1 Einleitung

Aktuelle Softwaresysteme müssen sich immer umfangreicheren funktionalen Anforderungen stellen, wobei gleichzeitig die nichtfunktionalen Anforderungen steigen. Die Zeit, die für die komplexen Enwicklungsprozesse zur Verfügung steht, ist durch den wachsenden Konkurrenzdruck begrenzt. Der entstehende Zeitdruck wird weiter verstärkt, da die Anforderungen selten zu Beginn eines Projektes festgeschrieben werden, sondern einem Änderungsprozess unterworfen sind. Im schlimmsten Fall müssen bereits fertig entwickelte funktionale Einheiten noch ausgetauscht werden. Unter der Einwirkung dieser Einflüsse auf die Entwicklung komplexer Software gewinnt die Wiederverwendung von Softwarebausteinen, so genannten Komponenten, mehr und mehr Bedeutung. Dabei geht man wie auch in der fertigenden Industrie davon aus, durch die Wiederverwendung vorgefertigter Bausteine viel Entwicklungsaufwand einsparen zu können. Zusätzlich zu der Einsparung an Zeit und Kosten erhofft man sich durch den Einsatz real erprobter Komponenten eine Steigerung der Qualität der Anwendungssysteme.
Bei der Suche nach geeigneten Komponenten für einen jeweiligen Anwendungskontext stellt man dabei fest, dass die Forderung exakter Übereinstimmung von gewünschter und bereits vorhandener Funktionalität zu strikt ist.
Also wird man an dieser Stelle eine unscharfe Suche zulassen und Komponenten ermitteln, die nicht von vornherein exakt in den Anwendungskontext passen. Dabei ist allerdings immer wieder zu beobachten, dass die Wiederverwendung nicht genau passender Komponenten sowohl das Risiko lokaler Inkonsistenzen zwischen Paaren kooperierender Komponenten, als auch das Risiko globaler Inkonsistenzen zwischen den bewährten

Komponenten und ihrem neuen Anwendungskontext birgt.
Um nun also eine ökonomische Wiederverwendung bestehender Komponenten in neuen Systemen betreiben zu können, besteht Bedarf an systematischer Unterstützung. Diese muss Kandidaten zur Wiederverwendung identifizieren, sie auf ihre Eignung im jeweiligen Anwendungskontext hin untersuchen und gegebenenfalls auftretende Unstimmigkeiten zwischen Anwendungskontext und Komponenten überbrücken.
Nur wenn diese Schritte automatisiert unterstützt werden, kann das Einsparungspotential der Wiederverwendung voll ausgeschöpft werden.

1.1 Zielsetzung

Zielsetzung dieser Arbeit ist es daher, Art und Ursache von Inkonsistenzen zu untersuchen und auf Basis der Untersuchungsergebnisse eine ausreichend genaue Beschreibungsmöglichkeit für komponentenbasierte Anwendungen zu erarbeiten.
Als weiteres Ziel soll die Beschreibungsform für die automatisierte Erzeugung von Schutzmechanismen nutzbar sein: Die auf Basis der zusätzlichen Information erzeugten Schutzmechanismen verhindern ein Auftreten von inkonsistenzbasierten Fehlern, das daraus resultierende Versagen der Software und damit eventuelle kritische Ausfälle mit wirtschaftlichen oder gar personellen Schäden.
Dabei sollen sowohl die Beschreibung selbst als auch das Werkzeug zur Erzeugung der Schutzmechanismen so gestaltet werden, dass sie mit geringem Mehraufwand im industriellen Umfeld einsetzbar sind, um das Einsparungspotential durch die Wiederverwendung optimal zu nutzen.

1.2 Gliederung der Arbeit

Die vorliegende Arbeit liefert in Kapitel 2 mit einer Einführung in die komponentenbasierte Softwareentwicklung notwendige Grundlagen. Diese Einführung schließt mit einer eingehenden Betrachtung historischer Unfälle, an denen jeweils der Ausfall eines Softwaresystems ursächlich beteiligt

war. Die Ursachen der Unfälle werden nach einer Analyse in Klassen von Integrationsfehlern eingeteilt.
Die Analyseergebnisse sowie diese Klassifikation bilden die Grundlage des in Kapitel 3 vorgestellten formalen Modells zur Beschreibung von Softwarekomponenten und aus ihnen aufgebauten Anwendungssystemen. Die so dargestellte Information ist zwar präzise und würde für diese Arbeit ausreichen, ist durch ihren formalen Charakter aber nur schwierig in reale Entwicklungsabläufe zu integrieren. Kapitel 4 befasst sich aus diesem Grund damit, eine in der Industrie verbreitet eingesetzte Beschreibungssprache (**U**nified **M**odeling **L**anguage, UML) so zu erweitern, dass die im formalen Modell geforderten Informationen erfasst werden können.
Nach der Charakterisierung der Information und ihrer Notation wird in Kapitel 5 die Konzeption und Umsetzung eines Werkzeugs beschrieben, das die Beschreibung komponentenbasierter Anwendungen auf mögliche Probleme hin untersucht und Programmcode erzeugt, der die erkannten Probleme weitgehend verhindern kann.
Kapitel 6 enthält eine Vorgehensbeschreibung für den Einsatz des Werkzeugs in konkreten Einsatzgebieten.
In Kapitel 7 wird schließlich ein Fazit gezogen und der Ausblick auf mögliche Weiterführungen der Arbeit gegeben.

2 Inkonsistenzen in komponentenbasierten Softwaresystemen

In diesem Kapitel wird diskutiert, warum komponentenbasierte Softwareentwicklung betrieben wird und wo die Risiken liegen, die dabei auftreten. Im Anschluss daran wird ein spezielles Risiko, das der Inkonsistenzen in komponentenbasierten Systemen, eingehender betrachtet.

2.1 Klassische Softwareentwicklung

Klassische Softwareentwicklung ist dadurch charakterisiert, dass jeweils ausgehend von der Anforderungsdefinition eines konkreten Systems hin zu einer dem konkreten Problem genau entsprechenden Lösung gearbeitet wird (siehe dazu z. B. [1]). Die dabei entstehende Lösung zeichnet sich meist dadurch aus, dass sie exakt auf den gegebenen Einsatzzweck zugeschnitten ist und alle dort vorliegenden Rahmenbedingungen berücksichtigt. Sie ist typischerweise monolithisch aufgebaut.
Monolithische Software hat u.a. zwei entscheidende Nachteile: Zum einen lassen sich aus ihr nur mit erheblichem Aufwand Teile herauslösen, um sie entweder aufgrund guter Erfahrungen in anderen Umfeldern zu nutzen, oder aufgrund schlechter Erfahrungen im Rahmen der Wartung auszutauschen (vgl. z. B. [2]).
Zum anderen können solche Monolithe auch kaum als komplette Blöcke wiederverwendet werden, weil sie zu spezifisch für ihren ursprünglichen Einsatzzweck sind.

Diese Nachteile klassischer Softwareentwicklung gewinnen vor dem Hintergrund steigender Anforderungen und kürzerer Entwicklungszeiten mehr Gewicht. Insbesondere im Bereich von Software-Produktfamilien, der steigende Bedeutung erfährt (siehe auch [3]), wirken sich die Probleme bei der Wiederverwendung besonders schwer aus.
Die Konzepte, mit denen in der klassischen Softwareentwicklung Wiederverwendung betrieben wird, wie z. B. prozedurale Bibliotheken oder objektorientierte Klassenpakete, eignen sich ebenfalls nur begrenzt zur Wiederverwendung. Beide stellen zwar mehr oder weniger gut dokumentierte Funktionalität zur Verfügung, aber beiden fehlt ein Beschreibungsmechanismus, der darstellt, welche Anforderungen an die Einsatzumgebung gestellt werden und wie die Abhängigkeiten zur Umgebung explizit aussehen.
Legt man der Softwareentwicklung die Zielsetzung der Wiederverwendung zugrunde, ist es also naheliegend, die Softwaresysteme aus Modulen zu konstruieren, die keine impliziten Abhängigkeiten mehr zum Kontext oder zu anderen Modulen haben. Derartig gekapselte Module werden als Komponenten bezeichnet.

2.2 Komponentenbasierte Softwareentwicklung

> „A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.“ [4]

Um Software zu erstellen, die aus derartigen Bausteinen aufgebaut ist, muss der Entwicklungsprozess angepasst werden. Es haben sich auch bereits einige solcher Prozessmodelle etabliert, wie etwa [5] oder [6].
Die entscheidenden Punkte, an denen die Prozessmodelle sich von den klassischen unterscheiden, sind:

1. Analysephase
   In der Analysephase wird nach der Identifikation der logischen Module nicht direkt deren Entwicklung angeschlossen, sondern es wird

erst eine Komponentenanalyse durchgeführt, welche der Module durch bereits bestehende Komponenten realisiert werden können und welche der Module eventuell dafür geeignet sind, als Komponente entwickelt zu werden.

Hierbei steht zum einen die Suche in Repositorien[1] im Vordergrund, bei der es entscheidend ist, nicht nur exakte Treffer auszuwerten, sondern auch für die jeweiligen Anwendungszwecke annähernd passende Ergebnisse zu liefern (vgl. [7]).

Zum anderen müssen für die Module, für die noch keine Komponenten bestehen, Überlegungen hinsichtlich der Kohäsion der Komponenten sowie ihrer Kopplung zum Rest des Systems angestellt werden. Lediglich Module, die über eine hohe Kohäsion verfügen, sind geeignet, in neuen Systemumgebungen eingesetzt zu werden. Schließlich wird eine Kombination aus mehreren Funktionalitäten, von der bei niedriger Kohäsion auszugehen ist, mit einer sehr viel geringeren Wahrscheinlichkeit wiederverwendbar sein als die Realisierung genau einer Funktionalität (siehe dazu [8]).

Der Grad der Kopplung eines Moduls zum restlichen System ist in dem Maße entscheidend, in dem eine hohe Kopplung auf viele Kontextabhängigkeiten schließen lässt, die ihrerseits im Falle einer Wiederverwendung schwer zu erfüllen sind.

2. Implementierungsphase
   Bei der Implementierung der fehlenden Module ist verstärkt darauf zu achten, sämtliche Kontextabhängigkeiten explizit festzuhalten.

3. Integrationsphase
   In einem zusätzlichen Schritt, der Integrationsphase, müssen die Komponenten miteinander verbunden und für das vorliegende System konfiguriert werden. Dazu wird in den Prozessmodellen häufig eine zusätzliche Rolle, der Integrator (vgl. dazu [9]), definiert.

[1] Als Repositorien werden Datenbanken bezeichnet, die zur Speicherung bestehender Komponenten dienen.

Der Integrator muss zunächst die Komponenten, wie im Entwurf vorgesehen, miteinander verbinden und dann feststellen, ob die vorliegende Auswahl an Komponenten korrekt zusammenarbeitet, wobei der Prozess der Integration durchaus komplex sein kann (vgl. [10]). Die Probleme, die dabei zwischen den Komponenten auftreten, müssen ohne genauere Kenntnis der einzelnen Komponenten verstanden und auf die korrekte Ursache zurückgeführt werden. Menschliches Zutun ist dabei noch unabdingbar, da bei Konflikten zwischen Komponenten nicht ohne weiteres die Ursache gefunden werden kann.

4. Testphase
   Die Testphase wird durch einen speziell auf die Prüfung der Komponenteninteraktion ausgerichteten, so genannten Integrationstest ergänzt. Im Integrationstest kann anhand einiger exemplarischer Szenarien getestet werden, ob Probleme bei der Zusammenarbeit einzelner Komponenten auftreten. Allerdings ist der Integrationstest, wie jede Form von Test, auf die vorliegenden Szenarien beschränkt und kann somit lediglich die Anwesenheit von Fehlern zeigen, nicht deren Abwesenheit. Systematisch können somit mögliche Inkonsistenzen durch Tests in der Integrationsphase nicht ohne weiteres aufgedeckt werden, hier muss Wissen über die Komponenten (typischerweise über deren innere Zustände und die Art der Interaktion) eingesetzt werden, um ausreichende Testszenarien zu erzeugen (siehe z. B. [11]).

Bei der Entwicklung komponentenbasierter Systeme lässt sich feststellen, dass wiederverwendete Komponenten aus unterschiedlichen Systemen kaum fehlerfrei zusammenarbeiten. Der Integrationstest ist zwar dafür geeignet, solche Fehler aufzudecken, doch wäre es wesentlich günstiger, die Fehler früher im Prozess zu finden und direkt zu beheben (vgl. [12]).
Um die Problematik der Inkonsistenzen systematisch angehen zu können, wird zunächst analysiert, welcher Art die Inkonsistenzen sein können, die beim Einsatz von Komponenten in neuen Anwendungskontexten auftreten.