschreiben. Eine präzisere Spezifikation des Komponentenverhaltens kann durch Verträge (contracts) erreicht werden. Ein Vertrag listet globale Einschränkungen auf, welche die Komponente einhält. Für jede Operation innerhalb der Komponente werden in dem Vertrag außerdem die Einschränkungen festgehalten, die vom Aufrufer der Komponente einzuhalten sind (Vorbedingungen, „preconditions“) und diejenigen, welche die Komponente als Rückgabe zusagt (Nachbedingungen, „postcondition“). Die Vorbedingung, globalen Einschränkungen und die Nachbedingung machen dann die Spezifikation des Komponentenverhaltens aus. In UML können diese Einschränkungen z.B. mittels der OCL (Object Constraint Language) ausgedrückt werden.
Verträge können auch genutzt werden um die Interaktionen von zusammengehörigen Komponenten zueinander zu beschreiben.

3.3 Komponentenmodell

Ein Komponentenmodell ist nach Sommerville [Som04] eine Definition von Standards für Komponentenimplementierung, -dokumentation und -deployment. Diese Standards sind für Komponentenentwickler gedacht um sicherzustellen, dass Komponenten zusammenarbeiten können. Sie sind ebenso für Anbieter der Komponenten-Infrastruktur gedacht, welche die Middleware zur Verfügung stellen um das Betreiben einer Komponente zu ermöglichen.

Es gibt verschiedene Ansätze für Komponentenmodelle – die wichtigsten Modelle sind das CORBA (Common Object Request Broker Architecture) Komponentenmodell der OMG, das Enterprise Java Beans (EJB) Modell von Sun und Microsoft's COM+ Modell. Die spezifischen Infrastrukturtechnologien, wie COM+ und EJB die im CBSE genutzt werden sind sehr komplex. Dies näher auszuführen sprengt allerdings den Rahmen dieser Arbeit. Ein Komponentenmodell spezifiziert unter Anderem, wie die Schnittstellen der Komponenten und die Elemente wie die zur Verfügung gestellten Operationen, Namen, Parameter etc. definiert sein sollten. Da

Komponenten generische Einheiten sind muss auch festgelegt werden, wie sie für den Betrieb angepasst werden können. Außerdem sollte im Komponentenmodell festgelegt werden, wie Komponenten bereitgestellt werden können (z.B. in Archiven mit Deskriptoren). Weiterhin sollte dort festgelegt sein, wie beispielsweise das Ersetzen von Komponenten durch andere Version möglich ist. Nicht zuletzt ist das Komponentenmodell auch Basis für die Middleware, welche die Ausführung der Komponenten erlaubt (und somit als Implementierung des Komponentenmodells bezeichnet werden kann).

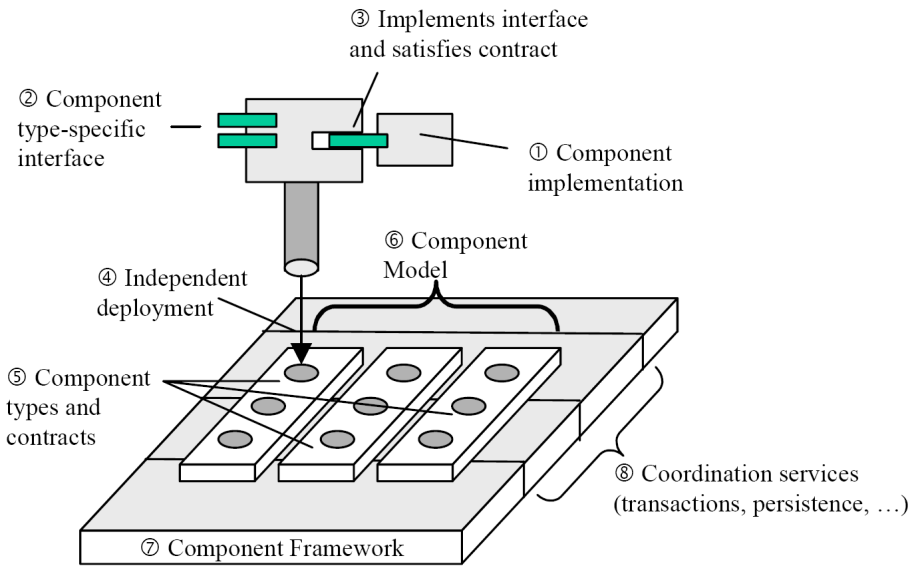

Abbildung 1: Schematische Darstellung für komponentenbasiertes Design (Quelle: [BBB+00])

Eine Komponente (1) ist eine Software Implementierung die ausgeführt werden kann. Eine Komponente Sie implementiert eine oder mehrere Schnittstellen die ihr typischerweise aufgezwungen werden (2). Die Komponente erfüllt somit einige Bedingungen, die als Vertrag (contract) beschrieben werden. Diese vertraglichen Bedingungen stellen sicher, dass unabhängig voneinander entwickelte Komponenten bestimmte Regeln einhalten, so dass Komponenten miteinander in beschriebenen Wegen interagieren (oder nicht interagieren) können und sie in Standard (Laufzeit)umgebungen bereitgestellt werden können (4). Ein komponentenbasiertes System basiert auf einer kleinen Anzahl verschiedener Komponententypen, von dem jede eine spezialisierte Rolle in dem System spielt (5) und von einer Schnittstelle (2) beschrieben wird. Ein Komponentenmodell (6) beschreibt eine Menge von Komponententypen, ihren Schnittstellen und zusätzlich einer Spezifikation erlaubter Interaktionsmuster zwischen Komponententypen. Ein Komponentenframework (7) stellt eine Vielzahl von Laufzeitdiensten (8) zur Verfügung um das Komponentenmodell zu unterstützen und es zur Geltung kommen zu lassen. In vielen Empfehlungen sind Komponentenframeworks wie Spezialbetriebssysteme zu sehen, obwohl sie auf einem wesenlich

höheren Abstraktionsniveau angesiedelt sind [BBB+00].

3.4 Der Entwicklungsprozess

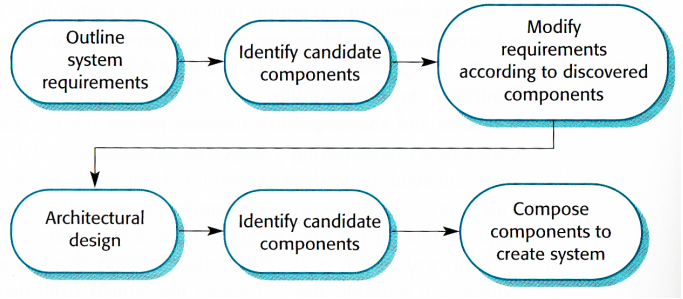

Abbildung 2: Der CBSE Entwicklungsprozess(Quelle: [Som04])

Der Entwicklungsprozess des CBSE nach [Som04] ist in Abbildung 2 zu sehen und wird an dieser Stelle kurz beschrieben. Der Prozess sieht im Wesentlichen vor zunächst die Anforderungen skizzieren und im Anschluss potentielle Komponenten zu identifizieren. Anschließend werden die Anforderungen entsprechend der festgestellten Komponenten angepasst es findet der Entwurf der Software Architektur statt. In dieser Phase wird wahrscheinlich eine Festlegung des Komponentenmodells stattfinden müssen. Danach können sich einige Komponenten möglicherweise als unbrauchbar herausstellen – daher müssen daraufhin wieder geeignete Komponenten identifizert werden. Die letztliche Systementwicklung findet dann mit der Integration/dem Zusammenbauen der Komponenten statt. Für die Möglichkeiten der Komposition sei hier auf weitere Literatur verwiesen (z.B. [Szy02]), da es den Rahmen dieses Ausblicks auf die komponentenbasierte Entwicklung sprengen würden.

3.5 Vergleich: Modul - Komponente

Komponenten sind standardisiert und in ein Komponentenmodell eingebettet. Module hingegen ergeben sich aus dem Gesamtsystem durch Dekomposition und dienen der Strukturierung. Module können auch Kopplungen zu anderen Modulen aufweisen, während Komponenten normalerweise unabhängig sein sollten. Module und Komponenten sind wiederverwendbar, kapseln Daten und erfüllen i.d.R. eine spezifische Aufgabe. Im Wesentlichen kann man wohl aber Komponenten als standardisierte und spezialisierte Module betrachten.

4 Unterstützung modularer Entwicklung

Wie in 2.2 teilweise schon beschrieben, gibt es in Programmiersprachen Möglichkeiten Module oder Schnittstellen zu beschreiben. Darüber hinaus sollen noch weitere Methoden gezeigt werden, die modulare Programmierung unterstützen können.

4.1 Entwurfsmuster

Entwurfsmuster (design patterns) beschreiben bewährte Lösungen für wiederkehrende Entwurfsprobleme. Die wohl bekanntesten Entwurfsmuster für objektorientierte Software wurden von Gamma et al. in [GHJV01] festgehalten. Es gibt aber weitere Entwurfsmuster z.B. zur Unterstützung der Softwarearchitektur.

Durch eine ausreichende Abstraktion des Problems ermöglichen Entwurfsmuster unter Umständen eine modularere Gestaltung des Gesamtsystems.
Im Folgenden sollen beispielhaft ein paar konkrete Entwurfsmuster aus [GHJV01] vorgestellt, die sinnvoll für den Einsatz in modularer Software sein können.

4.1.1 Facade

Das Facade (Fassade) Muster bietet eine einheitliche Schnittstelle für eine Menge von Schnittstellen in einem Subsystem. Dabei könnte es sich beispielsweise um eine Schnittstelle für ein identifiziertes Modul handeln, welches aus mehreren Komponenten zusammengesetzt ist. Durch die Zusammenfassung der Schnittstellen der Komponenten eines Moduls wird der Zugriff vereinfacht womit eine Reduktion der Komplexität einhergeht.
Durch die Facade können Änderungen im Modul vorgenommen werden ohne die Schnittstelle anpassen zu müssen.

4.1.2 Singleton

Mit dem Singleton Muster kann sichergestellt werden, dass es nur genau eine Instanz des eines Objektes geben kann. Wenn eine Komponente in besonderen Fällen in einem System nur ein mal existieren darf, so kann durch dieses Erzeugungsmuster sichergestellt werden, dass sie nur ein mal instantiiert wird.

4.1.3 Mediator

Das Mediator Muster wird eingesetzt um das Zusammenspiel mehrerer Objekte zu kapseln. Ein Vermittler fördert dabei die lose Kopplung, indem er Objekte davon abhält aufeinander explizit Bezug zu nehmen. Die Abhängigkeiten werden dadurch reduziert und der Austausch einzelner Komponenten aufgrund der Implementierung

vom Mediator spezifizierter und aufgerufener Schnittstellen erleichtert, was die Flexibilität erhöht.

4.2 Aspektorientierte Programmierung (AOP)

In der aspektorientierte Programmierung wird versucht die verschiedenen Aspekte eines Programms getrennt zu entwickeln um sie dann anschließend zu einem Programm zusammenzufügen. Die Aspekte werden dann z.B. vor Methodenaufrufen „eingewebt“ und dann an dieser Stelle ausgeführt. Das nachträgliche Einfügen der sogenannten „Aspekte“ in den Programmcode kann ebenfalls für Modularität sorgen, da zusammengehörige Einheiten (wie z.B. das Logging in Programmen) in einen Aspekt gekapselt werden können, welcher an mehreren Stellen verwendet werden kann. In Java ist das Einweben der Aspekte sowohl für die Anwendung im Quellcode als auch für Bytecode geeignet. Dies hat den großen Vorteil, dass Software auch nachträglich um bestimmte Aspekte erweitert werden kann.

4.3 Testen

Da die Testphase ein wichtiger Bereich des Entwicklungsprozesses ist, soll auch dies hier kurz erwähnt werden.

Bei der Entwicklung von Komponenten werden i.d.R. Komponententests („unit tests“) durchgeführt um die Funktionalität und das Verhalten der Komponenten sicherzustellen. Ein Komponententest z.B. mit J-Unit in Java ist ausführbarer Code, der als Regressionstest eingesetzt werden kann. In der testgetriebenen Entwicklung werden die Testfälle vor der Implementierung erdacht und sollten sicherstellen, dass sich das Verhalten der Komponenten innerhalb der Spezifikation/Anforderung befindet. Durch die Ausführbarkeit des Testfalles kann dem Entwickler schnell signalisiert werden, ob der Code/die Komponente bspw. nach einer Änderung immer noch das gewünschte Verhalten aufweist. Komponententests können auch Gegenstand der Auslieferung als Teil der Spezifikation von Komponenten sein.

Im Gegensatz zum Komponententest, der lediglich validiert, ob die Komponente sich bei einem als vollständig vorausgesetztem Testfall korrekt verhält, wird beim Integrationstest das Zusammenspiel mehrerer Komponenten gestestet. Ein Komponententest sollte Voraussetzung für einen Integrationstest sein, in dem ein möglichst vollständiges Testszenario ausgeführt wird um das Verhalten der Komponenten zueinander zu testen. Ein Problem taucht nämlich dann auf, wenn ein Integrationstest erfolgreich verlaufen ist, d.h. alle Komponenten wie erwartet zusammengearbeitet haben und eine Komponente nachträglich geändert wird. Die Annahme, dass die geänderte Komponente dann auch den Integrationstest besteht ist fatal, denn im Fehlerfall bzw. falls die Komponente unspezifizierte Werte liefert, kann das eine Gefahr für das Gesamtsystem (und nicht nur die Komponente) bedeuten. Daher ist

eine fehlertolerante Programmierung für die Komponenten modularer Software von Bedeutung, da andernfalls schwerwiegende Fehler das Gesamtsystem gefährden.

5 Fazit/Ausblick

Modulare Softwareentwicklung beschäftigt die Informatik schon länger. In der Literatur sind keine Patentrezepte zu finden, die es einem ermöglichen modulare Software zu entwickeln. Durch die Verbreitung objektorientierter Programmiersprachen und damit verbundener Techniken wie Kapselung und Konzepten wie „information hiding“ ist es sicher einfacher geworden modulare Software zu entwickeln – die Literatur gibt lediglich Kriterien an die Hand, einem dabei helfen können Entscheidungen zu treffen.

Auch der Ansatz komponentenbasierter Softwareentwicklung erleichtert die Entwicklung, da nur noch Systemteile entwickelt werden müssen die konform zu den Schnittstellen sind. Dadurch sind auch Anforderungen an komplexe Software leichter handhabbar. Weitere Vorteile wie größere Flexibilität durch den Austausch von Komponenten sind ebenfalls lohnenswert, dies bietet auch die Möglichkeit das System aktuell zu halten und langfristig die Betriebskosten zu senken. Aber auch der Test von Komponenten sollte nicht unterbleiben, denn dieses System funktioniert nur gut, wenn es innerhalb der Spezifikation betrieben wird.

Eine kurzer Einblick in die aspektorientierte Programmierung zeigt, dass es auch noch andere, aktuelle Ansätze gibt, die einen Einfluss auf die Art der modularen Programmierung haben können. Eine nähere Untersuchung bezüglich serviceorientierter Architekturen respektive ihrer Implementierungen mittels Webservices ist sicher interessant im Bezug auf Modularitätskriterien.

Literatur

[BBB+00] Bachmann, Felix, Len Bass, Charles Buhman, Santiago Comella-Dorda, Fred Long, John Robert, Robert Seacord und Kurt Wallnau: *Volume II: Technical Concepts of Component-Based Software Engineering, 2nd Edition.* Technischer Bericht, Carnegie Mellon Software Engineering Institute (SEI), 2000.

[BCK03] Bass, Len, Paul Clements und Rick Kazman: *Software Architecture in Practice (2nd Edition) (The SEI Series in Software Engineering).* Addison-Wesley Professional, 2003.

[CL02] Crnkovic, Ivica und Magnus Larsson: *Building Reliable Component-Based Software Systems.* Artech House Publishers, 2002.

[GHJV01] Gamma, Erich, Richard Helm, Ralph Johnson und John Vlissides: *Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software.* Addison-Wesley, 2001.

[HC01] Heineman, George T. und William T. Councill: *Component Based Software Engineering: Putting the Pieces Together.* Addison-Wesley Professional, 2001.

[Mül02] Müller, Peter: *Modular Specification and Verification of Object-Oriented Programs (Lecture Notes in Computer Science,).* Springer, Berlin, 2002.

[Par72] Parnas, D. L.: *On the criteria to be used in decomposing systems into modules.* Commun. ACM, 15(12):1053–1058, 1972.

[PCW84] Parnas, D. L., P. C. Clements und D. M. Weiss: *The modular structure of complex systems.* In: *ICSE '84: Proceedings of the 7th international conference on Software engineering*, Seiten 408–417, Piscataway, NJ, USA, 1984. IEEE Press.

[PS98] Pfister, Cuno und Clemens Szyperski: *Why objects are not enough.* In: *CUC '96: Proceedings of the first component user's conference on Component-based software engineering*, Seiten 141–147, New York, NY, USA, 1998. Cambridge University Press.

[Som04] Sommerville, Ian: *Software Engineering, 7th Edition.* Addison Wesley, 7 Auflage, 2004.

[Szy02] Szyperski, Clemens: *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley Professional, 2002.