Design for Future – Langlebige Softwaresysteme

1. Workshop des GI-Arbeitskreises „Langlebige Softwaresysteme (L2S2)“

15.-16. Oktober 2009

FZI Forschungszentrum Informatik, Karlsruhe

Gregor Engels, s-lab, Uni Paderborn
Ralf Reussner, FZI, Uni Karlsruhe (TH)
Christof Momm, FZI, Uni Karlsruhe (TH)
Stefan Sauer, s-lab, Uni Paderborn

Inhaltsverzeichnis

Vorwort

Software altert auch! Dieses Problem ist vor allem bei großen betrieblichen Informationssystemen unter dem Begriff *Legacy* bekannt und wird sich in Zukunft noch weiter verschärfen. Zum einen gewinnen eingebettete Systeme immer größere Bedeutung, in denen aufwändige Software in langlebigen technischen Geräten eingesetzt wird. Zum anderen macht die steigende Vernetzung von Systemen in großen Anwendungslandschaften die Situation zunehmend komplexer. Diese Probleme haben enorme ökonomische Bedeutung. Wissenschaft und Industrie sind aufgefordert, neue Methoden der Softwaretechnik zu entwickeln, um die erheblichen Investitionen in große Softwaresysteme zu schützen und massive Probleme durch steigende Software-Erosion zu verhindern.

Aktuelle Ansätze in der Softwaretechnik, insbesondere in den Bereichen Softwarearchitektur, Requirements Engineering und Reengineering, können dazu beitragen, die Situation zu verbessern, wenn sie geeignet weiterentwickelt und angewandt werden. Der neu gegründete Arbeitskreis „Langlebige Softwaresysteme (L2S2)“ der Fachgruppe Softwarearchitektur der Gesellschaft für Informatik e.V. (GI) hat sich zum Ziel gesetzt, Wissenschaftler und Praktiker im deutschsprachigen Raum zusammenzubringen, die an diesen Themenstellungen Interesse haben. In diesem 1. Workshop des Arbeitskreises werden die oben geschilderte Entwicklung, Erfahrungen hierzu sowie Lösungsansätze sowohl aus praktischer als auch aus wissenschaftlicher Sicht beleuchtet. Abgerundet wird das Programm des Workshops durch zwei eingeladene Vorträge von Herrn Henning Butz, Leiter des Entwicklungsbereichs „Information Management & Electronic Networks“ bei Airbus Deutschland GmbH, Hamburg, sowie von Herrn Dr. Uwe Dumslaff, Vorstand der Capgemini sd&m AG, München.

An dieser Stelle danken wir als Veranstalter des Workshops insbesondere den eingeladenen Sprechern sowie den Autoren der eingereichten und akzeptierten Beiträge für Ihr Interesse an dem Workshop. Außerdem danken wir allen Mitgliedern des Programmkomitees für ihre sorgfältige und konstruktive Arbeit bei der Auswahl des Programms für den Workshop.

Paderborn, Karlsruhe, Oktober 2009

Gregor Engels, s-lab, Uni Paderborn
Ralf Reussner, FZI, Uni Karlsruhe (TH)
Christof Momm, FZI, Uni Karlsruhe (TH)
Stefan Sauer, s-lab, Uni Paderborn

Workshopleitung

Ralf Reussner, FZI, Uni Karlsruhe (TH)

Programmkomiteevorsitz

Gregor Engels, s-lab, Uni Paderborn

Organisationskomitee

Christof Momm, FZI, Uni Karlsruhe (TH)
Stefan Sauer, s-lab, Uni Paderborn

Programmkomitee

Manfred Broy, TU München
Michael Goedicke, Uni Duisburg-Essen
Ursula Goltz, TU Braunschweig
Andrea Herrmann, TU Braunschweig
Christof Momm, FZI, Uni Karlsruhe (TH)
Florian Matthes, TU München
Barbara Paech, Uni Heidelberg
Klaus Pohl, Uni Duisburg-Essen
Andreas Rausch, TU Clausthal
Ralf Reussner, FZI, Uni Karlsruhe (TH)
Matthias Riebisch, TU Illmenau
Stefan Sauer, s-lab, Uni Paderborn
Klaus Schmid, Uni Hildesheim
Andreas Winter, Uni Oldenburg
Heinz Züllighoven, Uni Hamburg

Eingebettete Software in Flugzeugsystemen lebt lange und sicher. Wie macht sie das, trotz sinkender Lebens(er)haltungskosten?

Henning Butz

AIRBUS Deutschland GmbH
Information Management & Electronic Networks EDYND
Kreetslag 10
21129 Hamburg
henning.butz@airbus.com

Abstract: Seit gut 25 Jahren werden Flugzeugsystemfunktionen − exponentiell zunehmend – durch Software realisiert. Mitte der 80er Jahre verarbeiteten Bordcomputer 5-10 MByte Code, heute sind es mehr als 600 MByte. Die lange Lebensspanne von Flugzeugen bedingt notwendigerweise eine entsprechende Nachhaltigkeit der implementierten Software, trotz deutlich kürzerer Lebenszyklen der zugrunde liegenden Rechner- und Informationstechnologie. Die Flugzeugsystemtechnik hat demgemäß Entwicklungsprozesse, -plattformen und Technologien hervorgebracht, die sowohl die Wiederverwendung wie auch die Modifikation und Erweiterung von Softwarefunktionen auf hohem Sicherheitsniveau und komplexer Interoperabilität zuverlässig gewährleisten. Der Beitrag stellt diese Verfahren und Techniken vor und diskutiert notwendige F&T-Aufgaben, um die bestehende Qualität flugtauglicher Softwarefunktionen auch zukünftig nachhaltig zu gewährleisten.

Application Life Cycle Management: Langlebige betriebliche Informationssysteme entwickeln, warten und pflegen

Uwe Dumslaff

Capgemini sd&m AG
Carl-Wery-Straße 42
81739 München
uwe.dumslaff@capgemini-sdm.com

Abstract: Für betriebliche Informationssysteme wird jeden Tag mehr Code in den Betrieb genommen als abgestellt. Eine Mischung aus Host-Systemen, Client-Server- und Web-Anwendungen sowie erste Resultate der diversen SOA-Programme prägen Anwendungslandschaften von Unternehmen und Organisationen. Hinsichtlich Wartung und Pflege von Anwendungen wurden große Fortschritte sowohl in der Theorie als auch in der Praxis gemacht. Aber das Verhältnis von 20% Entwurfs- und Baukosten zu 80% Betriebs- und Pflegekosten macht deutlich, dass hier hinreichend Bedarf für weitere Verbesserungen ist. Jeder eingesparte Cent in der Betriebs- und Pflegephase kann für Innovationsprojekte eingesetzt werden, die uns alle voranbringen. Wartung und Pflege mittelmäßig strukturierter Anwendungen hemmt den Fortschritt!

Ähnliche, aber bei weitem komplexere Anforderungen kommen auf der Abstraktionsebene der Anwendungslandschaften auf uns zu. Die in den letzten Jahren dominierenden serviceorientierten Architekturprinzipien haben das Bauen forciert, aber weitere noch nicht gelöste Wartungsprobleme generiert. Bleiben sie ungelöst, werden sie durch die nächste Entwicklung hin zu SaaS (Software as a Service) und Cloud Computing noch verstärkt werden. Deutliche Indikatoren für die Qualitätsdefizite in der industriellen Praxis sind die Anfragen nach der Auslagerung von ganzen Teilen einzelner Anwendungslandschaften, die wegen der Wirtschaftskrise verstärkt auftreten.

Eine Antwort aus der industriellen Praxis auf die Herausforderungen einer kontinuierlichen Wartung und Verbesserung von Anwendungen und Anwendungslandschaften ist das Application Life Cycle Management. Hierbei geht es um das Denken und Gestalten sowohl auf Anwendungs- als auch auf Anwendungslandschaftsebene nach Prinzipien des modernen Software Engineering. Der Vortrag wird zeigen, wo wir heute stehen und wie viel Handlungsbedarf für Forschung und Industrie noch bestehen, um zukünftig langlebige Anwendungen und Anwendungslandschaften zu entwickeln und über lange Zeit zu pflegen.

Quelltextannotationen für stilbasierte Ist-Architekturen

Petra Becker-Pechau

Arbeitsbereich Softwaretechnik, Universität Hamburg
und C1 WPS GmbH
Vogt-Kölln-Straße 30, 22527 Hamburg
becker@informatik.uni-hamburg.de

Abstract: Langlebige Softwaresysteme müssen vielfach an sich ändernde Bedingungen angepasst werden. Dabei sollte jederzeit Klarheit über den aktuellen Zustand der Softwarearchitektur bestehen. Dieser Artikel präsentiert einen Ansatz, Quelltexte so mit Architekturinformationen anzureichern, dass die aktuelle Ist-Architektur alleine aus dem annotierten Quelltext extrahiert werden kann. Da Architekturstile die kontrollierte Evolution von Softwaresystemen unterstützen, betrachtet dieser Artikel stilbasierte Architekturen.

1 Motivation

Die Ist-Architektur von Softwaresystemen muss bekannt sein, um die Systeme kontrolliert ändern zu können. Nur, wenn die Ist-Architektur bekannt ist, kann sichergestellt werden, dass Änderungen sich an die geplante Architektur halten. Und nur, wenn die Ist-Architektur bekannt ist, können Entwicklungsteams ihre Systeme auf der abstrakteren Ebene der Architektur verstehen, kommunizieren und planen, ohne die Details des Quelltextes betrachten zu müssen. Doch die Ist-Architektur ist nicht eindeutig im Quelltext zu finden [RH09]. Das Problem lässt sich anhand eines Beispiels verdeutlichen. Unser Beispielsystem enthält die drei Klassen `PatientenAufnahmeToolGui`, `PatientenAufnahmeTool` und `Patient`. Es handelt sich um ein reales Beispiel aus einem kommerziellen Softwaresystem. Abbildung 1 zeigt die Klassen und ihre Referenzen. Das System hat als Architekturvorgabe, dass der `Patientenaufnehmer` den `Patienten` benutzen darf, der `Patient` jedoch den `Patientenaufnehmer` nicht kennen darf. Abbildung 2 zeigt die gewünschte Architektur.

Das Problem ist: dem Quelltext ist nicht eindeutig anzusehen, welche Klassen zu welchen Architekturelementen gehören. Wie wirkt sich also die vorgegebene Architektur auf den Quelltext aus? Hierzu fehlt die Information, dass die Klassen `PatientenAufnahmeTool` und `PatientenAufnahmeToolGui` zusammen den Patientenaufnehmer bilden, während die Klasse `Patient` dem Architekturelement Patient zugeordnet ist. Abbildung 3 verdeutlicht diesen Zusammenhang.

Erst mit dieser Information wird klar, was die Architekturvorgabe auf Klassenebene bedeutet: Die Klasse `Patient` darf weder das `PatientenAufnahmeTool` noch die

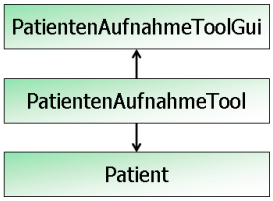


Abbildung 1: Klassendiagramm

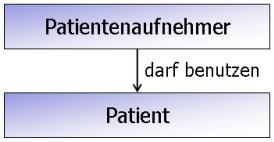


Abbildung 2: Soll-Architektur

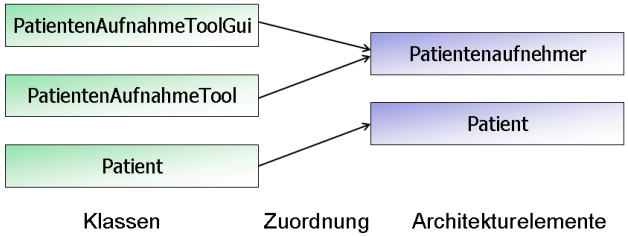


Abbildung 3: Zuordnung von Klassen und Architekturelementen

`PatientenAufnahmeToolGui` referenzieren. Beziehungen in der umgekehrten Richtung sind jedoch erlaubt.

Das Beispiel macht deutlich, dass zusätzlich zum Quelltext weitere Informationen benötigt werden, um die Ist-Architektur von Softwaresystemen zu ermitteln. Langlebige Softwaresysteme werden im Laufe ihres Einsatzes mehrfach überarbeitet und weiterentwickelt. Es genügt also nicht, die zusätzlich zum Quelltext benötigten Architekturinformationen zu Entwicklungsbeginn zu dokumentieren. Stattdessen muss diese Dokumentation permanent auf dem aktuellen Stand gehalten werden. Bei jeder Änderung des Quelltextes ist zu prüfen, ob sie noch korrekt ist.

Dieser Artikel adressiert das geschilderte Problem auf Basis von Quelltextannotationen, die zusätzliche Architekturinformationen in den Quelltext bringen. Wir halten diese Informationen direkt im Quelltext fest, um die Gefahr zu verringern, dass Quelltext und Architekturdokumentation auseinanderlaufen. Dabei konzentrieren wir uns auf statische Architekturen, auf die Modulsicht [HNS00]. Der Artikel liefert folgende Beiträge:

- Wir zeigen auf, welche Anteile der Ist-Architektur aus dem Quelltext entnommen werden können und welche zusätzlichen Informationen benötigt werden (siehe Abschnitt 3).
- Da Architekturstile für langlebige Softwaresysteme besonders relevant sind (siehe Abschnitt 2), diskutieren wir darüber hinaus, welche zusätzlichen Informationen für stilbasierte Ist-Architekturen notwendig sind (siehe Abschnitt 3).
- Wir zeigen, an welchen Stellen im Quelltext welche Architekturinformationen ein-

gefügt werden müssen. (Abschnitt 4).

- Abschnitt 5 präsentiert eine beispielhafte Umsetzung mit Java-Annotations. Mit Hilfe eines Prototyps, der die stilbasierte Ist-Architektur von Softwaresystemen ermittelt, konnten wir die Machbarkeit unseres Ansatzes zeigen.

2 Hintergrund

Um den fachlichen Hintergrund zu verdeutlichen, diskutiert dieser Abschnitt die Bedeutung von Architekturstilen für langlebige Softwaresysteme und gibt einen kurzen Überblick über die grundlegenden Begriffe.

2.1 Bedeutung von Architekturstilen für langlebige Softwaresysteme

Architekturstile machen Aussagen darüber, wie Software-Architekturen prinzipiell strukturiert werden sollen [RH09]. Sie legen fest, welche Strukturen erlaubt und welche Strukturen ausgeschlossen sind [GAO94, Kru95]. Insbesondere für langlebige, evolvierende Softwaresysteme bietet der Einsatz von Architekturstilen umfangreiche Vorteile: Architekturstile unterstützen konsistente und verständliche Architekturen, da verschiedene Systemversionen nach den selben Prinzipien strukturiert werden, selbst wenn die Architektur evolviert. Im Projektverlauf tragen sie maßgeblich dazu bei, die Komplexität von Softwaresystemen zu bewältigen [Lil08]. Für einige Unternehmen haben hauseigene Stile sogar eine strategische Bedeutung – beispielsweise für Capgemini sd&m der Quasar-Stil [Sie04] oder für die C1 WPS der Stil des Werkzeug-und-Material-Ansatzes (WAM-Ansatz) [Zül05]. Stile können auf bestimmte Systemarten ausgerichtet sein [KG06]. Ihre Architekturelement-Arten werden als ein Vokabular für die Architekturmodellierung verstanden [GS94, KG06]. Softwareteams können damit leichter über ihr Softwaresystem kommunizieren, da das Vokabular zur Systemart passt.

Unser Ansatz erlaubt es, die stilbasierte Ist-Architektur in der Entwicklungsumgebung sichtbar zu machen, als zweite Sicht auf den Quelltext. So brauchen die Softwareteams nicht mehr zwischen den Konstrukten der Programmiersprache und ihren eigenen Architekturkonstrukten zu übersetzen.

2.2 Grundlegende Begriffe

Betrachten wir das einleitende Beispiel, so lässt sich nachvollziehen, dass im Rahmen dieses Artikels unter einer *Software-Architektur* die Architekturelemente und deren Beziehungen verstanden werden. Das Verständnis ist angelehnt an bestehende Definitionen des Begriffs, wobei je nach Definition der Begriff auch weiter gefasst wird, als es für diesen Artikel nötig ist (vgl. [RH09, BCK03]).

Unter einer *Ist-Architektur* verstehen wir eine Architektur, die – soweit möglich – anhand eines Quelltextes ermittelt wurde. Wie genau unser Ansatz den Zusammenhang zwischen Ist-Architektur und Quelltext herstellt, erläutern die folgenden Abschnitte.

Stilbasierte Architekturen werden anhand eines Architekturstils entworfen. Das obige Beispiel stammt aus einem System, das auf dem WAM-Stil basiert. In diesem Stil gibt es unter anderem die Architekturelement-Arten "Werkzeug" und "Material". Werkzeuge dienen zur Benutzerinteraktion, mit ihnen werden Materialien bearbeitet. Materialien repräsentieren fachliche Gegenstände und werden Teil des Arbeitsergebnisses. Im WAM-Stil gilt, dass jedes Werkzeug ein Material referenzieren soll, Materialien jedoch nicht auf Werkzeuge zugreifen dürfen. Im Beispiel ist das Architekturelement namens Patientenaufnehmer ein Werkzeug, das Architekturelement namens Patient ist ein Material (siehe Abbildung 4).

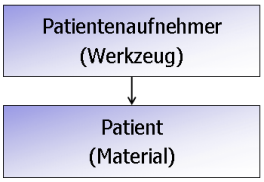


Abbildung 4: Stilbasierte Software-Architektur (Beispiel)

Architekturstile bewegen sich auf einer Metaebene zur Architektur, sie legen fest, aus welchen Architekturelementarten eine Architektur bestehen soll und definieren Regeln für Architekturelemente abhängig von ihrer Elementart. Um die stilbasierte Ist-Architektur zu ermitteln, müssen die Elementarten des zugehörigen Stils bekannt sein. Abbildung 5 zeigt ein Metamodell für stilbasierte Architekturen im Zusammenhang mit den Architekturelementarten. Formal verstehen wir unter einer stilbasierten Architektur eine Architektur, deren Architekturelemente einer Elementart zugeordnet sind.

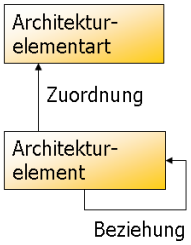


Abbildung 5: Stilbasierte Software-Architektur (Metamodell)

Dem Quelltext ist ohne Zusatzinformation nicht eindeutig anzusehen, wie die Klassen und andere Quelltextelemente den Architekturelementen zugeordnet sind. Es fehlt auch die Information, zu welcher Art die Architekturelemente gehören. Es existieren verschiedene Ansätze, diese zusätzlichen Informationen zu ermitteln:

- Der schwierigste Fall liegt vor bei Altsystemen, für die keine Architektur-Dokumen-

tation existiert und das Entwicklungsteam nicht mehr verfügbar ist. Dann müssen Reengineering-Techniken basierend auf Heuristiken verwendet werden. Dies kann manuell oder (teilweise) werkzeuggestützt erfolgen. Abhängig von der gewählten Heuristik liefern diese Techniken unterschiedliche Ist-Architekturen für ein und dasselbe Softwaresystem, da in diesen Fällen die ursprüngliche Intention des Entwicklungsteams nicht mehr bekannt ist. Man ist auf "Vermutungen" angewiesen [CS09].

- Ist die Architektur hingegen dokumentiert, so kann versucht werden, anhand der Dokumentation und des Quelltextes die Ist-Architektur zu rekonstruieren. In diesem Fall ist die Intention der Entwickler zumindest teilweise verfügbar. In der Praxis gibt es einige Stolpersteine. Es ist schwierig, die Dokumentation jederzeit konsistent zum Quelltext zu halten. Hinzu kommt, dass die Dokumentation meist informeller Natur ist, sie muss interpretiert werden.
- Sind noch Mitglieder des Entwicklungsteams verfügbar, so können diese nach ihrer ursprünglichen Intention gefragt werden. Doch gerade bei langlebigen Softwaresystemen ist es unrealistisch zu erwarten, dass die Entwickler beispielsweise für jede Klasse noch wissen, zu welchem Architekturelement sie gehören sollte. Auch ist sich das Entwicklungsteam nicht immer einig.

In der Praxis treten diese Ansätze meist gemischt auf. Wird beispielsweise die Ist-Architektur für eine Architektur-Konformanzprüfung benötigt, so müssen die zusätzlich zum Quelltext benötigten Informationen manuell im Prüfungswerkzeug beschrieben werden. Dies ist beispielsweise der Fall bei der Sotograph-Familie [BKL04], SonarJ[1], Lattix [SJSJ05] und Bauhaus[2]. Sofern noch Mitglieder des Entwicklungsteams verfügbar sind, so wird die Prüfung sicherlich von einem Mitglied durchgeführt oder unterstützt. Vorhandene Dokumentation wird genutzt, die Quelltextkommentare werden gelesen und es werden manuelle Heuristiken anhand von Bezeichnern und Quelltextstruktur verwendet. Wird das weiterentwickelte System mit demselben Werkzeug später erneut geprüft, so kann die in das Werkzeug eingegebene Beschreibung wiederverwendet werden. Wurde die Architektur des Systems jedoch geändert oder erweitert, so muss gegebenenfalls auch die Beschreibung manuell geändert werden.

Mit unserem Ansatz lassen sich stilbasierte Ist-Architekturen, wie in Abbildung 4, aus annotiertem Quelltext berechnen. Soll zu einem mit unserem Ansatz annotierten System die Ist-Architektur berechnet werden, so werden keine Heuristen benötigt, keine externe Dokumentation. Es ist nicht nötig, Mitglieder des Entwicklungsteams zu befragen. Stattdessen wird bereits bei der Entwicklung direkt im Quelltext vermerkt, wie die Quelltextelemente aus Architektursicht einzuordnen sind. Details stellen die Abschnitte 4 und 5 vor. Dieser Ansatz hat den Vorteil, dass die Intention der Entwickler direkt bei der Programmierung festgehalten wird. Natürlich ist auch hier nicht komplett sicherzustellen, dass immer korrekt annotiert wird, aber die Gefahr, dass die Architektur-Dokumentation und der Quelltext auseinanderlaufen, ist geringer.

Informationen über den Entwurf in den Quelltext zu bringen, ist nicht vollständig neu, jedoch für Architekturinformationen noch nicht sehr verbreitet. Der Ansatz gewinnt jedoch

[1] www.hello2morrow.com
[2] www.bauhaus-stuttgart.de

an Bedeutung (siehe Abschnitt 6). Neu an unserem Ansatz ist, dass wir es ermöglichen, *stilbasierte* Ist-Architekturen aus annotiertem Quelltext zu extrahieren. Stilbasierten Ist-Architekturen helfen das System zu verstehen und weiterzuentwickeln, darüber hinaus dienen sie als Basis für Konformanzprüfungen [BP09].

3 Stilbasierte Ist-Architekturen

3.1 Formale Definition

Dieser Abschnitt enthält die formale Definition für stilbasierte Ist-Architekturen. Das Metamodell für stilbasierte Architekturen zeigt bereits die Abbildung 5. Der Begriff der stilbasierten Ist-Architektur wurde im Zusammenhang mit der stilbasierten Architekturprüfung [BP09] eingeführt. Die für diesen Artikel relevanten Aspekte werden hier kurz wiederholt. Formal umfasst eine *stilbasierte Ist-Architektur* folgende Mengen und Relationen:

- die Menge der Architekturelemente E,
- eine Relation $B_I \subseteq E \times E$, die die bestehende Abhängigkeitsbeziehung zwischen diesen Architekturelementen beschreibt (kurz "Ist-Beziehungen"),
- die Relation $Z_A \subseteq E \times A$, mit der die Architekturelemente den Architekturelementarten zugeordnet werden. Diese Relation ist rechtseindeutig.

Darüber hinaus wird die Menge A der Architekturelementarten benötigt, um Z_A definieren zu können. Die Architekturelementarten ergeben sich aus dem gewählten Architekturstil. Im oben genannten WAM-Stil beispielsweise stehen unter anderem die Architekturelementarten Werkzeug und Material zur Verfügung.

Stilbasierte Ist-Architekturen werden soweit möglich direkt aus dem Quelltext berechnet. Dafür wird die *Quelltextstruktur* benötigt. Diese umfasst Folgendes:

- die Menge der Quelltextelemente Q,
- eine Relation $B_Q \subseteq Q \times Q$, die die bestehende Abhängigkeitsbeziehung zwischen diesen Quelltextelementen beschreibt.

Die stilbasierte Ist-Architektur und die Quelltextstruktur *hängen zusammen*, indem die Quelltextelemente den Architekturelementen in der folgenden Relation zugeordnet werden:

- die Relation $Z_Q \subseteq Q \times E$ legt für die Quelltextelemente fest, zu welchen Architekturelementen sie zugeordnet werden. Diese Relation ist rechtseindeutig.

Abbildung 6 zeigt das gesamte Metamodell für die Quelltextstruktur und die stilbasierte Ist-Architektur. Alle Informationen dieses Metamodells werden benötigt, um die stilbasierte Ist-Architektur eines Softwaresystems zu berechnen.

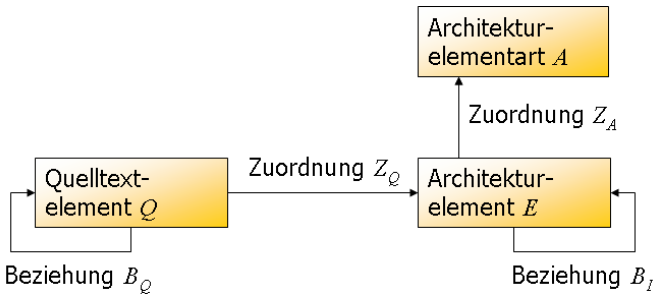


Abbildung 6: Quelltextstruktur und stilbasierte Ist-Architektur (Metamodell)

3.2 Ermittlung der stilbasierten Ist-Architektur

Die Quelltextstruktur (Q und B_Q) wird durch statische Quelltextanalyse berechnet, wie es unter anderem auch Konformanzprüfungs-Werkzeuge machen. Ein Quelltextelement kann dabei beispielsweise eine Klasse sein. Eine Abhängigkeit zu einer anderen Klasse ergibt sich beispielsweise durch Variablentypen oder eine Vererbungsbeziehung. Um den Ansatz einfach zu halten, unterscheiden wir bewusst nicht zwischen Abhängigkeitsarten. Dies ist auch in den meisten Architekturanalyse-Werkzeugen der Fall, da in der Regel sehr viel interessanter ist, von was ein Architekturelement abhängt, als welcher Art die einzelne Abhängigkeit ist. Unser Ansatz wäre jedoch leicht erweiterbar, sofern andere Fallstudien zeigen sollten, dass Abhängigkeitsarten benötigt werden.

Die Beziehungen der Ist-Architektur B_I lassen sich aus dem Quelltext berechnen. Dabei gilt: $B_I = \{(e, f) \mid \exists p, q : (p, q) \in B_Q \wedge (p, e) \in Z_Q \wedge (q, f) \in Z_Q\}$. Das heißt, wenn zwei Quelltextelemente p und q in Beziehung stehen, gibt es eine Beziehung in der Ist-Architektur zwischen den Architekturelementen e und f, sofern p und e sowie q und f einander zugeordnet sind.

Die Zuordnungen Z_Q und Z_A sowie die Menge der Architekturelemente E müssen zusätzlich beschrieben werden, sie sind dem reinen Quelltext nicht zu entnehmen.

4 Ein Konzept zur Quelltextannotation

Dieser Abschnitt behandelt die Frage, wie Z_Q, Z_A und E durch Quelltextannotation beschrieben werden können. Betrachten wir unser Beispiel in Abbildung 3. Es gilt für das Architekturelement namens Patientenaufnehmer:

- $(PatientenAufnahmeToolGui, Patientenaufnehmer) \in Z_Q$
- $(PatientenAufnahmeTool, Patientenaufnehmer) \in Z_Q$
- $(Patientenaufnehmer, Werkzeug) \in Z_A$

Damit die Annotationen mit dem Quelltext einfach konsistent gehalten werden können, sollten sie in dem jeweils betroffenen Quelltextelement festgehalten werden. Das heißt in unserem Beispiel: in den beiden Klassen `PatientenAufnahmeToolGui` und `PatientenAufnahmeTool` wird die Information benötigt "diese Klasse gehört zu dem Werkzeug namens Patientenaufnehmer". Quelltextelemente werden annotiert mit dem zugehörigen Architekturelement und dessen Elementart. Formal dargestellt heißt das:

- annotiert werden alle Quelltextelemente q, für die gilt: $\exists(q, e) \in Z_Q$.
- Die Annotation umfasst erstens das Architekturelement e, für das gilt: $(q, e) \in Z_Q$,
- und zweitens die Elementart a, für die gilt: $(e, a) \in Z_A$.

In einigen Fällen kann die Annotation verkürzt werden. Bei der Klasse `Patient` beispielsweise liegt so ein Sonderfall vor: hier besteht zwischen dem Architekturelement und dem Quelltextelement eine 1:1-Beziehung. Dieser Fall kann auftreten, wenn der gewählte Stil eine detaillierte Anleitung für die Architektur gibt und so viele feine Architekturelemente entstehen. Beispiele für solche Stile sind der WAM-Stil oder der Quasar-Stil. In diesem Fall können die Klasse und ihr Architekturelement gleich benannt werden. Dadurch fällt die benötigte Annotation kürzer aus. Es genügt, die Elementart zu nennen.

5 Beispielhafte Umsetzung mit Java-Annotations

Um den Ansatz anhand bestehender Softwaresysteme erproben zu können, wurde eine beispielhafte Syntax für die Quelltextannotationen definiert, die auf der im vorherigen Abschnitt definierten Semantik beruht. Wir haben uns entschieden, Java-Annotations zu verwenden. Dies ist eine in die Sprache Java integrierte Möglichkeit zur Quelltext-Annotation. Java-Annotations werden ähnlich wie Typen definiert und können dann an Quelltextabschnitte geschrieben werden. In unserem Beispiel werden die Klassen `PatientenAufnahmeToolGui` und `PatientenAufnahmeTool` annotiert mit: Werkzeug("Patientenaufnehmer"). Die Definition der Annotation für Werkzeuge ist in Abbildung 7 dargestellt.

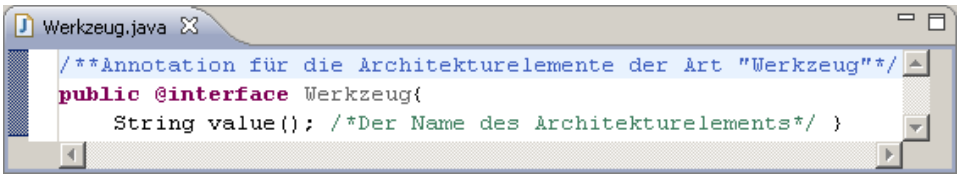


Abbildung 7: Definition der Java-Annotation für die Architekturelement-Art "Werkzeug"

Wir haben einen Prototyp namens ArchitectureExtractor realisiert, der anhand eines annotierten Quelltexts die stilbasierte Ist-Architektur ermittelt. Der Prototyp betrachtet alle

Java-Typen (Klassen, Interfaces etc.) als Quelltextelemente und ermittelt anhand des abstrakten Syntax-Baums die Referenzen zwischen diesen Elementen. Dafür nutzt er das entsprechende Eclipse-API.

Um die Machbarkeit des Ansatzes zu zeigen, haben wir den ArchitectureExtractor bisher für sechs Softwaresysteme verwendet, in der Größenordnung von 25.000 bis gut 100.000 LOC. Die Systeme folgten entweder dem WAM-Stil oder hatten einen speziell für das Projekt entworfenen, individuellen Stil. Die bereits existierenden Systeme wurden von einer projektexternen Person um Quelltextannotationen ergänzt. Die Person stützte sich dabei auf ihre guten Kenntnisse des Stils und auf Quelltextkommentare. Eine vollständige, externe Dokumentation der Architekturen existierte nicht, dazu waren die Architekturen zu feingranular, solch eine Dokumentation wäre zu schnell veraltet. Die Ergebnisse wurden mit WAM-Experten und Projektmitgliedern besprochen. Die befragten Personen bestätigten, dass die extrahierte Ist-Architektur aus ihrer Sicht hilfreich ist, da so ohne zusätzlichen Aufwand für externe Dokumentation ein Blick auf die Architektur ermöglicht wird, der sonst durch die Details des Quelltextes überdeckt wird. Die Quelltextannotationen verursachen nach unserer Einschätzung kaum zusätzlichen Aufwand, da vorher genau dieselben Informationen natürlichsprachlich dokumentiert werden mußten.

6 Verwandte Arbeiten

Architekturinformation im Quelltext unterzubringen, wird in der letzten Zeit vermehrt gefordert. Koschke und Popescu beispielsweise vergleichen verschiedene Ansätze zur Architektur-Konformanzprüfung [KP07]. Dabei kommen sie zu der Überzeugung, dass es einfacher wäre, den Quelltext und die Architekturinformationen konsistent zu halten, wenn man Architekturelemente im Quelltext kennzeichnen könnte. Clements und Shaw schreiben, dass manche Architekturinformationen im Quelltext nicht zu finden sind und nennen Schichtenarchitekturen als Beispiel [CS09]. Sie beklagen, dass bestehende Architekturanalyse-Werkzeuge nicht adäquat seien, da sie sich auf Vermutungen über Architekturkonstrukte stützen müssen. Sie sind der Meinung, dass Architektur-Markierungen im Quelltext helfen würden. Diese Forschungsrichtung betrachten sie als eine der drei vielversprechendsten für Softwarearchitekturen. Keiner der im Folgenden präsentierten Ansätze betrachtet *stilbasierte* Architekturen. Unseres Wissens sind wir die ersten, die im Quelltext Architekturinformationen unterbringen, bei denen der Zusammenhang zum gewählten Architekturstil explizit hergestellt wird.

Abi-Antoun und Aldrich extrahieren Objektgrafen durch statische Analyse aus speziell annotierten Quelltexten [AAA09]. Für die Annotationen nutzen sie Java-Generics. Direkt bei der Programmierung kann das Entwicklungsteam Architekturinformationen im Quelltext unterbringen. Die Autoren nennen als Vorteil, dass so der extrahierten Architektur die *Intention* des Entwicklungsteams zugrunde liegt und nicht eine beliebige Heuristik des genutzten Analyse-Werkzeugs. Genau das trifft auch für unseren Ansatz zu und ist der Hauptgrund, dass wir uns mit Quelltextannotationen beschäftigen. Im Gegensatz zu Abi-Antoun und Aldrich extrahieren wir jedoch keinen Objektgrafen, sondern betrachten die statische Sicht.

Einer der mittlerweile älteren und mehrfach zitierten Ansätze, Quelltexte mit Architekturinformationen anzureichern, behandelt ebenfalls die Laufzeitsicht [LR03]. Die Sprache Java wird um Subsysteme und Tokens erweitert. Eine Klasse kann einem Subsystem angehören, wenn ein Objekt erzeugt wird, so kann ihm ein Token gegeben werden. Lam und Rinard erzeugen mit Hilfe dieser Informationen Objektgrafen und ermitteln den Zusammenhang zwischen Subsystemen und Objekten. Sie heben hervor, dass ihre Modelle nicht manuell, sondern automatisch nur anhand des annotierten Quelltextes generiert werden und somit die Programmstruktur garantiert akkurat reflektieren.

Aldrich beschäftigt sich, wie wir, mit der statischen Sicht von Softwarearchitekturen [Ald08]. Er ergänzt die Sprache Java um sogenannte Komponenten-Klassen. Die Spracherweiterung ist eine Weiterentwicklung von ArchJava [ACN02]. Komponenten-Klassen können Ports definieren, über die sie sich verbinden lassen. Sie können hierarchisch geschachtelt werden. Ein mit solchen Komponentenklassen strukturierter Quelltext kann auf bestimmte Architektureigenschaften geprüft werden. Im Gegensatz zu unserem Ansatz mit Java-Annotations ist es nicht möglich, normale Java-Klassen als Architekturelemente zu kennzeichnen. Zwar nimmt Aldrich als Beispiel eine Pipes-and-Filters-Architektur, Stilinformationen können mit seiner Spracherweiterung jedoch nicht annotiert werden.

7 Zusammenfassung, Diskussion und Ausblick

Dieser Artikel präsentiert einen Ansatz, der es erlaubt, stilbasierte Ist-Architekturen aus annotierten Quelltexten zu extrahieren. Einige Informationen über Ist-Architekturen lassen sich aus dem reinen Quelltext extrahieren. Doch nicht alle Architekturkonstrukte sind hier wiederzufinden, einige verschwinden, sobald ein Architekturentwurf in Quelltext umgesetzt wird. Damit Werkzeuge nicht darauf angewiesen sind, die Architekturkonstrukte im Quelltext zu “erraten”, benötigen sie weitere Informationen, die wir durch Quelltextannotation hinzufügen. Neu an unserem Ansatz ist:

- Wir betrachten Architekturen, die auf Architekturstilen beruhen. Solche stilbasierten Architekturen spielen insbesondere für langlebige Softwaresysteme eine große Rolle.
- Wir nutzen die bestehenden Annotationsmöglichkeiten der Programmiersprache Java, um zusätzliche Informationen über die statische Architektur in den Quelltext zu bringen. Dadurch können bestehende Systeme von unserem Ansatz profitieren, ohne auf eine andere Sprache umsteigen zu müssen.

Der Ansatz, den Quelltext mit Architekturinformationen anzureichern, bietet verschiedene Vorteile:

- Entwicklungsteams können auf diese Weise ihre Intention bezüglich der Architektur direkt bei der Entwicklung festhalten.
- Es ist ein pragmatischer Ansatz. Dadurch, dass die Architekturinformationen direkt

im Quelltext festgehalten werden, wird es einfacher, die Konsistenz dieser Informationen zum Quelltext zu wahren.

- Entwicklungsumgebungen können erweitert werden, so dass dem Entwicklungsteam während der Programmierung eine Architektursicht auf den Quelltext gegeben wird.

Wir zeigen eine beispielhafte Umsetzung mit Java-Annotations. Wir haben einen Prototyp erstellt, den ArchitectureExtractor. Mit ihm wurden mehrere annotierte Quelltexte untersucht und erfolgreich die stilbasierten Ist-Architekturen extrahiert. Unser Prototyp beinhaltet bisher nur eine sehr einfache Darstellung der extrahierten Architektur, wir planen, einen entsprechenden ArchitectureViewer zu ergänzen und zu evaluieren, um eine gute Architektursicht direkt zum Programmierzeitpunkt zu bieten.

Unser Ansatz ist für statisch getypte Sprachen geeignet, da die Abhängigkeitsbeziehungen der Ist-Architektur durch statische Analyse aus dem Quelltext gewonnen werden. Wie andere Ansätze zur Quelltextannotation ist auch unser Ansatz darauf angewiesen, dass korrekt annotiert wird. Dass Entwickler tatsächlich ihre Intention in den Quelltext schreiben, lässt sich selbstverständlich nicht sicherstellen. Die im Abschnitt 6 vorgestellten verwandten Arbeiten diskutieren dieses Thema nicht. Wir vermuten jedoch, dass durch Werkzeugunterstützung fehlerhafte Annotationen reduziert werden könnten. Wird beispielsweise eine annotierte Klasse umbenannt oder durch Kopieren erzeugt, so könnte ein “Dirty-Flag” den Entwickler darauf hinweisen, dass die zugehörige Annotation möglicherweise überarbeitet werden muss. Hierfür planen wir, den ArchitectureExtractor in laufenden Projekten einzusetzen, um mögliche Fehlerquellen zu identifizieren und unterstützende Ansätze wie das Dirty-Flag zu evaluieren.

Die praktischen Untersuchungen haben gezeigt, dass nur einige Klassen als architekturrelevant eingestuft werden und annotiert werden müssen. Da die Annotationen die Intention der Entwickler beschreiben, liegt es bei ihnen, zu entscheiden, ob sie ein Quelltextelement annotieren, genauso, wie sie vorher entscheiden mußten, welche Kommentare sie schreiben. Fehlerhafte Annotationen können - genauso wie fehlerhafte Kommentare - nur durch zustätzliche Maßnahmen, wie beispielsweise Code-Reviews, aufgedeckt werden. Unser Ansatz bietet den Vorteil, dass die Architektur nicht manuell anhand der Quelltextkommentare verstanden werden muss, sondern automatisiert extrahiert werden kann.

Wir haben den hier präsentierten Ansatz bereits erweitert, um auch hierarchische Architekturen behandeln zu können. Wir planen, diesen erweiterten Ansatz weiter zu evaluieren und zu veröffentlichen.

Literatur

[AAA09] Marwan Abi-Antoun und Jonathan Aldrich. Static extraction of sound hierarchical runtime object graphs. In *TLDI ’09: Proceedings of the 4th international workshop on Types in language design and implementation*, Seiten 51–64, New York, NY, USA, 2009. ACM.

[ACN02] Jonathan Aldrich, Craig Chambers und David Notkin. ArchJava: Connecting Software Architecture to Implementation. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, Seiten 187–197, New York, NY, USA, 2002. ACM.

[Ald08] Jonathan Aldrich. Using Types to Enforce Architectural Structure. In *WICSA '08*, Seiten 211–220, Washington, DC, USA, 2008. IEEE Computer Society.

[BCK03] Len Bass, Paul Clements und Rick Kazman. *Software Architecture in Practice*. SEI Series in Software Engineering. Addison-Wesley, Reading, Mass., 2003.

[BKL04] Walter Bischofberger, Jan Kühl und Silvio Löffler. Sotograph - A Pragmatic Approach to Source Code Architecture Conformance Checking. In F. Oquendo, Hrsg., *EWSA 2004*, Seiten 1–9. Springer-Verlag Berlin Heidelberg, 2004.

[BP09] Petra Becker-Pechau. Stilbasierte Architekturprüfung. Angenommener Artikel auf der Informatik-Konferenz. 2009.

[CS09] Paul Clements und Mary Shaw. "The Golden Age of Software Architecture" Revisited. *IEEE Software*, 26(4):70–72, 2009.

[GAO94] David Garlan, Robert Allen und John Ockerbloom. Exploiting style in architectural design environments. In *SIGSOFT '94: Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*, Seiten 175–188, USA, 1994. ACM.

[GS94] David Garlan und Mary Shaw. An Introduction to Software Architecture. Bericht CS-94-166, Carnegie Mellon University, 1994.

[HNS00] Christine Hofmeister, Robert Nord und Dilip Soni. *Applied Software Architecture*. Object technology series. Addison-Wesley, 2000.

[KG06] Jung Soo Kim und David Garlan. Analyzing architectural styles with alloy. In *Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis*, Seiten 70–80. ACM Press, Portland, Maine, 2006.

[KP07] Jens Knodel und Daniel Popescu. A Comparison of Static Architecture Compliance Checking Approaches. In *Proceedings of the Sixth Working IEEE/IFIP Conference on Software Architecture (WICSA'07)*, Seite 12. IEEE Computer Society, 2007.

[Kru95] P. Kruchten. The 4+1 View Model of Architecture. *IEEE Software 12*, 6:42–50, 1995.

[Lil08] Carola Lilienthal. *Komplexität von Softwarearchitekturen - Stile und Strategien*. Dissertation, Universität Hamburg, 07 2008.

[LR03] Patrick Lam und Martin Rinard. A Type System and Analysis for the Automatic Extraction and Enforcement of Design Information. In *Proceedings of the 17th European Conference on Object-Oriented Programming*, Seiten 275–302, 2003.

[RH09] Ralf Reussner und Wilhelm Hasselbring, Hrsg. *Handbuch der Software-Architektur*, Jgg. 2. dpunkt.verlag, Heidelberg, 2009.

[Sie04] Johannes Siedersleben. *Moderne Softwarearchitektur: Umsichtig planen, robust bauen mit Quasar*. dpunkt.verlag, Heidelberg, 2004.

[SJSJ05] Neeraj Sangal, Ev Jordan, Vineet Sinha und Daniel Jackson. Using dependency models to manage complex software architecture. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, Seiten 167–176. ACM Press, San Diego, CA, USA, 2005.

[Zül05] Heinz Züllighoven. *Object-Oriented Construction Handbook*. Morgan Kaufmann Publishers, San Francisco, 2005.

Haben wir Programmverstehen schon ganz verstanden?

Rainer Koschke Rebecca Tiarks

Arbeitsgruppe Softwaretechnik
Fachbereich Mathematik und Informatik
Universität Bremen
Postfach 33 04 40
28334 Bremen
{koschke,beccs}@tzi.de

Abstract: Langlebige Systeme müssen kontinuierlich von Entwicklern angepasst werden, wenn sie nicht an Wert verlieren sollen. Ohne ausreichendes Verständnis des Änderungswunsches und des zu ändernden Gegenstands kann die Anpassung nicht effizient und effektiv vorgenommen werden. Deshalb ist es wichtig, das System so zu strukturieren, dass Entwickler es leicht verstehen können. Methoden und Werkzeuge müssen bereitgestellt werden, um die Aktivitäten bei der Änderung zu unterstützen. Dazu ist ein umfassendes Verständnis notwendig, wie überhaupt Entwickler Programme verstehen.

In diesem Beitrag geben wir eine Übersicht über den Stand der Wissenschaft zum Thema Programmverstehen und identifizieren weiteren Forschungsbedarf.

1 Einführung

Die Möglichkeit zur kontinuierlichen Wartung ist die Voraussetzung für langlebige Systeme, weil nur stete Anpassungen an geänderte Rahmenbedingungen Software nützlich erhält. Eine Reihe von Studien zeigt, dass die meisten Ressourcen in der Softwareentwicklung gerade in der Wartung und nicht für die initiale Erstellung von neuen Systemen verwendet werden [AN91, Art88, LS81, Mar83]. In der Wartung werden Fehler beseitigt, umfassende Restrukturierungen vorgenommen oder neue Funktionalität eingebaut.

Wartbarkeit ist keine absolute Eigenschaft eines Systems; sie hängt neben inneren Eigenschaften des Systems immer ab von der Art der Änderung und nicht zuletzt auch von den Entwicklern, die sie durchführen sollen. So können manche Entwickler eine Änderung schnell vornehmen, während andere dafür sehr viel mehr Zeit benötigen. Diese Unterschiede ergeben sich aus unterschiedlichen kognitiven Fähigkeiten, Vertrautheit mit dem System und Erfahrung in der Wartung, aber auch durch unterschiedliche Vorgehensweisen.

Wenn wir also bei langlebigen Systemen für nachhaltige Wartbarkeit sorgen wollen, müssen wir nicht nur zukünftige Änderungen beim initialen Entwurf antizipieren, sondern das System auch so strukturieren und dokumentieren, dass die zukünftigen Wartungsentwickler in der Lage sind, das System für den Zweck ihrer Änderung soweit zu verstehen, dass sie es anpassen und testen können. Dies erfordert ein grundlegendes Verständnis, wie Entwickler

Programme überhaupt verstehen. Wenn dieser Verstehensprozess ausreichend verstanden ist, können wir für verständnisfördernde Systemstrukturen sorgen und Wartungsentwicklern effektive und effiziente Methoden und Werkzeuge bereitstellen. Diese sollten sich nach dem jeweiligen Wartungsziel richten, um sowohl die korrektive als auch die adaptive Wartung optimal unterstützen zu können.

Die Forschung hat sich deshalb dem Thema Programmverstehen intensiv gewidmet. Jedoch flachte das Interesse nach einer Blütezeit in den Achtziger Jahren, in der vorwiegend Theorien über kognitive Strategien entwickelt wurden, wieder ab. Hin und wieder erscheinen auch in den letzten Jahren einzelne Publikationen zu diesem Thema. Dennoch können wir nicht belastbar behaupten, die Forschungsagenda, die im Jahre 1997 bei einem internationalen Workshop zu empirischen Studien in der Softwarewartung aufgestellt wurde, abgearbeitet zu haben [KS97]. Die dort speziell in Bezug auf Entwickler geäußerten offen Fragen lauten:

1. What tasks do people spend time on?
2. Time spent on comprehension?
3. Demographics of maintainers – age, technical experience, language experience, application experience?
4. What skills are needed?
5. What makes a great maintainer great?

Diese Fragen sind noch immer weitgehend offen, auch wenn einzelne Studien versucht haben, Teile von ihnen zu beantworten. Eine sehr bekannte und vielzitierte Studie, die die zweite offene Frage adressiert, stammt aus dem Jahr 1979 [FH79]. In dieser Studie kommen die Autoren zum Schluss, dass Wartungsentwickler rund die Hälfte (bei korrektiver Wartung sogar bis zu 60 %) ihrer Zeit mit dem Verständnis des Problems und des Systems verbringen. Schon auf dem bereits angesprochenen Workshop wurde auf fehlende Vergleichsstudien hingewiesen [KS97]. Umso mehr ist es heute unklar, ob diese Zahlen auf heutige Verhältnisse übertragbar sind. Außerdem hat diese Studie nur eine globale Aussage zum Programmverstehen gemacht (50 % des Aufwands in der Wartung), aber nicht weiter untersucht, wie lange die Programmierer dabei mit einzelnen Aktivitäten beschäftigt sind. Schließlich ist Zweifel an der Verlässlichkeit der Aussage zum Aufwand des Programmverstehens angebracht, da die Zahlen lediglich über Umfragen erhoben wurden, ohne tatsächlich selbst den Aufwand bei Wartungsaufgaben zu messen. Dennoch zitieren viele wissenschaftlichen Publikationen in Ermangelung von Alternativen diese Studie.

In diesem Beitrag geben wir eine Übersicht über den Stand der Wissenschaft zum Thema Programmverstehen und identifizieren weiteren Forschungsbedarf. Die folgende Darstellung zum Stand der Forschung im Programmverstehen gliedern wir in drei Themenfelder:

- **Untersuchung von kognitiven Aspekten:** In dieser Kategorie geht es um die Frage, wie ein Entwickler bzw. eine Entwicklerin vorgeht, wenn er oder sie ein Programm verstehen möchte.

- **Entwicklung von Hilfsmitteln:** Dieses Feld befasst sich mit der Frage, wie Werkzeuge beim Programmverstehen helfen können und wie diese aussehen sollten.
- **Bisherige empirische Studien:** Hier fassen wir frühere Studien zusammen, die empirisch neue Technologien und Werkzeuge sowie kognitive Aspekte untersuchten.

2 Untersuchung von kognitiven Aspekten

Die Erforschung der kognitiven Aspekte des Programmverstehens befasst sich mit menschlichen Denkprozessen von Programmierern bei der Analyse von Programmen. Sie ergründet, welche Strategien ein Programmierer beim Verstehen von Programmen verfolgt. Um diese Strategien zu untersuchen, muss man ermitteln, welche Informationen der Programmierer nutzt, um ein Stück Software und das zugrunde liegende Modell zu begreifen, und wie er diese verwendet. Im Verstehensprozess können zwei grundlegend verschiedene Strategien verfolgt werden. Der *Top-Down*-Ansatz geht davon aus, dass Hypothesen generiert werden und diese in Richtung auf niedrigere Abstraktionsebenen geprüft, verfeinert und gegebenenfalls revidiert werden. Beim *Bottom-Up*-Ansatz hingegen wird die Repräsentation eines Programms von unten nach oben aufgebaut, d.h. bei der Ableitung der Verfahrensweise eines Programmteils wird allein vom Quelltext ausgegangen, ohne dass vorher Annahmen über das Programm getroffen werden.

Ein Beispiel für den *Bottom-Up*-Ansatz ist das Modell von Soloway und Ehrlich [SE84]. Ihre Analyse basiert auf der Erkennung von Konzepten im Programm-Code. Diese Konzepte werden zusammengefasst zu Teilzielen und diese wiederum zu Zielen. Verschiedene Experimente bestätigen dieses Modell, und ein von Letowsky [Let88] entwickeltes Werkzeug führt Teile des Analyseprozesses automatisch aus. Auch Shneiderman und Mayer [SM79] beschreiben ein *Bottom-Up*-Modell. Sie unterscheiden zwei Arten von Wissen über ein Programm: syntaktisches und semantisches Wissen. Das syntaktische Wissen ist sprachabhängig und beschreibt die grundlegenden Einheiten und Anweisungen im Quelltext. Das semantische Wissen hingegen ist sprachunabhängig. Es wird schrittweise aufgebaut, bis ein mentales Modell entsteht, das den Anwendungsbereich beschreibt. Das *Bottom-Up*-Modell von Pennington [Pen87] besteht aus zwei Teilen: dem Situationsmodell und dem Programmmodell. Bei unbekanntem Code wird durch die Abstraktion des Kontrollflusses das Programmmodell gebildet. Dies geschieht in einem *Bottom-Up*-Prozess durch Zusammenfassen von kleinen Programmfragmenten (Anweisungen, Kontrollstrukturen) zu größeren Einheiten. Nach dem Programmmodell wird das Situationsmodell gebildet. Das Situationsmodell ergibt sich *bottom-up*, indem durch Rückverweise auf das Programmmodell eine Abstraktion des Datenflusses und eine funktionale Abstraktion entsteht.

Ein Verfechter des *Top-Down*-Modells ist Brooks [Bro83]. Die Grundlage seines Modells des Programmverstehens bilden drei Pfeiler:

- Der Programmierprozess ist eine Konstruktion von Abbildungen von einem Anwendungsbereich über ein oder mehrere Zwischenbereiche auf einen Programmierbereich.

- Der Verstehensprozess umfasst die Rekonstruktion von allen oder Teilen dieser Abbildungen.
- Der Rekonstruktionsprozess wird durch Erwartungen geleitet, d.h. durch das Aufstellen, Bestätigen und Verfeinern von Hypothesen.

Beim Verstehensprozess werden die Abbildungen, die bei der Entwicklung des Programms als Grundlage dienten, rekonstruiert. Die Menge von Abbildungen und ihre Abhängigkeiten untereinander sind baumähnlich aufgebaut. Hypothesen sind Annahmen über diese Abhängigkeiten. Die primäre Hypothese bildet die Wurzel. Sie beschreibt, was das Programm nach dem gegenwärtigen Verständnis des Programmierers tut. Das Verfeinern der primären Hypothese führt zu einer Kaskade von ergänzenden Hypothesen. Dieser Prozess wird solange fortgeführt, bis eine Hypothese gefunden ist, die präzise genug ist, um sie anhand des Codes und der Dokumentation zu verifizieren oder zu falsifizieren. Präzise genug heißt, dass die Hypothese Abläufe oder Datenstrukturen beschreibt, die mit sichtbaren Elementen im Quellcode oder der Dokumentation in Zusammenhang gebracht werden können. Diese sichtbaren Elemente werden mit dem Begriff *Beacons* beschrieben. Ein typischer *Beacon* für das Sortieren in einem Datenfeld wäre z.B. ein geschachteltes Paar von Schleifen, in denen zwei Elemente miteinander verglichen und vertauscht werden [Bro83].

Ein weiteres Modell, das aus einer Kombination der beiden vorherigen Modelle besteht, ist das *wissensbasierte Modell* von Letovsky [Let86]. Es geht davon aus, dass Programmierer sowohl *top-down* als auch *bottom-up* vorgehen. Es besteht aus drei Komponenten:

- Die Wissensbasis beinhaltet Fachwissen, Wissen über den Problembereich, Stilregeln, Pläne und Ziele.
- Das mentale Modell beschreibt das gegenwärtige Verständnis des Programmierers über das Programm.
- Der Prozess des Wissenserwerbs gleicht Quellcode und Dokumentation mit der Wissensbasis ab. Dadurch entwickelt sich das mentale Modell weiter. Der Prozess kann sowohl *top-down* als auch *bottom-up* erfolgen.

Nach Letovsky [Let86] gibt es drei Arten von Hypothesen:

- Warum-Vermutungen fragen nach dem Zweck eines Code-Fragments.
- Wie-Vermutungen fragen nach der Umsetzung eines Zwecks.
- Was-Vermutungen fragen nach, was etwas ist und was es tut (z.B. eine Funktion, die eine Variable setzt).

Das *integrierte Metamodell* von Mayrhauser und Vans[vMV95] ist ebenso eine Synthese der bereits erläuterten Modelle. Es besteht aus drei Komponenten, die Verstehensprozesse darstellen und verschiedene mentale Repräsentationen von Abstraktionsebenen enthalten. Die vierte Komponente bildet die Wissensbasis, die die Informationen für den Verstehensprozess bereit stellt und diese speichert.

In allen Modellen lassen sich gemeinsame Elemente wiederfinden. So verfügt ein Programmierer über Wissen aus zwei Bereichen: das allgemeine und das systemspezifische

Softwarewissen. Das allgemeine Wissen ist unabhängig von einem konkreten System, das verstanden werden soll, und umfasst Kenntnisse über Algorithmen, Programmiersprachen und allgemeine Programmierprinzipen. Das systemspezifische Wissen hingegen repräsentiert das gegenwärtige Verständnis des spezifischen Programms, das der Programmierer betrachtet. Während des Verstehensprozesses erwirbt der Programmierer weiteres systemspezifisches Wissen, jedoch kann ebenso zusätzliches allgemeines Wissen nötig sein [vMV95]. Der Prozess des Programmverstehens gleicht neu erworbenes Wissen mit Bestehendem ab und prüft, ob zwischen beiden eine Beziehung besteht, d.h. ob sich bekannte Strukturen im neuen Wissen befinden. Je mehr Korrelationen erkannt werden, desto größer ist das Verständnis, und es bildet sich ein mentales Modell. Das mentale Modell beim Programmverstehen ist eine interne Repräsentation der betrachteten Software und setzt sich aus statischen und dynamischen Elementen zusammen [vMV95].

Man unterscheidet beim Programmverstehen zwischen zwei Strategien: die *opportunistische* und die *systematische* Vorgehensweise. Bei einem systematischen Ansatz geht ein Programmierer in einer systematischen Reihenfolge vor, um ein Verständnis für das Programm als Ganzes zu erlangen, z.B. durch zeilenweises Verstehen des Codes [vMVL98]. Beim opportunistischen Ansatz geht der Programmierer nach Bedarf vor und konzentriert sich nur auf die Teile des Codes, von denen er denkt, dass sie für seine aktuelle Aufgabe von Bedeutung sein könnten, ohne die weiteren Abhängigkeiten zu anderen Programmteilen weiter zu betrachten. [LPLS86, KR91, LS86].

Die kognitive Psychologie befasst sich außerdem mit der Frage, welche weiteren Faktoren einen Einfluss darauf haben, wie gut ein Programm verstanden werden kann. Mögliche Faktoren sind z.B. die Programmeigenschaften, die individuellen Unterschiede von Programmierern und die Aufgabenabhängigkeit [SWM00]. Zu den Programmeigenschaften gehören unter anderem die Dokumentation, die Programmiersprache und das Programmierparadigma, das einer Sprache zugrunde liegt. Die Programmierer selber unterscheiden sich ebenso durch eine Reihe von individuellen Eigenschaften in Bezug auf ihre Fähigkeiten und Kreativität. Weiterhin können sich die Vorgehensweisen beim Programmverstehen je nach Aufgabe und Wartungsfragestellung unterscheiden. Programmverstehen stellt kein eigenständiges Ziel im Wartungsprozess dar, sondern ist vielmehr ein nötiger Schritt, um Änderungen an einem Programm durchzuführen, Fehler zu beheben oder Programmteile zu erweitern. Somit hängt auch der Verstehensprozess von der jeweiligen Aufgabe ab.

3 Entwicklung von Hilfsmitteln

Computergestützte Hilfsmittel können beim Programmverstehen von großem Nutzen sein, indem sie Programmierern helfen, Informationen herzuleiten, um daraus Wissen auf einem höheren Abstraktionsniveau zu schaffen.

Reverse-Engineering ist ein wichtiger Begriff im Zusammenhang mit der Unterstützung des Programmverstehens. Nach Chikofsky und Cross [CC90] ist Reverse-Engineering der Analyseprozess eines Systems, um die Systemkomponenten und deren Beziehungen zu identifizieren. Das Ziel des Reverse-Engineerings ist es, ein Softwaresystem zu verstehen, um das Korrigieren, Verbessern, Umstrukturieren oder Neuschreiben zu erleichtern

[Rug92]. Reverse-Engineering kann sowohl auf Quellcodeebene als auch auf Architektur- oder Entwurfsebene einsetzen.

Die maschinelle Unterstützung von Programmverstehen ist Gegenstand der aktiven Forschung. Die resultierenden Werkzeuge lassen sich in drei Kategorien einteilen [TS96]: Werkzeuge zur Extraktion, zur Analyse und zur Präsentation. Werkzeuge, die sich mit der Extraktion von Daten befassen, sind beispielsweise Parser. Analysetools erzeugen Aufrufgraphen, Modulhierarchien oder Ähnliches mit Hilfe von statischen und dynamischen Informationen. Bei der statischen Analyse werden die Informationen über ein Programm aus dem zugrunde liegenden Quelltext extrahiert. Im Gegensatz dazu wird bei der dynamischen Analyse das Programm mit Testdaten ausgeführt. Zu den Tools, die sich mit der Präsentation der Informationen befassen, gehören Browser, Editoren und Visualisierungswerkzeuge. Moderne Entwicklungsumgebungen und Reverse-Engineering-Werkzeuge vereinen oft Eigenschaften aus mehreren Kategorien in einem integrierten Werkzeug.

Die existierenden Analysen werden in *grundlegende* und *wissensbasierte* Analysen unterschieden [KP96]. Die grundlegenden Analysen ermitteln Informationen über das Programm und präsentieren diese in geeigneter Form. Der Betrachter nimmt mithilfe dieser Informationen die Abstraktion selber vor. Grundlegende statische Analysen erleichtern den Zugriff auf Informationen aus dem Quelltext. Sie verwenden ausschließlich Daten, die direkt aus dem Quelltext oder anderen Dokumenten abgeleitet werden können. Ein Beispiel hierfür sind Kontrollflussanalysen. Grundlegende dynamische Analysen hingegen ermitteln Informationen während des Programmlaufs.

Die wissensbasierten Analysen verfügen über Vorwissen. Dieses spezielle Wissen über Programmierung, Entwurf oder den Anwendungsbereich ermöglicht es den Analysen, automatisch zu abstrahieren. Nach Soloway und Ehrlich [SE84] verfügen erfahrene Programmierer über eine Basis an Wissen, die es ihnen ermöglicht, schneller als Unerfahrene zu abstrahieren. Anhand dieser Wissensbasis versuchen wissensbasierte Analysen, automatisch zu abstrahieren.

Die Forschung auf dem Gebiet der Softwarevisualisierung hat zu einer Reihe von Werkzeugen zur Unterstützung des Programmverstehens geführt. Bei der Bildung eines mentalen Modell spielt die Präsentation der extrahierten Information eine wichtige Rolle. Ein Modell zur Bewertung von Visualisierungen wurde von Pacione et al. [PRW04] vorgestellt. Mit Hilfe dieses Modells können Visualisierungen anhand ihrer Abstraktion, ihres Blickwinkels und ihren zugrundeliegenden Informationen evaluiert werden. Neuere Ansätze befassen sich außerdem mit der Integration von auditiver Unterstützung in moderne Entwicklungsumgebungen [HTBK09, SG09].

Es ergeben sich verschiedene Anforderungen an die Werkzeuge, die das Programmverstehen unterstützen sollen. Nach Lakhotia [Lak93] sollte ein Werkzeug die partielle Rekonstruktion des System-Designs ermöglichen. Singer et al. [SLVA97] identifizieren drei wichtige Funktionen, die ein Werkzeug bieten sollte. Zum einen sollte es dem Benutzer ermöglichen, nach Artefakten im Programm zu suchen, und zum anderen sollte das Werkzeug die Abhängigkeiten und Attribute der gefundenen Artefakte in geeigneter Form darstellen können. Um bereits gefundene Informationen zu einem späteren Zeitpunkt wie-

derverwenden zu können, sollte das Werkzeug außerdem die Suchhistorie speichern.

Einen weiteren wichtigen Aspekt formulieren Mayrhauser und Vans [vMV93]: ein Werkzeug sollte die verschiedenen Arbeitsweisen von Programmierern unterstützen, statt die Aufgaben im gesamten Wartungsprozess fest vorzugeben.

Leider ist festzuhalten, dass bisherige empirische Studien zur Eignung von Werkzeugen für das Programmverstehen oft nur mit einer geringen Anzahl von Teilnehmern und an kleineren Systemen durchgeführt wurden. Außerdem stehen bei den meisten Untersuchungen die Werkzeuge im Vordergrund und nicht die Verhaltensweisen und kognitiven Prozesse der Programmierer. Der steigende Bedarf an weitreichenderen Werkzeugen, die ein Benutzermodell entwickeln und den Entwickler somit besser proaktiv unterstützen können, erfordert jedoch mehr empirische Grundlagen auf diesem Gebiet.

4 Bisherige empirische Studien

Sobald der Nutzen einer Methode, einer Theorie oder eines Werkzeugs von menschlichen Faktoren abhängt, sind Studien und Experimente mit Menschen erforderlich. In Bezug auf das Programmverstehen sind empirische Studien sowohl für die Erforschung kognitiver Aspekte als auch für die Entwicklung von unterstützenden Anwendungen wichtig. Sie bieten die einzige Möglichkeit, Hypothesen systematisch zu bestätigen oder zu widerlegen, da mathematische Beweisführung für dieses Feld nicht in Frage kommt [Tic98, RR08].

Eine Reihe von Experimenten zum Programmverstehen beschreiben Mayrhauser und Vans [vMV95]. Die meisten dokumentierten Versuche beziehen sich allerdings auf kleine Programme mit einigen hundert Zeilen Code. Es stellt sich die Frage, ob die Ergebnisse auch für größere Programme geltend gemacht werden können. Außerdem fehlen Aussagen über die Anwendung von Fachkenntnissen, da den meisten Untersuchungen lediglich Wissen über die statische Struktur eines Programms zu Grunde liegt. Ein großer Teil der Erkenntnisse stammt zudem von Studien, die bereits über zehn Jahre zurück liegen, aber bisher nicht wieder bestätigt oder widerlegt worden sind; so z.B. auch die Ergebnisse von Fjeldstad und Hamlen [FH79]. Bei den neueren Studien stellt sich die Frage nach der externen Validität und der statistischen Signifikanz, da sie auf einer starken Vereinfachung des Umfelds beruhen. So wurden bei der Untersuchung von Ko et al. [KDV07] beispielsweise nur ein Entwickler in einem einzelnen Unternehmen beobachtet. Murphy et al. [MN97] begründen den Nutzen der Reflektionsmethode anhand einer Fallstudie, an der nur ein Programmierer beteiligt war. Außerdem sind viele der Studien nur auf ein Werkzeug oder eine Programmiersprache beschränkt (z.B. Eclipse: [SDVFM05, KAM05, LOZP06], Java: [DH09b], C++:[MW01].

Die Resultate aus der Studie von Soloway und Ehrlich [SE84] belegen, dass Programmierer Programme besser verstehen, wenn sie mit bestimmten Konzepten erstellt worden sind und sie bestimmten Programmierrichtlinien folgen, jedoch waren die verwendeten Programme kurze, künstlich erstellte Code-Fragmente, so dass sich wieder die Frage nach der Übertragbarkeit auf große Programme stellt. Littman et al. [LPLS86] haben Wartungsprogrammierer beobachtet und dabei festgestellt, dass diese entweder einer *opportunistischen* oder einer *systematischen* Vorgehensweise folgen. Die Programmierer, die systematisch

versucht haben, den Code zu verstehen, konnten ein mentales Modell des Programms aufbauen. Dadurch waren sie erfolgreicher bei der Umsetzung von Änderungen im Code. Ergebnisse zur Nützlichkeit von Werkzeugen stammen von Bennicke und Rust [BR04]. Die Experimente belegen, dass der Einsatz von Werkzeugen dazu führen kann, dass die Wartung effizienter und effektiver wird. Auch die Ergebnisse von Storey et al. [SWM00] zeigen, dass die drei von ihnen verglichenen Werkzeuge die bevorzugten Verstehensstrategien der Programmierer beim Lösen der gestellten Aufgabe unterstützen. Aktuellere Experimente haben sich mit der Frage befasst, inwieweit eine auditive Unterstützung, neben der visuellen, beim Programmverstehen hilfreich sein kann [SG09, HTBK09]. Sie belegen, dass der Einsatz von auditiven Informationen den Verstehensprozess positiv beeinflusst.

Eine Studie von Robillard et al. [RCM04] befasst sich mit den Kennzeichen effektiven Programmverstehens, also mit der Frage, inwieweit Strategien beim Programmverstehen Einfluss haben auf den Erfolg einer Änderungsaufgabe. Die Ergebnisse belegen zwar, dass systematisches Vorgehen beim Programmverstehen bessere Erfolgsquoten bei der Änderungsaufgabe erzielt als das zufällige Vorgehen; allerdings waren bei dem Experiment nur fünf Programmierer beteiligt. Weitere Studien untersuchen Behauptungen über den positiven Einfluss bestimmter neuer Techniken auf das Programmverstehen. Beispiele für diese Kategorie von Studien sind die Arbeiten von Prechelt et al. [PULPT02] über den Einfluss von Dokumentation von Entwurfsmustern, von Arisholm et al. [AHL06] über den Einsatz von UML sowie von Murphy et al. [MWB98] über den Einfluss von aspektorientierten Konzepten auf Aktivitäten in der Wartung. Ein weiterer Mehrwert dieser Studien über die Überprüfung einer konkreten Hypothese hinaus ist der Beitrag zur empirischen Methodik. Sie adaptieren und verfeinern allgemeine empirische Methoden für konkrete Untersuchungen im Bereich Programmverstehen. Arbeiten zur empirischen Methodik finden sich unter anderem bei Di Penta et al. [PSK07, LP06].

Mit den individuellen Unterschieden von Programmierern befasst sich eine Studie von Crosby et al. [CSW02]. Sie untersucht, welchen Einfluss der Wissensstand eines Programmierers auf das Erkennen der von Brooks [Bro83] identifizierten *Beacons* hat. Demnach erkennen erfahrenere Programmierer eher die sogenannten *Beacons* und finden dadurch schneller die für das Verständnis wichtigen Teile im Quelltext. Nach Höfer und Tichy [HT06] fehlen allerdings empirische Untersuchungenen zu den individuellen Eigenschaften von Programmierern, die neben der Softwaretechnik auch andere Disziplinen wie Sozialwissenschaften und Psychologie miteinbeziehen.

Eine Untersuchung von Ko [Ko03] belegt, dass die Erfahrung eines Programmierers Einfluss hat auf die Strategie, die er anwendet, wenn er mit einer unbekannten Programmierumgebung und Programmiersprache konfrontiert ist. Die Ergebnisse zeigen auch, dass für Wartungsarbeiten im Bereich der Fehlerbeseitigung das systemspezifische Wissen eher von Bedeutung ist als die Erfahrung eines Programmierers. Wie Programmierer vorgehen, wenn sie nach einer Unterbrechung eine Aufgabe erneut aufnehmen und somit sich somit an bereits erlangtes Wissen erinnern müssen, beschreibt eine Studie von Parnin und Rugaber [PR09].

Die Frage nach dem Einfluss von verschiedenen Codecharakteristika auf das Programmverstehen hat in den letzten Jahren zu einer Reihe von Untersuchungen geführt. Eine ältere Studie von Arab [Ara92] untersucht, inwieweit die Formatierung von Quelltext und Kom-

mentare das Programmverstehen unterstützen können. Weitere Studien über den Nutzen von Dokumentation stammen unter anderem von [BC05, DH09a, DH09b, SPL+88]. Alle Untersuchungen haben gezeigt, dass verschiedene Codecharateristika das Programmverstehen beeinflussen.

Inwieweit sich geschlechtsspezifische Unterschiede bei der räumlichen Wahrnehmung auf den Verstehensprozess übertragen lassen, untersuchen Fisher et al. [FCZ06] basierend auf der Hypothese von Cox et al. [CFO05], dass sich zwischen dem Programmverstehen und der Raumkognition Gemeinsamkeiten identifizieren lassen. Diese Studie ist eine der wenigen, die explizit die Unterschiede des Programmverstehens hinsichtlich des Geschlechtes betrachtet.

5 Fazit

Das Ziel, den Aufwand der Wartung und damit die Gesamtkosten der Softwareentwicklung zu reduzieren, hat den Anstoß zu einer Reihe von Untersuchungen zum Programmverstehen gegeben. Diesen Studien liegen jedoch hauptsächlich Experimente mit relativ kleinen Programmen zu Grunde. Die meisten Untersuchungen wurden zudem nur mit einer kleinen Anzahl von Teilnehmern durchgeführt, wodurch sich die Frage nach der Verallgemeinerbarkeit der Ergebnisse stellt. Es fehlen Aussagen zu größeren Systemen sowie eine klare Methodologie für das Programmverstehen, die methodisches Vorgehen nach der Wartungsaufgabe differenziert. Außerdem liegen viele der Studien bereits Jahrzehnte zurück, und die heutige Gültigkeit der Ergebnisse angesichts moderner Programmiersprachen und Entwicklungsumgebungen steht in Frage. Die Trends der heutigen Softwareprojekte, wie z.B. das stärkere Einbinden von Open-Source-Komponenten, die zunehmende Verteilung der Entwickler oder die kürzeren Entwicklungszyklen, haben auch zu neuen Erkenntnisse über die Entwicklungsmethoden, die Eigenschaften der entwickelten Programme und das Verhalten der Entwickler geführt. Das hinterfragt zusätzlich die früheren Erkenntnisse über das Programmverstehen und erhöht den Bedarf nach aktuellen, detaillierten und signifikanten Untersuchungen.

Auf Basis unserer Literaturstudie zum Thema Programmverstehen formulieren wir die folgenden offenen Fragen, die zukünftige Forschung adressieren sollte:

- **Vorgehensweisen:** Wie gehen Programmierer – abhängig von der Art ihrer Aufgabe – genau vor, wenn sie Programme ändern sollen? Aus welchen Einzelaktivitäten besteht der Programmverstehensprozess für welche Art von Aufgaben? Welche Informationen werden abhängig von der Aufgabe benötigt? Warum gehen Programmierer so vor? Worin unterscheiden sich die Vorgehensweisen unterschiedlicher Programmierer? Welche Vorgehensweisen führen eher zum Erfolg?
- **Aufwand:** Wie hoch ist der Aufwand des Programmverstehens in der Wartungsphase im Verhältnis zu Änderung und Test? Wie hoch ist der Aufwand der Einzelaktivitäten des Programmverstehens? Wie hoch sind die Kosten, wenn bestimmte Informationen fehlen?

- **Werkzeuge:** Wie werden moderne Entwicklungswerkzeuge beim Programmverstehen benutzt? Wie gut eignen sie sich für ihren Zweck? Wie lassen sie sich verbessern? Für welche Aspekte existiert noch unzureichende Werkzeugunterstützung? Welches Wissen kann kaum von Werkzeugen verwaltet werden?
- **Methoden:** Was sind gut geeignete Vorgehensweisen beim Programmverstehen für wiederkehrende Aufgabenarten? Welche Informationen und Werkzeuge sind geeignet, um diese Vorgehensweisen zu unterstützen?
- **Code-Strukturen:** Welche Auswirkungen haben die *Bad Smells* von Beck und Fowler auf das Programmverständnis?
- **Einfluss von Architektur** Wie wird Architektur in der Praxis dokumentiert? Wie wird sie von Programmierern benutzt? Welche Probleme gibt es für den Austausch des Architekturwissens?
- **Wissenschaftliche Methoden:** Welche Methoden aus den Sozial- und Kognitionswissenschaften und der Psychologie lassen sich für die Forschung im Programmverstehen wie übertragen? Wie können empirische Beobachtungen und Experimente durch Softwaresysteme unterstützt werden?

Literatur

[AHL06] L.C. Arisholm, E.and Briand, S.E. Hove und Y. Labiche. The impact of UML documentation on software maintenance: an experimental evaluation. *IEEE Computer Society TSE*, 32(6):365–381, Juni 2006.

[AN91] A. Abran und H. Nguyenkim. Analysis of maintenance work categories through measurement. In *Software Maintenance, 1991., Proceedings. Conference on*, Seiten 104–113, Oct 1991.

[Ara92] Mouloud Arab. Enhancing program comprehension: formatting and documenting. *SIGPLAN Not.*, 27(2):37–46, 1992.

[Art88] Lowell Jay Arthur. *Software Evolution: A Software Maintenance Challenge*. John Wiley & Sons, 2 1988.

[BC05] Scott Blinman und Andy Cockburn. Program comprehension: investigating the effects of naming style and documentation. In *Proc. of AUIC'05*, Seiten 73–78, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc.

[BR04] M. Bennicke und H. Rust. Programmverstehen und statische Analysetechniken im Kontext des Reverse Engineering und der Qualitätssicherung, April 2004. Virtuelles Software Engineering Kompetenzzentrum.

[Bro83] Ruven E. Brooks. Towards a Theory of the Comprehension of Computer Programs. *IJMMS*, 18(6):543–554, 1983.

[CC90] Elliot J. Chikofsky und James H. Cross. Reverse Engineering and Design Recovery: a Taxonomy. *IEEE Software*, 7(1):13–17, Januar 1990.

[CFO05] Anthony Cox, Maryanne Fisher und Philip O'Brien. Theoretical Considerations on Navigating Codespace with Spatial Cognition. In *Proc. of PPIG'05*, Seiten 92–105, 2005.

[CSW02] Martha E. Crosby, Jean Scholtz und Susan Wiedenbeck. The Roles Beacons Play in Comprehension for Novice and Expert Programmers. In *Proc. PPIG'02*, Seiten 18–21, 2002.

[DH09a] Uri Dekel und James Herbsleb. Improving API Documentation Usability with Knowledge Pushing. In *Proc. of ICSE'09*, 2009.

[DH09b] Uri Dekel und James D. Herbsleb. Reading the Documentation of Invoked API Functions in Program Comprehension. In *Proc. of ICPC'09*, Seiten 168–177, Vancouver, Canada, 2009. IEEE Computer Society.

[FCZ06] Maryanne Fisher, Anthony COx und Lin Zhao. Using Sex Differences to Link Spatial Cognition and Program Comprehension. In *Proc. ICSM'06*, Seiten 289–298, Washington, DC, USA, 2006. IEEE Computer Society.

[FH79] R.K. Fjeldstad und W.T. Hamlen. Application Program Maintenance Study: Report to our Respondents. In *Proc. of GUIDE 48*, Philadelphia, PA, 1979.

[HT06] Andreas Höfer und Walter F. Tichy. Status of Empirical Research in Software Engineering. In *Empirical Software Engineering Issues*, Seiten 10–19, 2006.

[HTBK09] Khaled Hussein, Eli Tilevich, Ivica Ico Bukvic und SooBeen Kim. Sonification DEsign Guidelines to Enhance Program Comprehension. In *Proc. ICPC'09*, Seiten 120–129, Vancouver, Canada, 2009. IEEE Computer Society.

[KAM05] Andrew J. Ko, Htet Htet Aung und Brad A. Myers. Eliciting Design Requirements for Maintenance-Oriented IDEs: A Detailed Study of Corrective and Perfective Maintenance Tasks. In *Proc. ICSE'05*, Seiten 126–135, 2005.

[KDV07] Andrew J. Ko, Robert DeLine und Gina Venolia. Information Needs in Collocated Software Development Teams. *Proc. ICSE'07*, Seiten 344–353, 2007.

[Ko03] Andrew Jensen Ko. Individual Differences in Program Comprehension Strategies in an Unfamiliar Programming System. In *IWPC'03*, 2003.

[KP96] R. Koschke und E. Plödereder. *Ansätze des Programmverstehens*, Seiten 159–176. Deutscher Universitätsverlag/Gabler Vieweg Westdeutscher Verlag, 1996. Editor: Franz Lehner.

[KR91] Jürgen Koenemann und Scott P. Robertson. Expert problem solving strategies for program comprehension. In *Proc. of SIGCHI'91*, Seiten 125–130, New York, NY, USA, 1991. ACM.

[KS97] Chris F. Kemerer und Sandra Slaughter. Methodologies for Performing Empirical Studies: Report from WESS'97. *EMSE*, 2(2):109–118, Juni 1997.

[Lak93] Arun Lakhotia. Understanding someone else's code: Analysis of experiences. *JSS*, 23(3):269–275, 1993.

[Let86] Stanley Letovsky. Cognitive Processes in Program Comprehension. In *Workshop ESP*, Seiten 58–79, Norwood, NJ, USA, 1986. Ablex Publishing Corp.

[Let88] Stanley Letovsky. Plan analysis of programs. Bericht Research Report 662, Yale University, Dezember 1988.

[LOZP06] Andrea De Lucia, Rocco Oliveto, Francesco Zurolo und Massimiliano Di Penta. Improving Comprehensibility of Source Code via Traceability Information: a Controlled Experiment. *Proc. of ICPC'06*, 0:317–326, 2006.

[LP06] Giuseppe A. Di Lucca und Massimiliano Di Penta. Experimental Settings in Program Comprehension: Challenges and Open Issues. In *Proc. of ICPC'06*, Seiten 229–234, Washington, DC, USA, 2006. IEEE Computer Society.

[LPLS86] D. C. Littman, Je. Pinto, S. Letovsky und E. Soloway. Mental Models and Software Maintenance. In *Workshop ESP*, Seiten 80–98, Norwood, NJ, USA, 1986. Ablex Publishing Corp. Reprinted in Journal Systems and Software 7, 4 (Dec. 1987), 341–355.

[LS81] Bennet P. Lientz und E. Burton Swanson. Problems in application software maintenance. *Commun. ACM*, 24(11):763–769, 1981.

[LS86] S. Letovsky und E. Soloway. Delocalized Plans and Program Comprehension. *Software, IEEE*, 3(3):41–49, May 1986.

[Mar83] James Martin. *Software Maintenance the Problem and Its Solution*. Prentice Hall, third printing. Auflage, 6 1983.

[MN97] Gail C. Murphy und David Notkin. Reengineering with Reflexion Models: A Case Study. *IEEE Computer*, 30(8):29–36, August 1997. Reprinted in Nikkei Computer, 19, January 1998, p. 161-169.

[MW01] Russel Mosemann und Susan Weidenbeck. Navigation and Comprehension of Programs by Novice Programmers. In *Proc. of IWPC'01*, Seite 79, Washington, DC, USA, 2001. IEEE Computer Society.

[MWB98] G. Murphy, R. Walker und E. Baniassad. Evaluating Emerging Software Development Technologies: Lessons Learned from Assessing Aspect-oriented Programming. Technical Report TR-98-10, University of Britsh Columbia, Computer Science, 1998.

[Pen87] N. Pennington. Comprehension Strategies in Programming. In *Workshop ESP*, Seiten 100–113, Norwood, NJ, USA, 1987. Ablex Publishing Corp.

[PR09] Chris Parnin und Spencer Rugaber. Resumption Strategies for Interrupted Programming Tasks. *Proc. of ICPC'09*, 0:80–89, 2009.

[PRW04] Michael J. Pacione, Marc Roper und Murray Wood. A Novel Software Visualisation Model to Support Software Comprehension. In *Proc. of WCRE'04*, Seiten 70–79, Washington, DC, USA, 2004. IEEE Computer Society.

[PSK07] Massimiliano Di Penta, R. E. Kurt Stirewalt und Eileen Kraemer. Designing your Next Empirical Study on Program Comprehension. In *ICPC*, Seiten 281–285. IEEE Computer Society, 2007.

[PULPT02] L. Prechelt, B. Unger-Lamprecht, M. Philippsen und W.F. Tichy. Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance. *IEEE Computer Society TSE*, 28(6):595–606, Juni 2002.

[RCM04] M. Robillard, W. Coelho und G. Murphy. How Effective Developers Investigate Source Code: An Exploratory Study. *IEEE Computer Society TSE*, 30(12):889–903, 2004.

[RR08] R. L. Rosnow und R. Rosenthal. *Beginning behavioral research: A conceptual primer*. Pearson/Prentice Hall, 6th. Auflage, 2008.

[Rug92] Spencer Rugaber. Program Comprehension for Reverse Engineering. In *AAAI Workshop*, Juli 1992.

[SDVFM05] J. Sillito, K. De Voider, B. Fisher und G. Murphy. Managing software change tasks: an exploratory study. In *ESE'05*, Seiten 10 pp.–, Nov. 2005.

[SE84] Elliot Soloway und Kate Ehrlich. Empirical Studies of Programming Knowledge. *IEEE TSE*, 10(5):595–609, 1984.

[SG09] Andreas Stefik und Ed Gellenbeck. Using Spoken Text to Aid Debugging: An Emperical Study. *Proc. of ICPC'09*, 0:110–119, 2009.

[SLVA97] Janice Singer, Timothy Lethbridge, Norman Vinson und Nicolas Anquetil. An examination of software engineering work practices. In *Proc. of CASCON'97*, Seite 21. IBM Press, 1997.

[SM79] B. Shneiderman und R. Mayer. Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results. *IJCIS*, 8(3):219–238, 1979.

[SPL+88] E. Soloway, J. Pinto, S. Letovsky, D. Littman und R. Lampert. Designing Documentation to Compensate for Delocalized Plans. *Commun. ACM*, 31(11):1259–1267, November 1988.

[SWM00] M.-A. D. Storey, K. Wong und H. A. Müller. How do program understanding tools affect how programmers understand programs? *Science of Computer Programming*, 36(2–3):183–207, 2000.

[Tic98] Walter F. Tichy. Should computer scientists experiment more? *IEEE Computer*, 31(5):32–40, Mai 1998.

[TS96] Scott R. Tilley und Dennis B. Smith. Coming Attractions in Program Understanding. Bericht, Software Engineering Institute, Pittsburgh, PA 15213, Dezember 1996.

[vMV93] Anneliese von Mayrhauser und A. Marie Vans. From Program Comprehension to Tool Requirements for an Industrial Environment. In *IWPC*, Seiten 78–86. IEEE Computer Society Press, Juli 1993.

[vMV95] Anneliese von Mayrhauser und A. Marie Vans. Program Comprehension During Software Maintenance and Evolution. *IEEE Computer*, 28(8):44–55, 1995.

[vMVL98] Anneliese von Mayrhauser, A. Marie Vans und Steve Lang. Program Comprehension and Enhancement of Software. In *Proc. of the IFIP World Computing Congress*. IEEE, August-September 1998.

Anwendungslandschaften und ihre Verwendung durch exemplarische Geschäftsprozessmodellierung verstehen

Stefan Hofer

Universität Hamburg, Departement Informatik, Arbeitsbereich Softwaretechnik
und
C1 WPS GmbH
Vogt-Kölln-Str. 30
22527 Hamburg
stefan.hofer@c1-wps.de

Abstract: Projekte zur Umgestaltung komplexer Anwendungslandschaften erweisen sich in der Praxis als Herausforderung für Unternehmen. Die laufende Anpassung an sich ändernde fachliche und technische Gegebenheiten ist für die Zukunftssicherheit einer Anwendungslandschaft unverzichtbar. Dieser Artikel schlägt einen Modellierungsansatz vor, der Anwendungslandschaften zum Gegenstand eines von allen Beteiligten gestalteten Veränderungsprozesses macht. Durch die ineinandergreifende Modellierung von technischer Architektur und Geschäftsprozessen wird Verständnis für jene Zusammenhänge geschaffen, welche die Umgestaltung so herausfordernd machen.

1 Einleitung

Unternehmen fordern heutzutage integrierte IT-Systemen, die anwendungsübergreifende Geschäftsprozesse unterstützen. Dadurch rücken *Anwendungslandschaften* in den Mittelpunkt der Unternehmens-IT. Um eine zukunftssichere Anwendungslandschaft konzipieren und betreiben zu können, benötigt man nicht nur langlebige IT-Systeme. Zusätzlich müssen sich Unternehmen mit dem stetigen Anpassungsdruck auseinandersetzen, dem die Landschaften ausgesetzt sind. Sich ändernde technische und fachliche Gegebenheiten erfordern die häufige Umgestaltung von Anwendungslandschaften. Die Notwendigkeit ihrer strategischen Planung und die Beschäftigung mit den dafür nötigen Modellen hat sich in Wissenschaft und Praxis unter den Begriffen *IT-Unternehmensarchitektur* [Kel07] und *Software-Kartographie* [MW04] etabliert. Die Durchführung entsprechender Umgestaltungsprojekte bleibt jedoch eine große Herausforderung für Unternehmen, wie folgendes Beispiel verdeutlicht:

Eine Bank wechselt aus technischen Gründen das Betriebssystem aller Client-Server-Anwendungen und muss deswegen ihre Anwendungslandschaft anpassen. Eigenentwicklungen werden portiert, Standardanwendungen mit aktuellen Versionen ersetzt und einige Anwendungen wie das System für den Wertpapierhandel eingestellt, weil ihre Funktio-

nalität als externe Dienstleistung zugekauft wird. Für die Umsetzung dieser Änderungen muss die Anwendungslandschaft *im Kontext ihrer Verwendung* betrachtet und verstanden werden, da sonst viele relevante Fragen unbeantwortet bleiben. Bezogen auf die Veränderungen im Wertpapierhandel sind dies beispielsweise:

- Welcher Ausschnitt der Anwendungslandschaft ist von den Veränderungen im Wertpapierhandel betroffen?
- Kommt es durch die neue Anwendung zu Redundanzen in der Datenhaltung, so dass Daten aus mehreren Beständen synchronisiert werden müssen?
- Wie wirken die komplexen IT-gestützten und manuellen Arbeitsprozesse zusammen, die den Geschäftsprozess „Wertpapierhandel“ ausmachen?
- Welche dieser Arbeitsprozesse müssen angestoßen werden, um die neuen und geänderten Schnittstellen in der Anwendungslandschaft testen zu können?

In diesem Artikel wird untersucht, wie die gemeinsame Betrachtung von Anwendungslandschaften und ihrer Verwendung Umgestaltungsprojekte unterstützen kann (siehe Abschnitt 2). Als Grundlage für diese Analyse dienen Projekte der C1 WPS GmbH, in welcher der Autor als Berater tätig ist. Mit der an der Universität Hamburg entwickelten *exemplarischen Geschäftsprozessmodellierung* (*eGPM*, siehe Abschnitt 3) steht ein praxiserprobter Ansatz zur Verfügung, um Anwendungslandschaften und ihre Verwendung ineinandergreifend zu modellieren. In mehreren Projekten der C1 WPS wurden positive Erfahrungen mit eGPM gemacht. In einem Abgleich mit bestehenden Modellierungsansätzen für Anwendungslandschaften und Geschäftsprozesse (siehe Abschnitt 4) zeigt der Artikel, dass die vorgeschlagene Sichtweise auf Anwendungslandschaften bisher wenig beachtete Themen beleuchtet. Abschließend fasst Abschnitt 5 die bisherigen Erkenntnisse zusammen und zeigt weiteren Forschungsbedarf auf.

2 Umgestaltung von Anwendungslandschaften

2.1 Begriffe

Unternehmen betreiben nicht länger einzelne, isolierte Softwaresysteme, sondern eine große Zahl teilweise von einander abhängiger Anwendungen. Die Gesamtheit dieser Anwendungen wird zusammen mit ihren Abhängigkeiten als *Anwendungslandschaft* eines Unternehmens bezeichnet [Der03]. Änderungen im Geschäft und technische Gründe wie schlecht zu wartende IT-Systeme erfordern die laufende Anpassung von Anwendungslandschaften. Um sie umgestalten zu können, ist die Kenntnis ihrer Architektur genauso wichtig wie die Abstimmung mit den von ihr zu unterstützenden Arbeits- und Geschäftsprozessen. Dieser Artikel folgt der Definition der ANSI und versteht unter Architektur den Aufbau einer Anwendungslandschaft, verkörpert durch ihre Bestandteile und deren Beziehungen untereinander, sowie die Grundsätze ihres Entwurfs und ihrer Evolution [ANS00].

Als Geschäftsprozesse fasst man koordinierte Aktivitäten zur Erreichung von Geschäftszielen zusammen [Wes07]. Umgesetzt werden diese abstrakten Aktivitäten durch konkrete manuelle und IT-gestützte Arbeitsprozesse.

Ohne ein Verständnis der Architektur können die Auswirkungen der Anpassung einer Anwendungslandschaft nicht eingeschätzt werden, während eine fehlende Abstimmung mit den Prozessen den Sinn der Änderungen in Frage stellt. Zu einer ähnlichen Einschätzung kommen Conrad et al., die für das Gebiet der *Enterprise Application Integration* (*EAI*) die beiden Problembereiche „technische Realisierung“ und „organisatorische/soziale Integration“ identifizieren [CHKT06, S. 5]. EAI kann als Teilbereich der Umgestaltung von Anwendungslandschaften aufgefasst werden, weil das Einführen neuer Anwendungen, die Anpassung von Schnittstellen zwischen Anwendungen und das Abschalten von Anwendungen die wesentlichen Möglichkeiten darstellen, um eine Anwendungslandschaft zu verändern. Die in diesem Artikel vorgeschlagene Unterstützung für die Umgestaltung von Anwendungslandschaften basiert daher auf der Unterstützung dieser Veränderungsmöglichkeiten.

2.2 Herausforderungen in der Praxis

Dieser Abschnitt fasst die Erfahrungen zusammen, welche die C1 WPS GmbH in Umgestaltungsprojekten gesammelt hat. Neben Beratung zur IT-Strategie der Kunden und der Modellierung von Ist- und Soll-Anwendungslandschaften unterstützte sie die Umsetzung der erarbeiteten Pläne. Dabei traten immer wieder zwei grundlegende Fragen auf, deren Beantwortung die gemeinsame Betrachtung der Architektur der Anwendungslandschaften und der von ihnen unterstützten Prozesse erforderte:

- Was muss man über die Anwendungslandschaft und ihre Verwendung wissen, um sie umgestalten zu können?
- Welche Auswirkung hat die Umgestaltung auf die Anwendungslandschaft und ihre Verwendung?

Anhand eines fiktiven, aber realistischen Beispiels aus der Versicherungsbranche werden im Folgenden drei Kontexte vorgestellt, in denen diese Fragen auftauchen:

> Eine Versicherung verwendet Vorgangsmappen, um die arbeitsteilige Abarbeitung von Schadensfällen zu koordinieren und den Bearbeitungsstatus eines Falls nachvollziehen zu können. Die einzelnen Arbeitsschritte der Sachbearbeiter werden bereits durch IT-Systeme unterstützt. Hingegen ist die Vorgangsmappe eine nicht formal festgelegte, lose Kombinaton von E-Mails, Dateien und Papierdokumenten. Um die Bearbeitungsgeschwindigkeit zu erhöhen, sollen die Vorgangsmappen durch ein Workflow-Management-System (WfMS, vgl. [zM04]) umgesetzt werden.

Fachliche und technische Integration

Nur mit Detailwissen über die IT-Unterstützung der betroffenen Akteure können Schnittstellen zwischen Anwendungen auf technischer *und* fachlicher Ebene ermittelt und eine sinnvolle Integration erreicht werden. Im Beispiel:

Um herauszufinden, wie das WfMS und die bestehenden Anwendungen sinnvoll integriert werden können, müssen die Arbeitsprozesse der Sachbearbeiter und die vorhandene IT-Unterstützung modelliert werden.

- Welche Arbeitsgegenstände (eingescannte Dokumente, Anträge auf Papier etc.) verwenden die Sachbearbeiter?
- Welche Arbeitsschritte erledigen sie IT-gestützt und mit welchen Anwendungen? Tauschen diese untereinander Daten aus? Wie wird die Kommunikation angestoßen?
- Welche Anwendung ist führend bei der Verwaltung welcher Daten?

Migrationsplanung

Unternehmen setzen große Änderungen in ihren Anwendungslandschaften oft in kleinen Schritten um, sodass die Landschaft und die von ihr unterstützten Prozesse auf dem Weg vom Ausgangs- zum Zielzustand mehrere produktive Zwischenzustände durchlaufen. In jedem dieser Schritte müssen die Veränderungen in der Landschaft gut abgestimmt und von Fachabteilung und IT verstanden werden, um den reibungslosen Betrieb aufrecht erhalten zu können. Im Beispiel:

- In welcher Reihenfolge sollen die Arbeitsprozesse der Sachbearbeiter im WfMS abgebildet werden?
- Wie verändern sich die Prozesse in der Abteilung für Schadensfälle und die Schnittstellen in der Anwendungslandschaft durch das WfMS? Wie sehen sie nach dem ersten, zweiten, dritten etc. Migrationsschritt aus?

Testszenarios

Anwendungsübergreifende Tests entlang fachlicher Prozesse mindern das Risiko, das neue und geänderte Schnittstellen für die Verwendung einer Anwendungslandschaft darstellen. Passende Testszenarien zu identifizieren und konkrete Testfälle aufzustellen erfordert die detaillierte Kenntnis der Zusammenhänge zwischen technischer Architektur der Anwendungslandschaft und den von ihr unterstützten Arbeitsprozessen. Im Beispiel:

- Welche anwendungsübergreifenden Arbeitsprozesse führt ein Sachbearbeiter aus?
- Gibt es manuelle Schritte in diesen Arbeitsprozessen, die Kommunikation zwischen Anwendungen auslösen?

- Welche Abhängigkeiten zwischen Anwendungen sind rein fachlich und nicht technisch (z.B. Nachbearbeitung von eingescannten Dokumenten)?

3 Exemplarische Geschäftsprozessmodellierung – eGPM

Dieser Abschnitt stellt einen Ansatz zur Geschäftsprozessmodellierung vor, der sich in der Praxis als Hilfsmittel zur Beantwortung der oben genannten Fragestellungen erwiesen hat. Da eine ausführliche Beschreibung des Ansatzes den Umfang dieses Artikels sprengen würde, wird er nur im Überblick dargestellt.

3.1 Kurzvorstellung

Ursprung der exemplarischen Geschäftsprozessmodellierung (vgl. [BKS06, Bre03]) ist die Analyse der Unterstützung kooperativer Arbeit durch Software – speziell der Aspekte *Kommunikation*, *Kooperation* und *Koordination* (vgl. [TSMB95]). Die Modellierung findet in diesem Ansatz typischerweise in Workshops statt, in denen z.B. Vertreter aus Fachabteilung und IT ihr Wissen und ihre unterschiedlichen Sichten in gemeinsame Modelle einbringen. Das ermöglicht unmittelbare Rückkopplung, ob die Darstellung des modellierten Gegenstandsbereichs zutreffend ist. Da eine Werkzeugunterstützung für diese Vorgehensweise sehr nützlich ist, wurde eGPM in Zusammenarbeit mit der BOC Group[1] als Modellierungsmethode für das Geschäftsprozessmanagement-Werkzeug ADONIS implementiert.

eGPM besteht aus mehreren Modelltypen, die in ein gemeinsames Metamodell eingebettet sind. Durch die Werkzeugunterstützung können Modelle verschiedener Typen auf Ebene von Modellelementen verknüpft werden, was eine konsistente Modellierung erleichtert. Zusätzlich stehen Überblicks- und Begriffsmodelltypen zu Verfügung, die einen Überblick über zusammengehörige Modelle verschaffen bzw. Begriffe und Konzepte des modellierten Gegenstandsbereichs zueinander in Beziehung setzen. Nur die beiden für diesen Artikel wichtigsten Modelltypen werden im Folgenden näher beschrieben.

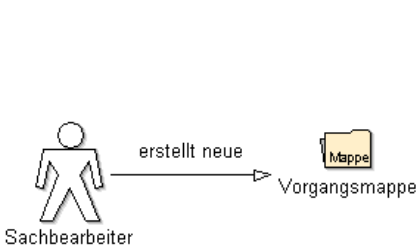
Kooperationsbilder

Das *Kooperationsbild* ist der zentrale Modelltyp der eGPM und baut auf den Arbeiten von Krabbel, Wetzel et al. (siehe u.a. [KWR96a, KWR96b]) auf. Kooperationsbilder stellen Kommunikation, Kooperation und Koordination zwischen Akteuren graphisch dar. Sowohl Menschen als auch IT-Systeme können als Akteure eine aktive Rolle übernehmen. Dadurch machen Kooperationsbilder die IT-Unterstützung von Geschäftsprozessen auf der Ebene einzelner Arbeitsschritte sichtbar. Akteure sind durch typisierte Pfeile mit anderen Akteuren und Gegenständen (z.B. „Geschäftsobjekte"wie in [BELM08]) verbunden. Diese

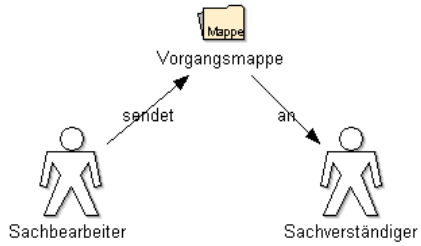
[1] www.boc-group.com

Relationen beschreiben folgende Aktivitäten:

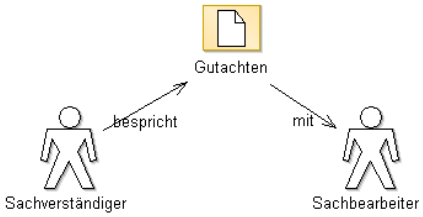
- Akteure bearbeiten Gegenstände (siehe Abbildung 1a)
- Akteure geben Gegenstände untereinander weiter (siehe Abbildung 1b)
- Akteure geben Informationen über ein Medium (E-Mail, Telefon usw.) weiter (siehe Abbildung 1c)



(a) Akteur bearbeitet Gegenstand



(b) Akteur gibt Gegenstand weiter



(c) Akteur informiert mit Medium

Abbildung 1: Relationen zwischen Akteuren und Gegenständen im Kooperationsbild

Den Kooperationsbildern liegt keine ablaufsteuernde, algorithmische Denkweise zu Grunde. Stattdessen zeigen sie einen konkreten Ablauf eines Prozesses aus der Sicht der beteiligten Akteure. Die einzelnen Aktivitäten des Prozesses werden dazu in eine exemplarische Reihenfolge gebracht. Auf die Darstellung von Fallunterscheidungen wird verzichtet – Kooperationsbilder „erzählen“ Szenarios (vgl. [Car00]).

Wie sich im Praxiseinsatz gezeigt hat, ist eGPM durch die grafische Darstellung der Kooperationsbilder und den Verzicht auf Fallunterscheidungen so verständlich, dass die an den Workshops teilnehmenden Personen auch selbst Modelle erstellen können, während ein geschulter Modellierer die Rolle des Moderators annimmt. Dieser achtet unter anderem darauf, dass das gewählte Szenario durch klare Randbedingungen von anderen Fällen abgegrenzt wird (z.B. Schadensfall Privathaftpflichtversicherung ohne Zusatzdeckungen, Schadenssumme unter 10.000€, Schaden wird vollständig gedeckt). Abbildung 2 stellt den fiktiven Geschäftsprozess „Bearbeitung eines Schadensfalls“ mit den beschriebenen Rahmenbedingungen als Kooperationsbild dar. Dieses Beispiel ist im Vergleich zu realen Prozessen stark vereinfacht und verwendet nicht alle graphischen Elemente des Modelltyps. Anhand der Nummern kann der Ablauf des Szenarios nachvollzogen werden.

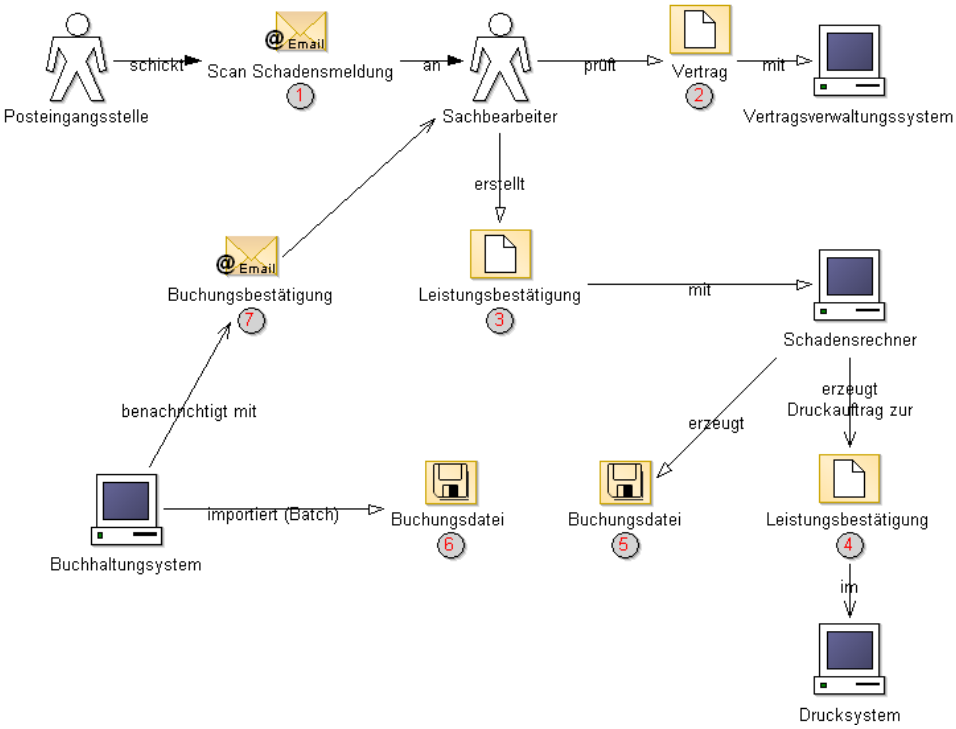


Abbildung 2: Bearbeitung eines Schadensfalls als Kooperationsbild

IT-Landschaft

Der Modelltyp *IT-Landschaft*[2] stellt die Zusammenhänge mehrerer IT-gestützter Prozesse dar. Es handelt sich dabei um ein Begriffsmodell, das IT-Systeme zueinander in Beziehung setzt. Ein geschulter Modellierer erstellt auf Basis der aus den Kooperationsbildern gewonnenen Informationen über die IT-Unterstützung ein Modell des relevanten Teils der Anwendungslandschaft. Dieses Modell wird gegebenenfalls in Workshops mit der IT-Abteilung um technische Aspekte ergänzt und von Widersprüchen bereinigt. Kooperationsbilder können auf IT-Landschaften Bezug nehmen, in dem die Anwendungen mit denen der IT-Landschaft verknüpft werden. Das Modell der IT-Landschaft dient dem mit der Auswahl der Szenarien verbundenen Zweck – etwa der Herausarbeitung von technischen, fachlichen und organisatorischen Berührungspunkten einer Anwendungslandschaft zu einem neu einzuführenden Workflow-Management-System. Abbildung 3 zeigt den für die Schadensbearbeitung relevanten Teil der IT-Landschaft vor der Einführung des WfMS. Ein solches Modell führt die technikbezogenen Informationen mehrerer Kooperationsbilder (u.a. dem in Abbildung 2) zusammen. Auch diese Abbildung ist im Vergleich zu

[2] „IT-Landschaft“ ist in eGPM ein Oberbegriff für „Anwendungslandschaft“, der zusätzlich Hardware- und Software-Infrastruktur mit einschließt.

echten IT-Landschaften stark vereinfacht und enthält nur einige wenige der möglichen Darstellungsmittel. Zu sehen sind ausgewählte IT-Systeme, für welche Daten sie führend zuständig sind und welche Daten von ihnen aus anderen Anwendungen importiert (grauer Pfeil), an andere Anwendungen gesendet (schwarzer Pfeil) und mit anderen Anwendungen synchronisiert (schwarzer Doppelpfeil) werden.

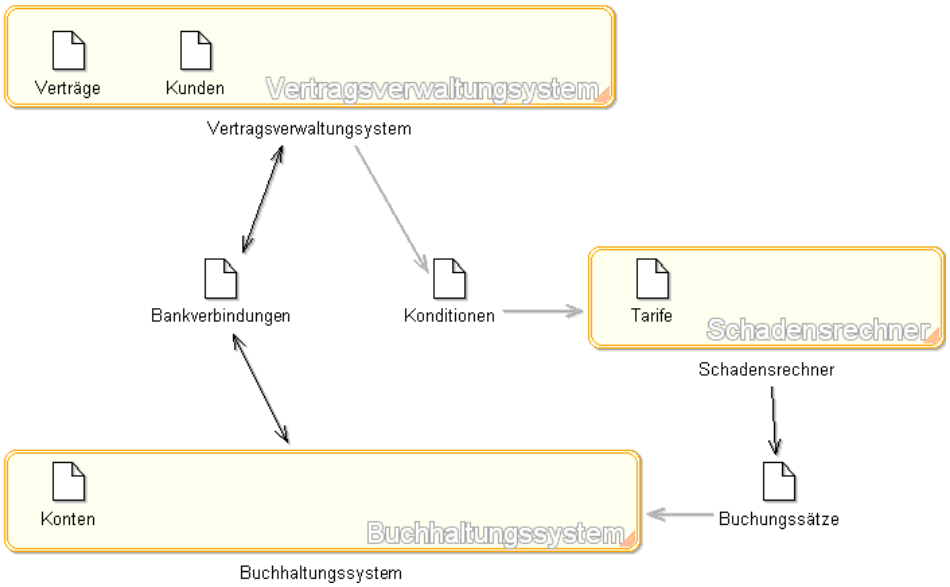


Abbildung 3: Beispiel einer fiktiven IT-Landschaft

3.2 Unterstützung der Umgestaltung durch eGPM

Dieser Abschnitt beschreibt unsere Erfahrungen beim Einsatz der eGPM in Umgestaltungsprojekten. Er zeigt, mit welchen Eigenschaften eGPM den in Abschnitt 2.2 aufgelisteten Herausforderungen begegnet.

Gemeinsames Verständnis schaffen

Die Umgestaltung einer Anwendungslandschaft involviert viele Personen mit unterschiedlichen Fähigkeiten und Hintergründen, beispielsweise Mitarbeiter aus Fachbereichen und IT sowie externe Dienstleister. Diese Personen haben direkt mit der Umgestaltung zu tun, verfügen aber über verschiedene Sichten auf die Anwendungslandschaft. Selbst wenn bereits Informationen zu den einzelnen Anwendungen vorliegen und in ein Modell verdichtet wurden, heißt das nicht, dass die Beteiligten über ein *gemeinsames* Verständnis verfügen.

Dieses ist aber wegen der nötigen Abstimmung in solchen Projekten (siehe z.B. die Frage nach der Planung produktiver Zwischenschritte im Abschnitt 2.2) von großer Bedeutung.

Die eGPM leistet mit ihren für alle Beteiligte verständlichen Darstellungsformen einen Beitrag zur Schaffung eines gemeinsamen Verständnisses einer Anwendungslandschaft. Beim Erstellen der Modelle erarbeiten sich die Beteiligten außerdem ein gemeinsames Begriffsmodell. Daher kann der Modellierungsprozess der eGPM als „Verständnisprozess" aufgefasst werden, in dessen Verlauf die Beteiligten explizit Informationen einbringen und unterschiedliche Sichten verbinden.

Einfordern von Fragen zur Anwendungslandschaft

Viele der Fragen aus Abschnitt 2.2 beschäftigen sich mit Kommunikation, Koordination und Kooperation. Im Kontext von Anwendungslandschaften spielen diese Aspekte sowohl in der Zusammenarbeit von IT-Systemen untereinander als auch zwischen Anwendungen und ihren Benutzern eine Rolle. Daher greifen rein technisch motivierte Modelle zu kurz.

Die Kooperationsbilder der eGPM stellen unter anderem dar, *wer - was - womit - wozu* bearbeitet und *wer - wen - mit welchen Mitteln* informiert. Dadurch fordert sie von den Modellierern, explizit Fragen zur IT-Unterstützung von Kommunikation, Koordination und Kooperation zu stellen. Die Antworten zu diesen Fragen gehen in das Modell der Anwendungslandschaft ein.

Umgang mit Unvollständigkeit

In der Praxis kann kaum eine vollständige Modellierung einer Anwendungsanlandschaft erreicht werden: Zum einen wären die zu erhebenden Daten extrem umfangreich und nicht aktuell zu halten. Zum andern verfügen viele Unternehmen nicht über die Möglichkeit, die Vollständigkeit der Modellierung z.B. anhand der vom Lizenzmanagement erfassten Anwendungen zu überprüfen.

Eines der Kernkonzepte der eGPM ist die Fokussierung auf Szenarios. Diese auszuwählen und dabei Wesentliches von Unwesentlichem zu trennen bedeutet, die Unvollständigkeit von Modellen zu thematisieren und einen konstruktiven Umgang damit zu suchen. Die Szenarios werden anhand des *konkreten Ziels* (vgl. „Pragmatisches Merkmal" in [Sta73]) eines Umgestaltungsprojekts (z.B. Einführung eines Workflow-Management-Systems) gewählt. Die bisherigen Erfahrungen beim Einsatz der eGPM zeigen, dass dadurch genügend Informationen für die Modellierung der betroffenen Teile der Anwendungslandschaft erhoben werden können.

4 Abgleich mit bestehenden Ansätzen

Dieser Abschnitt stellt andere Ansätze vor, die sich mit der Modellierung von Anwendungslandschaften und ihrer Einordnung in einen unternehmensweiten Kontext befassen.

4.1 Unternehmensarchitektur

Als Unternehmensarchitektur werden unternehmensweite, integrierte Modelle von Geschäftsstrategie, Geschäftsprozessen, Anwendungen und Infrastruktur verstanden [Kel07]. Es handelt sich also um die Beschreibung der existierenden Strukturen von Unternehmen. Das Modell einer Anwendungslandschaft ist Teil einer Unternehmensarchitektur.

Unternehmensarchitekturen sind nicht als Hilfsmittel für die detaillierte Planung und Durchführung von Änderungen in Anwendungslandschaften gedacht. So wird in [ARW08] und [WF06] gefordert, Unternehmensarchitekturen auf aggregierten Informationen aufzubauen und sie strategisch auszurichten. D.h. Unternehmensarchitekturen bestehen aus grobgranularen Informationen über alle Aspekte eines Unternehmens hinweg statt vertiefender Detailinformationen zu den einzelnen Aspekten. Winter und Fischer empfehlen, spezialisierte Architekturansätze für die operativen Aufgaben in Teilbereichen des Unternehmens einzusetzen [WF06]. Die eGPM ist ein solcher spezialisierter Ansatz.

4.2 Softwarekartographie

Dieser Ansatz stellt Anwendungslandschaften in Form von *Softwarekarten* [LMW05] dar. In [BELM08] werden die Darstellungskonzepte (beispielsweise verschiedene Kartentypen) und die damit verbundenen Einsatzmöglichkeiten in Form von Mustern zusammengefasst. Ziel dieses Musterkatalogs ist, bestehende Ansätze zur Beschreibung von Unternehmensarchitekturen (siehe Abschnitt 4.1) zu ergänzen. Dazu wurden in der Praxis häufig auftretende Fragestellungen gesammelt und passende Lösungsansätze beschrieben, die aus Handlungsanweisungen, dafür benötigten Informationen und geeigneten Darstellungsformen für diese Informationen bestehen.

Der Großteil dieser Fragestellungen (*Concerns*) betrifft strategische Themen wie die langfristige Planung von Anwendungslandschaften. Zwar gibt es zwischen den Fragen und den Einsatzzwecken der eGPM einige Überschneidungen (etwa „Concern C-51: Which business objects are used or exchanged by which business applications or services?“ [BELM08, S. 34]), doch unterscheiden sich die Modelle für deren Beantwortung. Sofwarekarten basieren auf der Visualisierung von im Vorfeld (z.B. per Fragebogen) erhobenen Daten und sind nicht das Resultat eines Modellierungsprozesses im Sinne der eGPM. Die eGPM bindet die betroffenen Akteure in den Prozess mit ein und macht ihre Aufgaben zum Bestandteil von Kooperationsbildern, wodurch die Zusammenhänge zwischen Anwendungslandschaft und Geschäftsprozessen auf Ebene von Arbeitsprozessen (siehe Abschnitt 2.1) dargestellt werden. Softwarekarten zeigen diese Zusammenhänge auf Ebene von Geschäftsobjekten (siehe z.B. *Viewpoint* 48 im Katalog) oder direkt auf Geschäftsprozessebene (siehe z.B. Viewpoint 17). Ein weiterer Unterschied besteht im Umgang mit der (Un-)Vollständigkeit von Modellen (siehe Abschnitt 3.2).

4.3 Geschäftsprozessmodellierung

Im Gebiet der Geschäftsprozessmodellierung dominieren Ansätze, die Prozesse als *Kontrollflüsse* [KNS92] beschreiben. Als Vertreter dieses Konzepts führt Weske unter anderem die *Business Process Modeling Notation* und *ereignisgesteuerte Prozessketten* auf (vgl. [Wes07]). „Anwendungen“ spielen in diesen Ansätzen eine untergeordnete Rolle. Beispielsweise bietet der verbreitete Ansatz der *ereignisgesteuerten Prozessketten* [KNS92] lediglich die Möglichkeit, *Informationsobjekte* im Sinne von Datenquellen oder Speicherorten in einem Geschäftsprozessmodell zu verwenden.

5 Zusammenfassung und Ausblick

Anwendungslandschaften rücken immer mehr in den Blickwinkel von Unternehmen und stehen aus fachlichen und technischen Gründen unter ständigem Änderungsdruck. Zukunftssicher können Anwendungslandschaften daher nur sein, wenn sie immer wieder durch Umgestaltungsprojekte an neue Gegebenheiten angepasst werden. In solchen Projekten werden Modelle von Anwendungslandschaften für die Planung und Umsetzung verwendet, jedoch stellt dabei die Verknüpfung von technischer Architektur und Arbeitsprozessen eine von der Informatik noch wenig beachtete Herausforderung dar. Eine solche Sicht auf Anwendungslandschaften ist aber für viele Aufgaben in Umgestaltungsprojekten hilfreich, etwas für die Konzeption fachlich und technisch passender Schnittstellen, für die detaillierte Migrationsplanung und für die Erarbeitung von Testszenarios.

Mit der exemplarischen Geschäftsprozessmodellierung steht ein vielversprechender und praxiserprobter Ansatz zur Verfügung, um Anwendungslandschaften und die von ihnen unterstützten Prozesse ineinandergreifend zu modellieren. Er bildet die Grundlage für das Promotionsvorhaben des Autors, in dessen Verlauf die eGPM weiter für den Einsatz in Umgestaltungsprojekten optimiert werden soll. Dazu werden die Darstellungsmittel der eGPM weiter verbessert und eine passende Modellierungsmethode ausgearbeitet. Ergänzend werden Muster für die Anwendung der Methode bereitgestellt.

Ein weiterer, noch zu erforschender Aspekt ist die Einordnung der eGPM in eine Unternehmensarchitektur. Die Verbindung des Metamodells der eGPM mit dem Metamodell eines Ansatzes für die Modellierung von Unternehmensarchitekturen könnte eine Brücke zwischen strategisch ausgerichteten, stark aggregierten Darstellungen von Anwendungslandschaften und operativ ausgerichteten, detaillierten Darstellungen schlagen.

Literatur

[ANS00] ANSI/IEEE Computer Society. ANSI/IEEE-Standard-1471. IEEE Recommended Practice for Architectural Description of Software-Intensive Systems, 2000.

[ARW08] Stephan Aier, Christian Riege und Robert Winter. Unternehmensarchitektur. Literaturüberblick und Stand der Praxis. *Wirtschaftsinformatik*, 4:292–304, 2008.

[BELM08] Sabine Buckl, Alexander M. Ernst, Josef Lankes und Florian Matthes. Enterprise Architecture Management Pattern Catalog. Technical Report TB 0801, Technische Universität München, Lehrstuhl Software Engineering betrieblicher Informationssysteme, Februar 2008.

[BKS06] Holger Breitling, Andreas Kornstädt und Joachim Sauer. Design Rationale in Exemplary Business Process Modeling. In Alen H. Dutoit, Ray McCall, Ivan Mistrik und Barbara Paech, Hrsg., *Rationale Management in Software Engineering*, Seiten 191–208. Springer-Verlag, 2006.

[Bre03] Holger Breitling. Integrierte Modellierung von Geschäftsprozessen und Anwendungssoftware. Treffen der Fachgruppe 2.1.6. Requirements Engineering der GI, November 2003.

[Car00] John Carroll. *Making Use. Scenario-Based Design of Human-Computer Interaction*. MIT Press, 2000.

[CHKT06] Stefan Conrad, Wilhelm Hasselbring, Arne Koschel und Roland Tritsch. *Enterprise Application Integration*. Elsevier, 2006.

[Der03] Gernot Dern. *Management von IT-Architekturen*. Vieweg, 2003.

[Kel07] Wolfgang Keller. *IT-Unternehmensarchitektur. Von der Geschäftsstrategie zu optimalen IT-Unterstützung*. dpunkt.verlag, Heidelberg, 2007.

[KNS92] G. Keller, M. Nüttgens und A.-W. Scheer. Semantische Prozeßmodellierung auf der Grundlage Ereignisgesteuerter Prozeßketten (EPK). *Forschungsberichte des Instituts für Wirtschaftsinformatik,*, Heft 89, 1992.

[KWR96a] Anita Krabbel, Ingrid Wetzel und Sabine Ratuski. Objektorientierte Analysetechniken für übergreifende Aufgaben. In Jürgen Ebert, Hrsg., *GI-Fachtagung Softwaretechnik '96*, Seiten 65–72, September 1996.

[KWR96b] Anita Krabbel, Ingrid Wetzel und Sabine Ratuski. Participation of Heterogeneous User Groups: Providing an Integrated Hospital Information System. *Proceedings of the Participatory Design Conference*, PDC 96:241–249, 1996.

[LMW05] Josef Lankes, Florian Matthes und André Wittenburg. Softwarekartographie: Systematische Darstellung von Anwendungslandschaften. In Otto K. Ferstl, Elmar J. Sinz, Sven Eckert und Tilman Isselhorst, Hrsg., *Wirtschaftsinformatik Proceedings 2005*, Seiten 1443–1462. Physica-Verlag, 2005.

[MW04] Florian Matthes und André Wittenburg. Softwarekarten zur Visualisierung von Anwendungslandschaften und ihren Aspekten - Eine Bestandsaufnahme. Technical report, Technische Universität München, sebis, 2004.

[Sta73] Herbert Stachowiak. *Allgemeine Modelltheorie*. Springer, 1973.

[TSMB95] Stephanie Teufel, Christian Sauter, Thomas Mühlherr und Kurt Bauknecht. *Computerunterstützung für die Gruppenarbeit*. Addison-Wesley, Bonn, 1995.

[Wes07] Mathias Weske. *Business Process Management*. Springer, Berlin, 2007.

[WF06] Robert Winter und Ronny Fischer. Essential Layers, Artifacts, and Dependencies of Enterprise Architecture. In *EDOC Workshop on Trends in Enterprise Architecture Research*, Seiten 30–38. IEEE Computer Society, 2006.

[zM04] Michael zur Muehlen. *Workflow-based Process Controlling*. Logos Verlag, 2004.

Supporting Evolution by Models, Components, and Patterns

Isabelle Côté and Maritta Heisel
Universität Duisburg-Essen
{Isabelle.Cote, Maritta.Heisel}@uni-duisburg-essen.de

Abstract: Evolvability is of crucial importance for long-lived software, because no software can persist over a long time period without being changed. We identify three key techniques that contribute to writing long-lived software in the first place: models, patterns, and components. We then show how to perform evolution of software that has been developed according to a model-, pattern-, and component-based process. In this situation, evolution means to transform the models constructed during development or previous evolutions in a systematic way. This can be achieved by using specific operators or rules, by replacing used patterns by other patterns, and by replacing components.

1 Introduction

As pointed out by Parnas [Par94], software ages for two reasons: first, because it is changed. These changes affect the structure of the software, and, at a certain point, further changes become infeasible. The second reason for software aging is *not* to change the software. Such software becomes outdated soon, because it does not reflect new developments and technologies.

We can conclude that software evolution is indispensable for obtaining long-lived software. However, the evolution must be performed in such a way that it does not destroy the structure of the software. In this way, the aging process of the software can be slowed down.

An evolution process that does not destroy the structure of the software first of all requires software that indeed possesses some explicit structure that can be preserved. Hence, different artifacts (not only the source code) should be available. In conclusion, to avoid the legacy problems of tomorrow [EGG+09], we first need appropriate development processes that provide a good basis for future evolutions. Second, we need systematic evolution approaches that can make use of that basis.

In this paper, we first point out that the use of models, patterns, and components are suitable to provide the basis that is necessary for an evolution that does not contribute to software aging. Second, we briefly describe a specific process (called ADIT: Analysis, Design, Implementation, Testing) that makes use of these techniques. Third, we sketch how evolution can be performed for software that was developed using ADIT. There, replay of development steps as well as model transformations play a crucial role.

The rest of the paper is organized as follows: Section 2 discusses the role of models, patterns, and components for the construction of long-lived software. Section 3 introduces the development process ADIT. How the ADIT artifacts can be used to support evolution is shown in Sect. 4. Related work is discussed in Sect. 5, and we conclude in Sect. 6.

2 Models, Patterns, and Components

Models, patterns, and components are all relatively recent techniques that have turned out to be beneficial for software development. Models provide suitable abstractions, patterns support the re-use of development knowledge, and components allow one to assemble software from pre-fabricated parts. These three techniques contribute to the maturing of the discipline of software technology by introducing engineering principles that have counterparts in other engineering disciplines.

2.1 Models

The idea of model-based software development is to construct a sequence of models that are of an increasing level of detail and cover different aspects of the software development problem and its solution. The advantage of this procedure is that it supports a separation of concerns. Each model covers a certain aspect of the software to be developed, and ignores others. Hence, to obtain specific information about the software, it suffices to inspect only a subset of the available documentation.

Using models contributes to developing long-lived software, because the models constitute a detailed documentation of the software that is well suited to support evolution. Of course, the models and the code must be kept consistent. The process we describe in Sect. 4 guarantees that this is the case, because it first adjusts the models, before the code is changed.

Today, the Unified Modeling Language [For06] is commonly used to express the models set up during a model-based development process. For UML, extensive tool support is available. The ADIT process described in Sect. 3 mostly makes use of UML notations.

2.2 Patterns

Patterns are abstractions of software artifacts (or models). They abstract from the application-specific parts of the artifact and only retain its essentials. Patterns are used by instantiation, i.e., providing concrete values for the variable parts of the pattern.

Patterns exist not only as design patterns [GHJV95] (used for fine-grained software design), but for every phase of software development, including requirements analysis [Jac01, Fow97], architectural design [SG96], implementation [Cop92], and testing [Bin00].

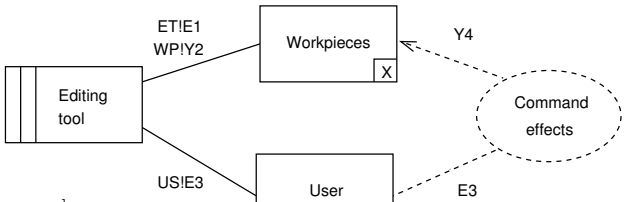
Since patterns can be regarded as templates for software development models, model- and pattern-based software development approaches fit very well together. Patterns further enhance the long-livedness of software: first, since the purpose of the different patterns is known, the use of patterns supports program comprehension. Second, patterns enhance the structure of software, e.g., by decoupling different components. Thus, evolution tasks can be performed without tampering too much with the software's structure.

Problem Frames [Jac01] are patterns that can be used in requirements analysis. Since they are less known than the other kinds of patterns just mentioned, we briefly describe them in the following. The ADIT process makes use of problem frames.

2.2.1 Problem Frames

Problem frames are a means to describe software development problems. They were invented by Jackson [Jac01], who describes them as follows: *"A problem frame is a kind of pattern. It defines an intuitively identifiable problem class in terms of its context and the*

characteristics of its domains, interfaces and requirement." Problem frames are described by *frame diagrams*, which consist of rectangles, a dashed oval, and links between these (see frame diagram in Fig. 1). All elements of a problem frame diagram act as placeholders which must be instantiated by concrete problems. Doing so, one obtains a problem description that belongs to a specific problem class.




```
E3:  user commands
E1:  editing operations
Y2:  workpieces status
Y4:  workpieces properties
```

Figure 1: *Simple workpieces* problem frame

Plain rectangles denote *problem domains* (that already exist in the application environment), a rectangle with a double vertical stripe denotes the *machine* (i.e., the software) that shall be developed, and *requirements* are denoted with a dashed oval. The connecting lines between domains represent interfaces that consist of *shared phenomena*. Shared phenomena may be events, operation calls, messages, and the like. They are observable by at least two domains, but controlled by only one domain, as indicated by an exclamation mark. For example, in Fig. 1 the notation `US!E3` means that the phenomena in the set `E3` are controlled by the domain `User`. A dashed line represents a requirements reference. It means that the domain is mentioned in the requirements description. An arrow at the end of such a dashed line indicates that the requirements constrain the problem domain. In Fig. 1, the `Workpieces` domain is constrained, because the `Editing tool` has the role to change it on behalf of user commands for achieving the required `Command effects`.

Each domain in a frame diagram has certain characteristics. In Fig. 1 the `X` indicates that the corresponding domain is a *lexical* domain. A lexical domain is used for data representations.A `B` indicates that a domain is *biddable*. A biddable domain usually represents people [Jac01].

Problem frames support developers in analyzing problems to be solved. They show what domains have to be considered, and what knowledge must be described and reasoned about when analyzing the problem in depth. Thus, the problem frame approach is much more than a mere notation.[1] Other problem frames besides simple workpieces frame are *required behaviour*, *commanded behaviour*, *information display*, and *transformation*.

After having analyzed the problem, the task of the developer is to construct a *machine* based on the problem described via the problem frame that improves the behavior of the environment it is integrated in, according to its respective requirements.

2.3 Components

Component-based development [CD01, SGM02] tries to pick up principles from other engineering disciplines and construct software not from scratch but from pre-fabricated

[1]The diagrams used in the problem frame approach can easily be translated to UML class models, using a number of stereotypes. For a UML metamodel of problem frames, see [HHS08].

parts. Here, interface descriptions are crucial. These descriptions must suffice to decide if a given component is suitable for the purpose at hand or not. It should not be necessary (and, in some cases, it may even be impossible) to inspect the code of the component.

Using components makes it easier to construct long-lived software, because components can be replaced with new ones that e.g. provide enhanced functionality, see [HS04, LHHS07, CHS09]. Again, it becomes apparent that evolvability and long-livedness are deeply intertwined.

Component-based software development fits well with model- and pattern-based development: components can be described by models, and they can be used as instances of architectural patterns.

3 The ADIT development process

In the following, we describe the original development process ADIT that serves as basis for our evolution method in Sect. 4. ADIT is a model-driven, pattern-based development process also making use of components. In this paper we consider the analysis and design phases. To illustrate the different steps within the phases, we briefly describe what the purpose of the different steps is and how they can be realized. We also indicate in bold face the model, pattern, and component techniques that are relevant for the respective step. Should any of the three mentioned techniques not be described in a certain step, then the technique is not used in the respective step.

A1 Problem elicitation and description

To begin with, we need *requirements* that state our needs. Requirements are expressed in natural language, for example "A guest can book available holiday offers, which then are reserved until payment is completed.", "A staff member can record when a payment is received." In this step, also *domain knowledge* is stated, which consists of *facts* and *assumptions*. An example of a fact is that each vacation home can be used by only one (group of) guests at the same time. An example of an assumption is that each guest either pays the full amount due or not at all (i.e., partial payments are not considered). We now must find an answer to the question: "Where is the problem located?". Therefore, the environment in which the software will operate must be described. A **model** which can be used to answer the question is a *context diagram* [Jac01]. Context diagrams are similar to problem diagrams but it does not take requirements references into account. A context diagram for our vacation rentals example is shown in Fig. 3. Thus, the output of this step is: the context diagram, the requirements and the domain knowledge.

A2 Problem decomposition

We answer the question: "What is the problem?" in this second step. To answer the question, it is necessary to decompose the overall problem described in A1 into small manageable subproblems. For decomposing the problem into subproblems, related sets of requirements are identified first. Second, the overall problem is decomposed by means of **decomposition operators**. These operators are applied to context diagram. Examples of such operators are *Leave out domain* (with corresponding interfaces), *Refine phenomenon*, and *Merge several domains into one domain* . After applying the decomposition operators we have our set of subproblems with the corresponding operators that were applied to obtain the different subproblems. Furthermore, the subproblems should belong to known classes of software development problems, i.e., they should *fit to* **patterns** for such known classes of problems. In our case it should be possible to fit them to *problem frames*. Fitting a problem to a problem frame is achieved by instantiating the respective frame diagram.

Instantiated frame diagrams are called *problem diagrams*. These serve as **models** for this second analysis step. Thus, the output of this step is a set of problem diagrams being instances of problem frames.

A3 Abstract software specification
In the previous step, we were able to find out what the problem is by means of problem diagrams. However, problem diagrams do not state the order in which the actions, events, or operations occur. Furthermore, we are still talking about requirements. Requirements refer to problem domains, but not to the machine, i.e., the software that should be built. Therefore, it is necessary to transform requirements into specifications (see [JZ95] for more details). We use UML sequence diagrams as **models** for our specifications. Sequence diagrams describe the interaction of the machine with its environment. Messages from the environment to the machine correspond to *operations* that must be implemented. These operations will be specified in detail in Step A5. The output of this step is a set of specifications for each subproblem.

A4 Technical infrastructure
In this step, the technical infrastructure in which the machine will be embedded is specified. For example, a web application may use the Apache web server. The notation for the **model** in this step is the same as in A1, namely a context diagram. As it describes the technical means used by the machine for communicating with its environment, we refer to it as *technical context diagram*. In this step we can make use of **components** for the first time. Usually, we rely on the APIs of those prefabricated components in order to describe the technical means, e.g., on the API of the Apache web server.

A5 Operations and data specification
The **models** set up in this step are class diagrams for the internal data structures and pre- and postconditions for the operations identified in step A3. These two elements constitute the output of this step.

A6 Software life-cycle
In the final analysis step, the overall behavior of the machine is specified. Here, behavioral descriptions such as life-cycle expressions [CAB+94] can be used as a **model**. In our case this means that in particular, the relation between the sequence diagrams associated to the different subproblems is expressed explicitly. We can relate the sequence diagram sequentially, by alternative, or in parallel.

With the software life-cycle as output we conclude the analysis phase and move on to the design phase.

D1 Software architecture
The first step of the design phase is aimed at giving a coarse-grained structure to the software. This structure can be illustrated by using structural description **models**, such as UML 2.0 composite structure diagrams. We assign a candidate architecture to each subproblem, making use of architectural **patterns** for problem frames [CHH06]. Thus, we obtain a set of sub-architectures. Some **components** within these sub-architectures are pre-fabricated or pre-existing, e.g. for the web application example we make use of a pre-existing SMTP-Client. The architectural patterns lead us to a layered architecture consisting of an application layer, an interface abstraction layer (IAL) abstracting technical phenomena to application related ones, and a hardware abstraction layer (HAL) representing hardware drivers. The overall architecture must be derived from the subproblem architectures. The crucial point of this step is to decide if two components contained in

different subproblem architectures should occur only once in the global architecture, i.e., if they should be merged. To decide this question, we make use of the information expressed in Step A6 and by applying **merging rules**. An example for such a rule is "Adapters and storage components belonging to the same physical device or data storage are merged." These rules are described in more detail in [CHH06].

Figure 2 illustrates the global software architecture of our vacation rentals example, i.e. the output of this step. Furthermore, we have a first skeleton of the interfaces connecting the different components in our architecture by taking advantage of the information of the analysis phase [HH09].

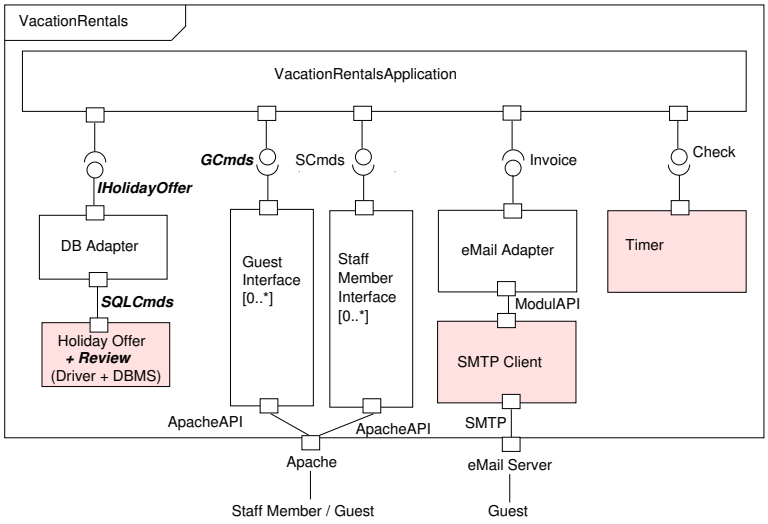


Figure 2: Global software architecture of vacation rentals

D2/ D3/ D4 Inter-component interaction/ intra-component interaction/ complete component or class behavior
Steps D2-D4 treat the fine-grained design. In these steps, the internal structure of the software is further elaborated. Several **patterns** can be applied in these steps, such as design patterns, patterns for state machines, etc.

Note that we are not treating these steps further in this paper.

4 Evolution Method

During the long lifetime of a software it is necessary to modify and update it to meet new or changed requirements and/or environment to accommodate it to the changing environment where it is deployed in. This process is called *software evolution*. The new requirements that should be met are called *evolution requirements*. An example for such an evolution requirement is "Guests can write a review, after they have left their vacation home." Modified or additional domain knowledge is referred to as aD. An example for some new domain knowledge is the additional assumption that "Guests write fair reviews".

We define a corresponding evolution step for each ADIT step. These steps address the special needs arising in software evolution by providing operators leading from one model to another and an explicit tracing between the different models. Basically, we perform a replay of the original method, adding some support for software evolution.

In the following we illustrate the steps to be performed for software evolution. As an example, we evolve the vacation rentals application introduced in Sect. 3.

EA0 Requirements relation
Not all development models will be affected by the evolution task. Therefore, we first have to identify those models that are relevant. For that purpose, we relate the evolution requirements to the original requirements. We identified several relations the original and evolution requirements may share:

- **similar**: a functionality similar to the one to be incorporated exists.
- **extending**: an existing functionality is refined or extended by the evolution task.
- **new**: the functionality is not present yet, and it is not possible to find anything that could be similar or extended.
- **replacing**: the new functionality replaces the existing one.

As a means of representation, we use a table. This table –together with other tables that are created during the process – serve for tracing purposes. An example of such a representation is shown in Tab. 1.

The entry "recordPayment" in the first row refers to the requirement involving the staff member which has been introduced in Sec. 3. The entry "writeReview" in the first column refers to the above mentioned evolution requirement. The table is read as follows: "writeReview" is similar to "recordPayment".

	...	recordPayment	...
writeReview		similar	
⋮		⋮	

Table 1: Excerpt of relation between requirements and evolution requirements for vacation rentals

The requirements sharing a relation with the evolution requirements are collected to form the set of relevant requirements (*rel_set*). This set then constitutes the focus for our further investigation.

EA1 Adjust problem elicitation and description
We use the output of A1, i.e., context diagram as input for the first step of the evolution method. We revise this context diagram by incorporating the new/changed requirements and domain knowledge to it. To support the engineers, we defined evolution operators, similar to the original operators, to ease modifying the context diagram. The **operators** relevant for this step are (for more details refer to [CHW07]):

add new domain – A new domain has to be added to the context diagram.

modify existing domain – A domain contained in the context diagram has to be modified, e.g. by splitting or merging.

add new phenomenon – A new phenomenon is added to an interface of the context diagram.

modify existing phenomenon – An existing phenomenon has to be modified in the context diagram, e.g. by renaming.

In some cases, it may also occur that neither domains nor shared phenomena are newly introduced. Then, no changes to the context diagram are necessary at this place, but the new requirements/domain knowledge may require changes in later steps.

The resulting context diagram now represents the new overall problem situation (cf. Fig. 3). The modifications are highlighted through bold-italic font, e.g., a phenomenon *writeReview* has been added to the interface between the domains guest and vacation rentals (operator add new phenomenon). Furthermore, add new domain and add new phenomenon have been applied to introduce the domain *Review* with the corresponding phenomenon *writingReview*.

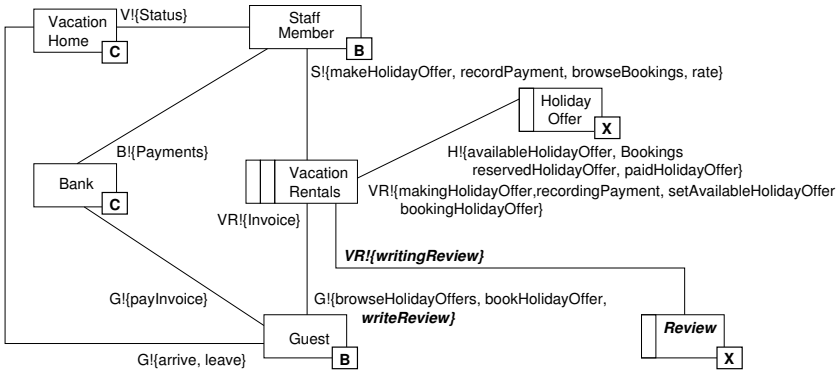


Figure 3: Context diagram with revisions

EA2 Adjust problem decomposition

In this step we take the output of A2, i.e., the resulting set of problem diagrams, as well as the operators that were applied to the context diagram for the decomposition as input. Furthermore, we use both the context diagram created in EA1, as well as the rel_set of EA0 as additional input. As we modified the context diagram by applying some operators, this has impacts on the problem decomposition, as well. Therefore, it is necessary to investigate the existing subproblems. We again define evolution operators to guide the engineer.

Examples of evolution **operators** for this step are:

integrate eR in existing problem diagram – New domains and associated shared phenomena may be added to an existing problem diagram.

eR cannot be added to existing subproblem - create new one – The eR is assigned to a given subproblem, but the resulting subproblem then gets too complex. Hence, it is necessary to split the subproblem into smaller subproblems.

More details on these operators can be found in [CHW07].

In addition it is necessary to check whether the resulting problem diagrams still fit to problem frames or, for new subproblems, to find an appropriate frame to be instantiated, e.g. via **operators** such as *fit to a problem frame* or *choose different problem frame*.

Figure 4 shows the problem diagram for the evolution requirement "writeReview" (operator *eR cannot be added to existing subproblem - create new one*). As we know that it is similar to "recordPayment" we check whether it is possible to apply the same problem frame. In this case, this is the problem frame *simple workpieces* (cf. Fig. 1). It is obvious that our problem diagram is an instance of that problem frame (operator *fit to a problem frame*).

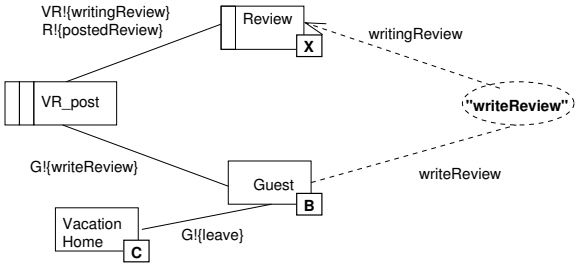


Figure 4: Problem diagram for "writingReview"

EA3 Adjust abstract software specification
The sequence diagrams of A3 serve as input for this step, together with the results of EA2. Whenever an existing sequence diagram has to be investigated, the following cases may occur:

- Operator *integrate eR in existing problem diagram* was applied.

  Operator name: use existing sequence diagram

  The existing diagram has to be modified in such a way that it can handle the additional behavior by adding messages, domains etc. to the corresponding sequence diagram. The changes reflect the modifications made in EA2. It may also be necessary to add new sequence diagrams, as well.

- The parameters of existing messages must be adapted.

  Operator name: modify parameter

  Existing parameters must be either modified (rename), extended (add) or replaced. Otherwise, the sequence diagram remains unchanged.

- Operator *eR cannot be added to existing subproblem - create new one* was applied.

  Operator name: no operator

  The original ADIT step must be applied.

EA4 Adjust technical software specification
The original technical context diagram is adapted to meet the new or changed situation. The procedure is the same as in A4.

EA5 Adjust operations and data specification
The actions performed in EA3 trigger the modification. For example, a message on the sequence diagram as been modified. These changes are then also made in the corresponding operations in this step. For newly introduced operations, the original method is applied.

EA6 Adjust software life-cycle
New relations need to be incorporated. Existing relations need to be adapted, accordingly. However, the behavior that has not been altered by the evolution task must be preserved.

ED1 Adjust software architecture
We use the revised subproblems of Step EA2 as input. For our example we were able to apply the same architectural style for "writeReview" as we did for "recordPayment". Therefore, we can apply the same merging rules, as well. Figure 2 illustrates the resulting global architecture. The changes are indicated in italics and bold face.

The interfaces are adapted according to the changes made in the analysis phase based on **operators** such as *modify interface*, *add new component*, or *add new interface*. For example, we introduced a new message "writeReview()" in step EA3 (as consequence of operator *eR cannot be added to existing subproblem - create new one*). By doing a replay of the merging rules, we know, that it is necessary to extend the existing interfaces *IHolidayOffer* and *GCmds* (operator *modify interface*). As writing a review is performed after guests have taken the holiday offer the component *Review* can be merged with the existing component *Holiday Offer* (operator *modify component*) resulting in a modification of the corresponding interface *SQLCmds* (operator *modify interface*), as well.

5 Related Work

This work takes up ideas from modern software engineering approaches and processes, such as the Rational Unified Process (RUP) [JBR99], Model-Driven Architecture (MDA) [MSUW04], and Service-Oriented Architecture (SOA) [Erl05]. All these approaches are model-driven, which means that in principle, an evolution process for them can be defined in a similar way as for the ADIT process.

The work of O'Cinnéide and Nixon [ON99] aims at applying design patterns to existing legacy code in a highly automated way. They target code refactorings. Their approach is based on a semi-formal description of the transformations themselves, needed in order to make the changes in the code happen. They describe precisely the transformation itself and under which pre- and postconditions it can successfully be applied.

Our approach describes and applies model transformation in a rather informal way. The same is true for the transformation approaches cited above. Czarnecki and Helsen give an overview of more formal model transformation techniques [CH03].

Researchers have used versions and histories to analyze different aspects considering software evolution. Ducasse et al. [DGF04] propose a history meta-model named HISMO. A history is defined as a sequence of versions. The approach is based on transformations aimed at extracting history properties out of structural relationships. Our approach does not consider histories andhow to analyze them. In contrast, we introduce a method for manipulating an existing software and its corresponding documentation in a systematic way to perform software evolution.

ROSE is the name of a tool created by Zeller et al. [ZWDZ05], which makes use of data mining techniques to extract recurring patterns and rules that allow to offer advice for new evolution tasks. The tool can propose locations where a change might occur based on the current change, help to avoid incomplete changes, and detect coupling that would not be found by performing program analysis. It does, however, not provide help on what to do once a potential location has been identified. With our method we intend to provide help to systematically modify source code after the corresponding part has been located.

Detecting logical coupling to identify dependencies among modules etc. has been the research topic of Gall et al.[GHJ98]. Those dependencies can be used to estimate the effort needed to carry out maintenance tasks, to name an example. Descriptions in change reports are used to verify the detected couplings. The technique is not designed to change the functionality of a given software.

Others investigate software evolution at run-time [Piz02]. Evolving a software system at run-time puts even more demands on the method used than ordinary software systems. Our approach does not take run-time evolution into account.

Sillito et al. [SMDV06] conducted a survey to capture the main questions programmers ask when confronted with an evolution task. These fit well to our method as they can be used to refine the understanding of the software at hand especially in later phases.

We agree with Mens and D'Hondt [MD00] that it is necessary to treat and support evolution throughout all development phases. They extend the UML meta-model by their so-called evolution contracts for that reason. The aim is to automatically detect conflicts that may arise when evolving the same UML model in parallel. This mechanism can very well be integrated into our method to enhance the detection of inconsistencies and conflicts. Our approach, however, goes beyond this detection process. It strives towards an integral method for software evolution guiding the software engineer in actually performing an evolution task throughout all development phases.

The field of software evolution cannot be examined in isolation as it has contact points with other disciplines of software engineering. An example is the work of Demeyer et al. [DDN02]. They provide a pattern system for object-oriented reengineering tasks. Since software evolution usually involves some reengineering and also refactoring [Fow00] efforts, it is only natural that known and successful techniques are applied in software evolution, as well. Software evolution, however, goes beyond restructuring source code. Its main goal is to change the functionality of a given software.

Furthermore, software evolution can profit from the research done in the fields of feature location e.g. [ESW06, KQ05], re-documentation, e.g. [CY07, WTM+95], and agile development processes such as extreme programming [Bec99].

Finally, we have to mention another paper of ours on pattern-based software evolution for component-based systems [CHS09]. In this paper, we give architectural evolution patterns that can be used to evolve component architectures.

6 Conclusions

In this paper, we have pointed out that using models, patterns, and components is a promising approach to develop long-lived software. However, long-lived software needs to undergo evolution. Therefore, we have shown how software that was developed according to a model-/pattern-/component-based process can be evolved in a systematic way. We have applied our method successfully on several software systems amongst them open source software such as *Doxygen* [Hee09].

In the future, we intend to elaborate on the model transformations that are needed to perform the evolution steps. We will formalize the model transformations and also provide further tool support for the evolution process. Furthermore, we plan on conducting further evolution tasks on other open source projects to validate the method as well as the tool.

References

[Bec99] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[Bin00] Robert Binder. *Testing Object-Oriented Systems*. Addison-Wesley, 2000.

[CAB+94] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development: The Fusion Method*. Prentice Hall, 1994. (out of print).

[CD01] John Cheesman and John Daniels. *UML Components – A Simple Process for Specifying Component-Based Software*. Addison-Wesley, 2001.

[CH03] Krzysztof Czarnecki and Simon Helsen. Classification of Model Transformation Approaches. In *Proc. of the 2nd OOPSLA Workshop on Generative Techniques in the Context of MDA*, 2003.

[CHH06] C. Choppy, D. Hatebur, and M. Heisel. Component composition through architectural patterns for problem frames. In *Proc. XIII Asia Pacific Software Engineering Conf.*, pages 27–34. IEEE Computer Society, 2006.

[CHS09] Isabelle Côté, Maritta Heisel, and Jeanine Souquières. On the Evolution of Component-based Software. In *4th IFIP TC2 Central and Eastern European Conf. on Software Engineering Techniques CEE-SET 2009*. Springer-Verlag, 2009. to appear.

[CHW07] Isabelle Côté, Maritta Heisel, and Ina Wentzlaff. Pattern-based Exploration of Design Alternatives for the Evolution of Software Architectures. *Int. Journal of Cooperative Information Systems, World Scientific Publishing Company*, Special Issue of the Best Papers of the ECSA'07, December 2007.

[Cop92] J. O. Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, 1992.

[CY07] Feng Chen and Hongji Yang. Model Oriented Evolutionary Redocumentation. In *COMPSAC '07: Proc. of the 31st Annual Int. Computer Software and Applications Conference - Vol. 1- (COMPSAC 2007)*, pages 543–548, Washington, DC, USA, 2007. IEEE Computer Society.

[DDN02] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[DGF04] Stéphane Ducasse, Tudor Gîrba, and Jean-Marie Favre. Modeling Software Evolution by Treating History as a First Class Entity. In *Proc. on Software Evolution Through Transformation (SETra 2004)*, pages 75–86, Amsterdam, 2004. Elsevier.

[EGG+09] G. Engels, M. Goedicke, U. Goltz, A. Rausch, and R. Reussner. Design for Future – Legacy-Probleme von morgen vermeidbar? *Informatik-Spektrum*, 2009.

[Erl05] T. Erl. *Service-Oriented Architecture (SOA): Concepts, Technology, and Design*. Prentice Hall PTR, 2005.

[ESW06] Dennis Edwards, Sharon Simmons, and Norman Wilde. An approach to feature location in distributed systems. In *Journal of Systems and Software*, 2006.

[For06] UML Revision Task Force. *OMG Unified Modeling Language: Superstructure*, April 2006. http://www.omg.org/docs/ptc/06-04-02.pdf.

[Fow97] M. Fowler. *Analysis Patterns: Reusable Object Models*. Addison Wesley, 1997.

[Fow00] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000.

[GHJ98] Harald Gall, Karin Hajek, and Mehdi Jazayeri. Detection of Logical Coupling Based on Product Release History. In *ICSM '98: Proc. of the Int. Conf. on Software Maintenance*, page 190, Washington, DC, USA, 1998. IEEE Computer Society.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, 1995.

[Hee09] D. van Heesch. Doxygen - A Source Code Documentation Generator Tool, 2009. `http://www.stack.nl/~dimitri/doxygen`.

[HH09] D. Hatebur and M. Heisel. Deriving Software Architectures from Problem Descriptions. In Jürgen Münch and Peter Liggesmeyer, editors, *Workshop Modellgetriebene Softwarearchitektur – Evolution, Integration und Migration (MSEIM), Software Engineering 2009*, LNI P-150, pages 383–392, Bonn, 2009. Bonner Köllen Verlag.

[HHS08] Denis Hatebur, Maritta Heisel, and Holger Schmidt. A Formal Metamodel for Problem Frames. In *Proc. of the Int. Conf. on Model Driven Engineering Languages and Systems (MODELS)*, volume 5301, pages 68–82. Springer Berlin / Heidelberg, 2008.

[HS04] Maritta Heisel and Jeanine Souquières. Adding Features to Component-Based Systems. In M.D. Ryan, J.-J. Ch. Meyer, and H.-D. Ehrich, editors, *Objects, Agents and Features*, LNCS 2975, pages 137–153. Springer, 2004.

[Jac01] M. Jackson. *Problem Frames. Analyzing and structuring software development problems*. Addison-Wesley, 2001.

[JBR99] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.

[JZ95] M. Jackson and P. Zave. Deriving Specifications from Requirements: an Example. In *Proc. 17th Int. Conf. on Software Engineering, Seattle, USA*, pages 15–24. ACM Press, 1995.

[KQ05] Rainer Koschke and Jochen Quante. On dynamic feature location. In *ASE '05: Proc. of the 20th IEEE/ACM Int. Conf. on Automated Software Engineering*, pages 86–95, New York, NY, USA, 2005. ACM.

[LHHS07] Arnaud Lanoix, Denis Hatebur, Maritta Heisel, and Jeanine Souquières. Enhancing Dependability of Component-Based Systems. In N. Abdennadher and F. Kordon, editors, *Reliable Software Technologies – Ada Europe 2007*, LNCS 4498, pages 41–54. Springer, 2007.

[MD00] Tom Mens and Theo D'Hondt. Automating support for software evolution in UML. *Automated Software Engineering Journal*, 7(1):39–59, February 2000.

[MSUW04] S. J. Mellor, K. Scott, A. Uhl, and D. Weise. *MDA Distilled*. Addison-Wesley Professional, 2004.

[ON99] M. O'Cinnéide and P. Nixon. A Methodology for the Automated Introduction of Design Patterns. In *ICSM '99: Proc. of the IEEE Int. Conf. on Software Maintenance*, page 463, Washington, DC, USA, 1999. IEEE Computer Society.

[Par94] D. L. Parnas. Software aging. In *ICSE '94: Proc. of the 16th Int. Conf. on Software Engineering*, pages 279–287, Los Alamitos, CA, USA, 1994. IEEE Comp. Soc. Press.

[Piz02] M. Pizka. STA – A Conceptual Model for System Evolution. In *Int. Conf. on Software Maintenance*, pages 462 – 469, Montreal, Canada, October 2002. IEEE CS Press.

[SG96] M. Shaw and D. Garlan. *Software Architecture. Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[SGM02] C. Szyperski, D. Gruntz, and S. Murer. *Component Software*. Pearson Education, 2002. Second edition.

[SMDV06] J. Sillito, G. C. Murphy, and K. De Volder. Questions programmers ask during software evolution tasks. In *SIGSOFT '06/FSE-14: Proc. of the 14th ACM SIGSOFT Int. Symposium. on Foundation of Software Engineering*, pages 23–34, New York, NY, USA, 2006. ACM.

[WTM+95] K. Wong, S. R. Tilley, H. A. Muller, M. D. Storey, and T. A. Corbi. Structural redocumentation: A case study. *IEEE Software*, 12:46–54, 1995.

[ZWDZ05] T. Zimmermann, P. Weissgerber, S. Diehl, and A. Zeller. Mining Version Histories to Guide Software Changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, June 2005.

Engineering and Continuously Operating Self-Adaptive Software Systems: Required Design Decisions

André van Hoorn[1], Wilhelm Hasselbring[2], and Matthias Rohr[1,3]

[1] Graduate School TrustSoft, University of Oldenburg, D-26111 Oldenburg
[2] Software Engineering Group, University of Kiel, D-24098 Kiel
[3] BTC AG – Business Technology Consulting AG, D-26121 Oldenburg

Abstract: Self-adaptive or autonomic systems are computing systems which are able to manage/adapt themselves at runtime according to certain high-level goals. It is appropriate to equip software systems with adaptation capabilities in order to optimize runtime properties, such as performance, availability, or operating costs. Architectural models are often used to guide system adaptation. When engineering such systems, a number of cross-cutting design decisions, e.g. instrumentation, targeting at a system's later operation/maintenance phase must and can be considered during early design stages.

In this paper, we discuss some of these required design decisions for adaptive software systems and how models can help in engineering and operating these systems. The discussion is based on our experiences, including those gathered from evaluating research results in industrial settings. To illustrate the discussion, we use our self-adaptation approach SLAstic to describe how we address the discussed issues. SLAstic aims to improve a software system's resource efficiency by performing architecture-based runtime reconfigurations that adapt the system capacity to varying workloads, for instance to decrease the operating costs.

1 Introduction

Self-managed or autonomic systems are those systems which are able to adapt themselves according to their environment [11]. Self-adaptation can be described as a cycle of three logical phases [19]: observation, analysis, and adaptation. The observation phase is concerned with monitoring (e.g., system behavior or system usage). The analysis phase detects triggers for adaptation and, if required, selects suitable adaptation operations, which are executed in the adaptation phase. A recent survey on self-adaptation research can be found in [5].

When engineering such systems, various non-functional aspects need to be considered. Trade-offs between QoS (Quality of Service) requirements, such as a performance, availability, and operating costs have to be addressed. Thus, requirements of system operation have a significant impact on the required design decisions of system engineering.

We illustrate the discussion of required design decisions based on the description of our self-adaptation approach SLAstic [24]. SLAstic aims to improve the resource efficiency

of component-based software systems, with a focus on business-critical software systems. Architectural models, specifying both functional and non-functional properties, play a central role in the SLAstic approach as they are not only used for the specification of the system at design time. The same models are updated by measurement data at runtime and used for the required analysis tasks in order to achieve the self-adaption goals. SLAstic explicitly considers a flexible system capacity in the software architecture.

The contribution of this paper is a discussion of design decisions that are required at *design time* for an effective operation at *runtime* of continuously operating software systems, such as business-critical software systems. We discuss the role of models in this context. This discussion is based on our experience and lessons learned from lab studies and industrial field studies with our monitoring framework Kieker [20].[1] We illustrate how these design decisions are addressed in our SLAstic approach for adaptive capacity management.

This paper is structured as follows: Section 2 describes the architecture-based self-adaptation approach SLAstic aiming to support a resource-efficient operation of software systems. Based on this, Section 3 discusses design decisions for engineering and operating self-adaptive software systems and how they are addressed in the SLAstic approach. The conclusions are drawn in Section 4.

2 The SLAstic Framework

With SLAstic, resource efficiency of continuously operating software is improved by adapting the number of allocated server nodes and by performing fine-grained architectural reconfigurations at runtime according to current system usage scenarios (workload) while satisfying required external QoS objectives (as specified in service level agreements). It is not the goal of this paper to present the SLAstic framework in detail. Instead, it is intended to be used for illustration in our discussion in Section 3.

2.1 Assumptions on Software System Architectures

We assume a hierarchical software system architecture consisting of the following types of hardware and software entities: (1) *server node*, (2) *component container*, and (3) *software component*. This allows to design the SLAstic self-adaptation framework in a way that provides reusability among different system implementations sharing common architectural concepts. Figure 1 illustrates this hierarchical structure with a Java EE example.

The business logic of a software system is implemented in a number of software components with well-defined provided and required interfaces. Software components constitute units of deployment [22] and can be assembled or connected via their interfaces. Through their provided interfaces, they provide services to other software components or systems. Software components are deployed into component containers, which constitute

[1] `http://kieker.sourceforge.net`

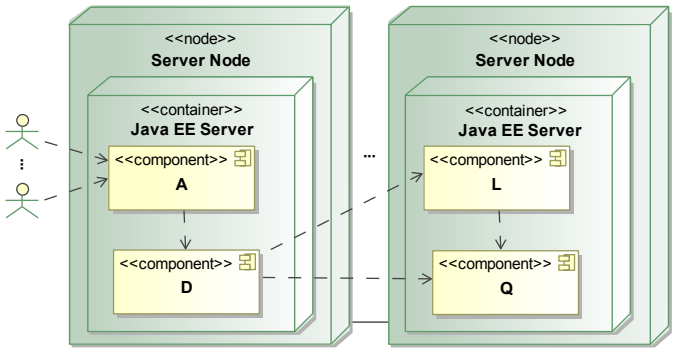


Figure 1: Illustration of a hierarchical software system architecture

their execution context and provide middleware services, such as transaction management. Moreover, a component container manages the thread pool used to execute external service requests. Web servers or Java EE application servers are typical examples for component containers. A component container is installed on a server node consisting of the hardware platform and the operating system. Server nodes communicate via network links. For a software system, a set of server nodes is allocated from a *server pool*. We denote the set of allocated server nodes, the assembly of software components, as well as its deployment to component containers as the software system's *architectural configuration*.

Structural and behavioral aspects of a software system's architecture can be described using architecture description languages (ADL). Components, interfaces, and connectors for architectural configurations are common (structural) features of ADLs [16]. A recent survey of ADLs can be found in [23].

2.2 Framework Architecture

The SLAstic framework aims to equip component-based software systems, as described in the previous Section 2.1, with self-adaptation capabilities in order to improve its resource efficiency. Figure 2 depicts how the concurrently executing SLAstic components for monitoring (SLAstic.MON), reconfiguration (SLAstic.REC), and adaptation control (SLAstic.CONTROL) are integrated and how they interact with the monitored software system. It shows the typical, yet rather general, external control loop for adaptation [9].

The software system is instrumented with monitoring probes which continuously collect measurement data from the running system. The SLAstic.MON component provides the monitoring infrastructure and passes the monitoring data to the SLAstic.CONTROL component. The SLAstic.CONTROL component, detailed later in Figure 4 of Section 3.2, analyzes the current architectural configuration with respect to the monitoring data and, if required, determines an adaptation plan consisting of a sequence of reconfiguration operations. The adaptation plan is communicated to the SLAstic.REC component which

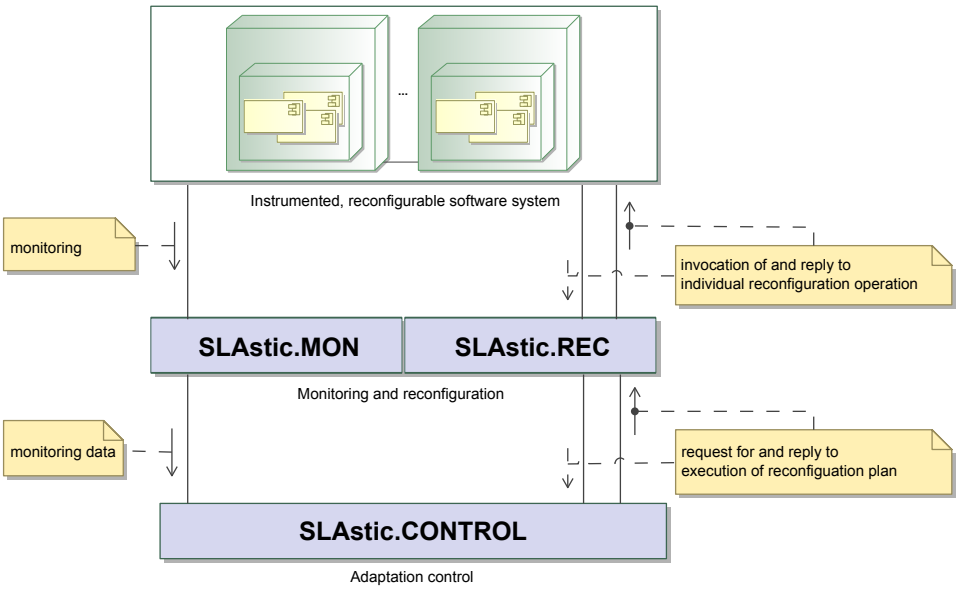


Figure 2: SLAstic self-adaptation system overview

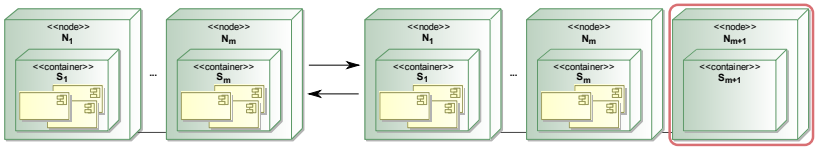
is responsible for executing the actual reconfiguration operations. SLAstic employs our Kieker framework for continuous monitoring and online analysis.

2.3 Reconfiguration Operations

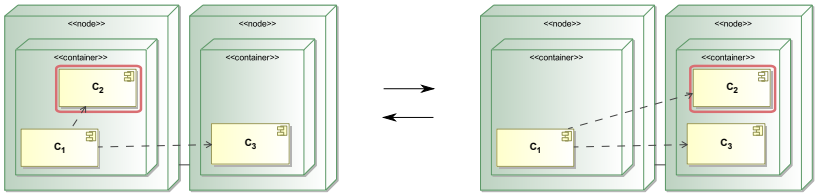
Although, the SLAstic framework, in principle, supports arbitrary architectural reconfiguration operations, we will restrict us to the following three operations, illustrated in Figure 3.

(1) **Node Allocation & Deallocation.** A server node is allocated or deallocated, respectively. In case of an allocation, this includes the installation of a component container, but it does not involve any (un)deployment operation of software components. Intuitively, the goal of the allocation is the provision of additional computing resources and the goal of the deallocation is saving operating costs caused by power consumption or usage fees, e.g. in cloud environments.

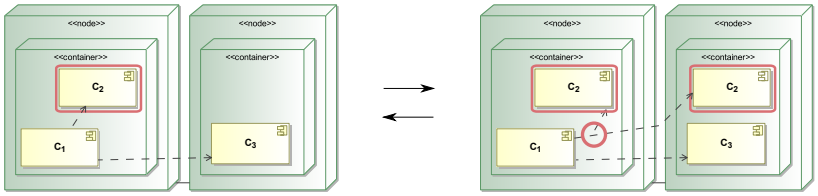
(2) **Component Migration.** A software component is undeployed from one execution context and deployed into another. The goals of this fine-grained application-level operation are both to avoid the allocation of additional server nodes or respectively to allow the deallocation of already allocated nodes by executing adaptation operation (1).



(a) node (de-)allocation



(b) component migration



(c) component (de-)replication

Figure 3: SLAstic reconfiguration operations

(3) **Component (de-)Replication** This application-level operation consists of the duplication of a software component and its deployment into another execution context (as well as the reverse direction). Future requests to the component are distributed between the available component instances. The goals of this application-level operation are the same as the goals of operation (2).

3 Required Design Decisions and the Role of Models

The following three Subsections 3.1–3.3 discuss design decisions and the role of models related to monitoring, analysis, and adaptation, respectively.

3.1 Monitoring

In order to obtain the required information to control adaptation, a system needs to be instrumented with monitoring probes. Monitoring can be performed at the different abstraction levels of a software system, i.e., platform (e.g., CPU utilization), container (e.g., number of available threads in a thread pool), and application (e.g., service response times)

level. For example, SLAstic continuously requires performance data, such as CPU utilization, workload (system usage), and service response times.

Since adaptation concerns a running system in production use, continuous monitoring of the system state is required. We argue that the integration of monitoring should be designed in a systematic way and although mainly becoming relevant not before a system's operation phase, its integration should be considered early in the engineering process. In the following paragraphs we will discuss design decisions regarding continuous monitoring. In [7], we present a process model for application-level performance monitoring of information system landscapes. The design decisions should be integrated into such an engineering process.

Selection of Monitoring Probes A monitoring probe contains the logic which collects and possibly pre-processes the data of interest from within the application. Probes can measure externally visible behavior, e.g. service response times, but also application-internal behavior, such as calling dependencies between components.

SLAstic requires different types of monitoring probes, collecting workload, resource usage, and timing data. An example not focusing on self-adaptation is application level failure diagnosis [1, 14] requiring probes for monitoring response times and internal control flow of operation executions.

In practice, new application-internal monitoring probes are often only introduced in an ad-hoc manner, as a result of a system failure. For example, a probe for monitoring the number of available database connections in a connection pool may have been introduced. The selection of the types of monitoring probes must be driven by the goal to be achieved by the gathered monitoring data and depends on the analysis concern.

Number and Position of Monitoring Points In addition to the above-mentioned decision of what types of monitoring probes are integrated into the system, important and very difficult decisions regard the number and the exact locations of monitoring points. This decision requires a trade-off between the information quality available to the analysis tasks and the overhead introduced by possibly too fine-grained instrumentation leading to an extensive size of the monitoring log. As the type of monitoring probes to use, does the number and position depend on the goal of monitoring. Additionally, the different usage scenarios of the application must be considered, since an equally distributed coverage of activated monitoring points during operation is desirable.

Intrusiveness of Instrumentation A major maintainability aspect of application-level monitoring is how monitoring logic is integrated into the business logic. Maintainability is reduced if the monitoring code is mixed with the source code of the business logic, because this reduces source code readability.

We regard the use of the AOP (Aspect-Oriented Programming) paradigm [12] as an extremely suitable means to integrate monitoring probes into an application [8]. A popular Java-based AOP implementation is AspectJ. Many middleware technologies provide sim-

ilar concepts. Examples are the definition of filters for incoming Web requests in the Java Servlet API, the method invocation interceptors in the Spring framework, or handlers for incoming and outgoing SOAP messages in different Web service frameworks. Kieker currently supports AOP-based monitoring probes with AspectJ, Spring, Servlets, and SOAP.

Physical Location of the Monitoring Log The monitoring data collected within the monitoring probes is written to the so-called monitoring log. The monitoring log is typically located in the filesystem or in a database. The decision which medium to use depends on the amount of monitoring data generated at runtime, the required timeliness of analysis, and possibly restricted access rights or policies. A filesystem-based monitoring log is fast, since usually no network communication is required. The drawback is that online analysis of the monitoring data is not possible or at least complicated in a distributed setting. A database brings the benefit of integrated and centralized analysis support such as convenient queries but is slower than a file system log due to network latencies and the overhead introduced by the DBMS. Moreover, the monitoring data should not be written to the monitoring log synchronously since this has a considerable impact on the timing behavior of the executing business service.

Since SLAstic performs the analysis online, it requires fast and continuous access to recent monitoring data. For this reason, the SLAstic.MON and SLAstic.CONTROL components communicates monitoring data via JMS messaging queues. Kieker includes different synchronous and asynchronous monitoring log writers for filesystem, database, and for JMS queues.[2] Customized writers can be integrated into Kieker.

Monitoring Overhead It is clear that continuous monitoring introduces a certain overhead to the running system. Of course, the overall overhead depends on the number of activated monitoring points and its activation frequency. The overhead for a single activated monitoring point depends on the delays introduced by the resource demand and process synchronizations in the monitoring probes, the monitoring control logic, and particularly I/O access for writing the monitoring data into the monitoring log.

It is a requirement that the monitoring framework is as efficient as possible and that the overhead increases linearly with the number of activated monitoring points, i.e., each activation of a monitoring point should add the same constant overhead. Of course, this linear scaling is only possible up to a certain point and depends on the average number of activated monitoring, i.e., the granularity of instrumentation.

Kieker has been used to monitor some production systems in the field. For these systems, no major impact was reported, despite the fact that detailed trace information is monitored for each incoming service request. In benchmarks, we observed an overhead that was in the order of microseconds on modern desktop PCs for each activated instrumentation point. We are currently working on a systematic assessment of the overhead introduced by the Kieker monitoring framework.

[2] `http://java.sun.com/products/jms/`

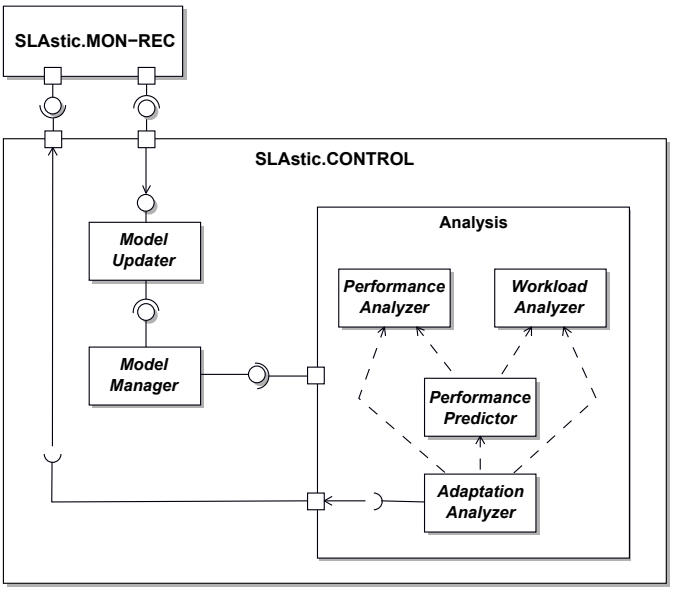


Figure 4: Internal architecture of the SLAstic.CONTROL component

Model-Driven Instrumentation Model-driven software development (MDSD) [21] provides a convenient means to lift the level of abstraction for instrumentation from source code to the architectural or even business level. Static or dynamic design models can be annotated with monitoring information whose transformation into the platform-specific instrumentation can be integrated into the MDSD process. Possible annotation variants are tagging, i.e., adding the instrumentation to the functional part of the model, and decoration, i.e., the definition of a dedicated monitoring model referring to the architectural entities captured in external models. Examples for the model-driven instrumentation for monitoring SLAs employing (standardized) meta-models (e.g., [6, 18]), come from the domain of component-based [4] and service-based systems [17].

3.2 Analysis

Analysis or adaptation control involves analyzing and, if required, planning and triggering the execution of system adaptation. Analysis and planning is based on architectural knowledge about the system, the interpretation of monitoring data, and specified adaptation. Analysis and planning may be performed autonomically or on-demand and manually by the system administrator.

SLAstic analyzes the instrumented and reconfigurable system while it is running, i.e. online. The internal architecture of the SLAstic.CONTROL component with its internal, concurrently executing subcomponents is shown in Figure 4.

Runtime Models Design models can be used as or transformed into architectural runtime models constituting an abstract representation of the current state and thus being usable as a basis for analysis. Models more suitable for the performed analysis methods can be automatically derived from the architectural models. For example, for performance prediction, a number of transformations from design models to analysis models, such as variants of queueing networks, were developed [2].

The architectural SLAstic runtime models for the analysis are accessible through the *Model Manager* component which synchronizes access to these models in order to maintain consistent model state. The *Model Updater* component updates the runtime models based on the monitoring data received from the SLAstic.MON component. In the SLAstic framework, we are planning to use (extended) model instances of the Palladio metamodel [3], a domain-specific ADL designed for performance prediction of component-based software systems.

Runtime Analysis Methods A challenging decision is the selection of appropriate runtime analysis methods, which depends on a number of factors, such as the arrival rate of incoming monitoring data, variability in the data, available computational resources, required accuracy of the analysis et cetera. The integration of the adaptation control in a feedback loop with the controlled system, usually requires timely analysis results and adaptation decisions.

The *Performance Predictor* in the SLAstic.CONTROL component predicts the performance of the current architectural configuration based on the performance and workload analysis, including a forecast of the near-future workload, performed by the *Performance Analyzer* and *Workload Analyzer* components. The *Adaptation Analyzer* determines an adaptation plan which is communicated to the SLAstic.REC component for execution. The *Model Updater* updates the runtime model with respect to the new architectural configuration. A number of analysis methods is imaginable in each of the analysis components. The SLAstic framework allows to evaluate different analysis methods by its extendable architecture.

3.3 Adaptation

The actual execution of an adaptation plan involves the execution of the included architectural reconfiguration operations. The reconfiguration can be performed offline or at runtime. Offline reconfiguration means, that the running system is shut down, reconfigured, and restarted. When reconfigured at runtime, a system is changed without a restart, and while it is running and serving requests. In practice, adaptation or maintenance, e.g. for system repair, is usually performed offline. Many companies have fixed “maintenance windows”, i.e., timeframes during which the provided services are not available and a new version of the software system is deployed. This section focuses on reconfiguration at runtime.

Specification of Reconfiguration Operations State charts, or other modeling techniques can be used to specify how reconfiguration operations are executed, e.g., in terms of a reconfiguration protocol. The benefit of having a formal protocol is that its correctness can be verified using formal analysis, and that quantitative analyses regarding the reconfiguration may be performed. This is helpful in determining an adaptation plan using certain reconfiguration properties. For example, regarding the SLAstic *Node Allocation* operation (see Section 2.3), the average time period between an allocation request for a node and its readiness for operation.

In this paper, we focus on business-critical software systems. In such systems, service requests are usually executed as transactions. A specification of the SLAstic component migration operation would include a definition of how transactions are handled. For example, when migrating a software component, active transactions using the component to be migrated would be completed and newly arriving transactions would be dispatched to the migrated instances. In [15], we described our J2EE-based implementation of a redeployment reconfiguration operation, that is based on a reconfiguration protocol defined using state charts.

Design of Reconfigurable Software Architectures Reconfiguration operations can be defined based on architectural styles [10]. This allows to re-use the reconfiguration operations for systems having this architectural style. The SLAstic reconfiguration operations can be applied to systems complying to the architectural assumptions sketched in Section 2.1.

This reconfiguration capability needs to be integrated into the software system architecture. Likewise to the annotation of design models with information regarding monitoring instrumentation, as described in the previous Section 3.1, reconfiguration-specific annotations can be added to design models. For example, software components could be annotated with the information whether they are designed to be replicated or migrated at runtime using the SLAstic reconfiguration operations.

Transactional Reconfiguration An adaptation plan should be executed as a transaction, in order to bring the system from one consistent architectural configuration into another [13]. This means, that an adaptation plan is only committed if all included reconfiguration operations were successfully executed. In SLAstic we will only permit the execution of one adaptation plan at a time.

4 Conclusions

Requirements of system operation have a significant impact on the required design decisions in system engineering. In this paper, we discussed the design decisions that are required at *design time* for an effective operation at *runtime* of continuously operating software systems, such as business-critical software systems. This discussion is based on our experiences and lessons learned from lab studies and industrial field studies with our

monitoring framework Kieker, and illustrated using our self-adaptation approach SLAstic. SLAstic aims to improve the resource efficiency of component-based software systems.

Adaptation impacts running systems in production use, thus, continuous monitoring of the system state is required if the systems are business-critical. The integration of monitoring should be considered early in the engineering process. Our message is that requirements of system operation should be considered early in the design process, not as an afterthought as often observed in practice. Architectural models should be used both at design and runtime. However, many of the discussed design decisions are not only relevant to online self-adaptation, but also to offline maintenance activities. Concerns of self-adaptation should, similar to monitoring, be considered as cross-cutting concerns.

References

[1] M. K. Agarwal, K. Appleby, M. Gupta, G. Kar, A. Neogi, and A. Sailer. Problem determination using dependency graphs and run-time behavior models. In *15th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM 2004)*, volume 3278 of *LNCS*, pages 171–182, 2004.

[2] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: A survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, 2004.

[3] S. Becker, H. Koziolek, and R. Reussner. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82(1):3–22, 2009.

[4] K. Chan and I. Poernomo. QoS-aware model driven architecture through the UML and CIM. *Information Systems Frontiers*, 9(2-3):209–224, 2007.

[5] B. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, editors. *Software Engineering for Self-Adaptive Systems*, volume 5525 of *LNCS*. Springer, 2009.

[6] Distributed Management Task Force. Common Information Model (CIM) standard. `http://www.dmtf.org/standards/cim/`, May 2009.

[7] T. Focke, W. Hasselbring, M. Rohr, and J.-G. Schute. Ein Vorgehensmodell für Performance-Monitoring von Informationssystemlandschaften. *EMISA Forum*, 27(1):26–31, Jan. 2007.

[8] T. Focke, W. Hasselbring, M. Rohr, and J.-G. Schute. Instrumentierung zum Monitoring mittels Aspekt-orientierter Programmierung. In *Tagungsband Software Engineering 2007*, volume 106 of *LNI*, pages 55–59. Gesellschaft für Informatik, Bonner Köllen Verlag, Mar. 2007.

[9] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004.

[10] D. Garlan, S.-W. Cheng, and B. R. Schmerl. Increasing system dependability through architecture-based self-repair. In *Architecting Dependable Systems*, volume 2677 of *LNCS*, pages 61–89. Springer, 2003.

[11] J. Kephart and D. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, Jan. 2003.

[12] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the 2007 European Conference on Object-Oriented Programming (ECOOP '97)*, volume 1241 of *LNCS*, pages 220–242. Springer, 1997.

[13] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990.

[14] N. S. Marwede, M. Rohr, A. van Hoorn, and W. Hasselbring. Automatic failure diagnosis in distributed large-scale software systems based on timing behavior anomaly correlation. In *Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR 2009)*, pages 47–57. IEEE, Mar. 2009.

[15] J. Matevska-Meyer, S. Olliges, and W. Hasselbring. Runtime reconfiguration of J2EE applications. In *Proceedings of the French Conference on Software Deployment and (Re) Configuration (DECOR'04)*, pages 77–84. University of Grenoble, France, Oct. 2004.

[16] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.

[17] C. Momm, T. Detsch, and S. Abeck. Model-driven instrumentation for monitoring the quality of web service compositions. In *Proceedings of the 2008 12th Enterprise Distributed Object Computing Conference Workshops (EDOCW '08)*, pages 58–67, Washington, DC, USA, 2008. IEEE Computer Society.

[18] OMG. UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms. `http://www.omg.org/spec/QFTP/`, Apr. 2008.

[19] M. Rohr, S. Giesecke, W. Hasselbring, M. Hiel, W.-J. van den Heuvel, and H. Weigand. A classification scheme for self-adaptation research. In *Proceedings of the International Conference on Self-Organization and Autonomous Systems in Computing and Communications (SOAS'2006)*, Sept. 2006.

[20] M. Rohr, A. van Hoorn, J. Matevska, N. Sommer, L. Stöver, S. Giesecke, and W. Hasselbring. Kieker: Continuous monitoring and on demand visualization of Java software behavior. In *Proceedings of the IASTED International Conference on Software Engineering 2008 (SE 2008)*, pages 80–85. ACTA Press, Feb. 2008.

[21] T. Stahl and M. Völter. *Model-Driven Software Development – Technology, Engineering, Management*. Wiley & Sons, 2006.

[22] C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 2. edition, 2002.

[23] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software Architecture: Foundations, Theory and Practice*. John Wiley & Sons, Inc., 2009.

[24] A. van Hoorn, M. Rohr, A. Gul, and W. Hasselbring. An adaptation framework enabling resource-efficient operation of software systems. In N. Medvidovic and T. Tamai, editors, *Proceedings of the 2nd Warm-Up Workshop for ACM/IEEE ICSE 2010 (WUP '09)*, pages 41–44. ACM, Apr. 2009.

Support for Evolution of Software Systems using Embedded Models

Michael Goedicke, Michael Striewe, Moritz Balz
Specification of Software Systems
Institute of Computer Science and Business Information Systems
University of Duisburg-Essen, Campus Essen
Essen, Germany
{michael.goedicke, michael.striewe, moritz.balz}@s3.uni-due.de

Abstract: In this paper we show a new approach to evolution of software systems. We embed high-level specification information into program code patterns, so that such program code is interpretable at different abstraction levels. Since these model information is also acessed at run time for execution, we can avoid the situation that program code and high-level specifications are out of synch. Since the program code is thus a valid notation for the model syntax, we can apply transformations based on model semantics to it. An example will be provided that transforms software based on state machines to process models. This leads to a new perspective of software evolution in which the program code can be considered at higher levels of abstraction.

1 Introduction

In many cases software systems are in use longer than their developers anticipate. This implies especially for larger software systems that essential structural information regarding the overall software architecture and the way operations inside the system actually work is needed beyond the initial development of the system. Usually this kind of information is called documentation and is notoriously out of synch with the actual system after some development time.

Model-driven software development (MDSD) approaches have been advocated to remedy some part of this problem which means that models at a higher level of abstraction containing less detail are used to design a system and generate the actual running version by means of a sequence of transformations. Such transformations take care of mapping the high level concepts to low level details at the programming language level in order to obtain an efficiently running software system. Such transformations can be executed during development time or during run time which means direct execution of the specified models.

Such approaches do not take into account the issues of evolution over a long period of time. Software systems being in use and in constant development over decades is by now the common case while green field developments are very rare these days. Such a long development time requires tight synchronisation between the abstract models on the one

side and the code which is actually executed on the other side. This is due to the fact that the abstract models contain the main part of the essential information regarding system structure and system operation. This abstract specification information contained in the models experiences less change and is thus more stable than the actual program code which has to keep up with changing hardware and software platforms. To make this issue of volatility even worse, hand coded pieces are necessary to overcome some limitations of the automatic transformations. The inevitable evolution of the software system requires then a careful and difficult evolution of the transformations and re-coding of the hand coded pieces and fine tuning related additions.

A common observation is that although all stake-holders in the development process know about these interdependencies between the various development stages, at the end of the day the only piece of information being up to date and available is the program code.

Given these circumstances one would expect tools which help the developers to infer the abstract information of a software system from the actual code. In fact, many approaches [Ant07, Mef06, Mik98, EBM07, SD05, Nec02] pursue this goal. However, it is very difficult to distinguish automatically between program code which represents essential statements regarding business logic and system structure on the whole and code which exists only due to some complicated requirements of some frameworks or libraries which were used or had to be used in the code. So the result of these tools comes in many cases attached with a degree of precision of less than 100 percent. This is not satisfying especially in large software systems since additional analysis has to be done in order to assess the outcome of changes to the source code or models (in case of MDSD).

The need to support long living evolving software systems thus requires a tight integration of program code with its specification (the model) at an abstract level. An additional benefit would be that the specification is contained in the program code in such a way that the specification can be extracted and interpreted during runtime. Such a feature would allow to analyse many important aspects during runtime. This is especially useful in systems which need to run non-stop 24x7. An important aspect, for example, is the set of dependencies a system has on other systems. Modern software systems come with mechanisms like plug-ins which allow to add additional functionality at deployment or even during runtime of the system. Thus the analysis of the interdependencies in a whole landscape of software systems can only be done at a specific point in time regarding an actual set of running systems containing all plug-ins etc.

If the model and the code is tightly intertwined a specific step of evolving a software system entails various checks and (mostly local) transformations which ensure the desired integration of program code and its model. Of course, this also means that not arbitrary program code structures are possible. Such a concept has been described in our earlier work [BSG08, BG09]. Here we show an additional benefit of our approach. This is illustrated by the problem to transform the model which belongs to a specific class of models into an equivalent model of another model class. Of course, this is only possible for compatible model classes e.g. those specifying actions. Here we consider the class of state machine models and process models. This is useful when there is a need to create a new view on the system which has been recognised as an important need in the research field of views and viewpoints in software engineering.

Below we describe our approach and illustrate it with the example transformation from state machine models to process models. In section 2 we describe our general approach which is not limited to models specifying actions. However, the specific approach using state machines and process models is explained as well. In section 3 we show how the actual program code based on a state machine model is transformed into a corresponding program code based on an equivalent process model derived by a small set of transformation rules. This presentation is complemented with a brief discussion of related work in section 4 and discusses pros and cons in the concluding section 5.

2 Embedded Models

Model-driven software development approaches have the objective of relating high-level models to executable systems. Thus, when high-level models are transformed between different notations, a mechanism must exist that derives algorithmic program code from these models. This development step is unidirectional since models cannot be unambiguously extracted from arbitrary program code again. While working with different abstraction levels is desirable for model-driven software development, this unidirectional step constitutes a break of the principle of using model transformations at a higher level of abstraction for software development.

2.1 Concept

We earlier proposed the concept of *embedded models* [BSG09] to overcome these problems. The basic idea of embedded models is to define program code patterns in object-oriented general-purpose programming languages that represent the abstract syntax of high-level models. The program code is thus interpretable at different levels of abstraction, that of the programming language itself and that of the formal model. For this purpose, only static object-oriented structures are used that are not only available at development time, but also at run time. We make especially use of the ability of modern programming languages to decorate object-oriented structures with type-safe meta data [Sch04], thus adding information to program code fragments relating them to high-level models.

Thus it is possible to consider the program code at development time with respect to a formal model. At run time, the same program code is executed by small frameworks that access the program code fragments by means of structural reflection [DM95]. According to the model semantics, acting on the static program code structures creates sequences of actions. Since the program code pattern is part of arbitrary program code, well-defined interfaces to the code outside the pattern must be defined to access the application's state and to invoke business logic. The complete definition of an embedded model thus consists of the following:

- A precise formal model definition.

- A program code pattern that is formed after the abstract syntax of this formal model.
- An appropriate execution semantics.
- Interfaces to arbitrary program code that are interpretable at the level of the model semantics and also provide an appropriate functionality at run time.

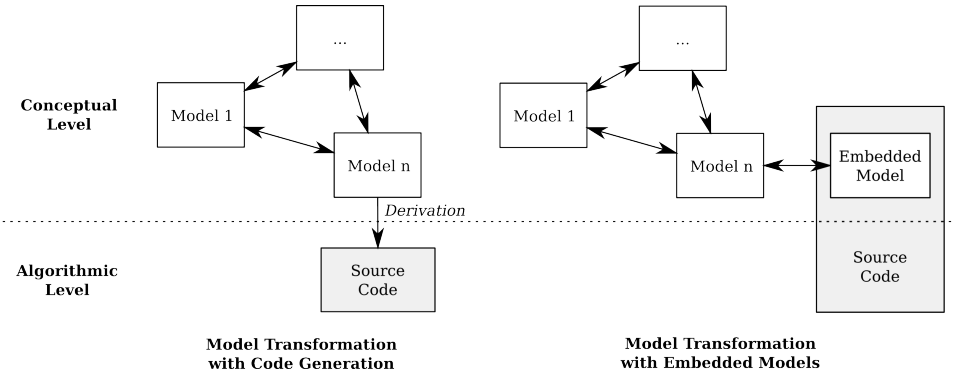


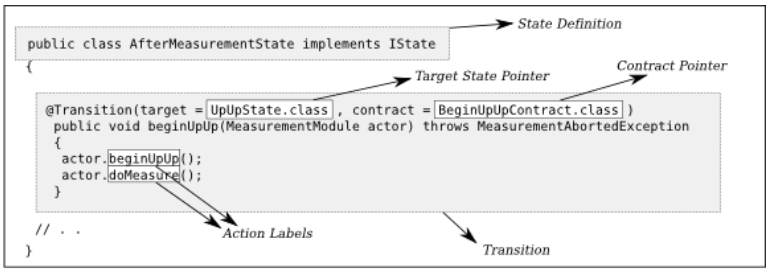
Figure 1: Model transformations with embedded models: The chain of possible bidirectional transformations does not terminate with program code generation. Instead, program code can carry different abstraction levels, thus making it another notation for embedded models and part of the related transformations .

With such embedded model definitions, the program is no longer unidirectionally derived from models. Instead, it is another notation carrying the semantics of the formal model, but includes the possibility to be executed by an appropriate framework while at the same time being integrated in arbitrary applications. Considering model transformations, this means that the program code can be fully integrated: When an embedded model exists, the related code can be source or target of a model transformation. This makes transformations a much more powerful tool for model-driven software development: Models can not only be transformed for communication and design purposes, but also to create *and* re-engineer executable systems. The principle of this approach is sketched in figure 1.

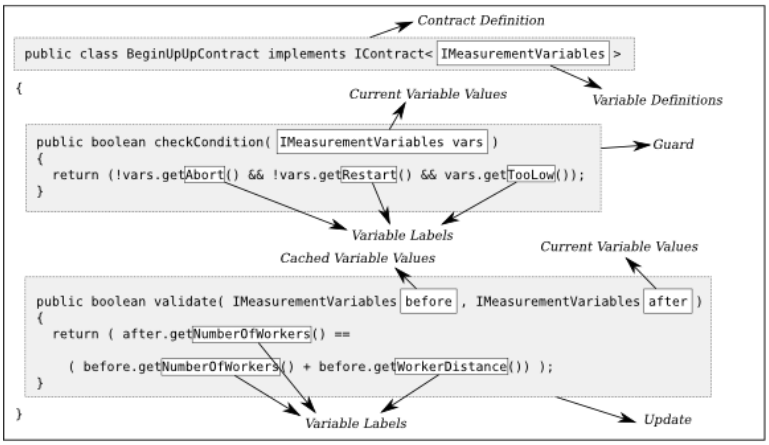
2.2 Example

An example for such a program code pattern is shown in figure 2. The program code fragments represent a part of a state machine model we described in previous publications [BSG08]. The language chosen for this implementation is Java [GJSB05] with its annotations enhancement for meta data inclusion [Sun04].

The class at the top represents a state where the class name equals the name of the state. The method in the state class represents a transition. It is decorated with meta data referring to the target state class and a “contract” class containing guards and updates. An



State and Transition Definition in Source Code



Contract Definition in Source Code

Figure 2: A state definition with an outgoing transitions and its contract. The first method of the contract checks a pre-condition with respect to the current variable values, while the second method checks a post-condition and may thus compare the current values to the previous values.

interface type referred to as "actor" is passed to transition methods. Its methods are interpreted as action labels which can be called when the transition fires.

Guards and updates are implemented as two methods in a "contract" class which is shown at the bottom of the figure. Both evaluate boolean expressions which serve as guards. These guards use the current variable values of the state machine to determine if a transition can fire, the update compares the current values with the values from the point in time before the transition fired to determine the changes to the state space. For this purpose both methods access a "variables" type which is a facade type representing the variables constituting the state space of the state machine. This type contains "get" methods for each variable, which are by this means defined with a label and a data type.

The execution framework can access the classes, methods and annotations by means of reflection. It invokes guards and determines a transition that can fire. After the transition method has been invoked, the update is called and the next state is reached. For this embedded model we already developed a transformation into the input language of the state machine model checker UPPAAL [LPY97] which enabled graphical design and verification of the model.

The complete example state machine consists of 10 states, 27 transitions and 8 variables. It belongs to a load generator application for performance tests and entails a user interface, networking functionality for remote controlled load generation and statistical evaluation of measurement results. These issues are hard to express in models, so that a complete model-driven development of the system was not feasible. However, the core of the load generation process is the strategy used to generate load. It can be modeled as an embedded state machine because it has a well-defined behavior, works with a limited set of variables and initiates the execution of business logic depending on the current state.

3 Program Code Evolution by Model Transformation

In this section we are going to discuss a concrete example on how software evolution can be supported by embedded models. In this example the core behaviour of a software system is designed as a finite state machine. At the level of models this state machine can be transformed into a process model automatically while losing only just a few features of state machines that cannot be expressed in process models. Such an automatic transformation is only possible if the program code adheres to the rules and complies with the model semantics. Thus, embedded models are needed, which define program code structures that are unambiguous.

3.1 Sketch of Concept

Program code can be expressed as its syntax tree generated by a parser. This tree can be enriched by additional semantic information (e.g. edges denoting references) and this way be extended to an attributed graph. Since embedded models rely on static structures, these parts of a model are reflected by the generated graph. Thus model transformation rules that are able to transform a state machine model into a process model or vice versa can be rewritten in order to transform program code with an embedded state machine into program code with an embedded process model or vice versa respectively. From state machines to process models, this transformation consists of several steps:

- All states of the state machine have to be converted to decision nodes in the process model.
- All transitions in the state machine have to be converted to activity nodes properly connected to decision nodes in the process model.
- Each activity node that contains more than one action label has to be split up into a sequence of activity nodes. This step can be performed here or at any later point in time.
- Each decision node having exactly one incoming and one outgoing transition can be discarded, connecting the nodes of the incoming and outgoing transition directly.

- Each decision node without incoming transitions is changed to a start node.
- Each decision node without outgoing transitions is changed to an end node.
- Each descision node with multiple incoming transitions and only one outgoing transition is changed to a merge node.

While this list of steps applies to the model transformation itself, using embedded models requires additional steps because formal aspects of program code (e.g. import statements) have to be taken into account. Note that special concepts like state machines communicating over channels, that have to be expressed by parallel and joining processes, are not considered here in order to keep this example short.

All rules can be implemented as graph transformation rules acting on the graph generated by parsing program code. All changes can be written back as local changes without overriding program code statements that are not part of the model. For example, methods can be moved, copied or renamed and even modified by adding or removing annotations or parameters without touching the body of the method. However, if the transformation requires additional code, new source files can be generated.

3.2 Graph Transformation Rules

The actual set of graph transformation rules used to implement the concept sketched above consists of 21 rules. The implementation has been done using AGG 1.6.4 [AGG] as graph transformation engine with GGX-Toolbox [GGX] for parsing and rewriting Java files. The algorithmic steps listed in the previous section could be implemented by transformation rules straight forward. At first, two rules are concerned with converting states to decision nodes and transitions to activity nodes, implementing the first two steps. After that, a set of six minor rules do some necessary housekeeping to the graph like reordering imports or removing unnecessary annotations. The next two rules remove source code not longer needed and useless decision nodes according to the fourth step of the algorithm. A set of three simple rules is the next to be executed, implementing the last three steps of the algorithm. Afterwards only one major rule is left for splitting up activitiy nodes, which was deferred until here. Another set of seven rules is finally concerned with some adjustments to the code.

One of the most important rules – changing states to process nodes and creating activity nodes – is shown in figure 3 in a simplified manner. Due to the use of embedded models, elements to be moved can easily be identified by their annotations on the left hand side of the rule and thus reassembled on the right hand side. Similarities between state machines and process models allow to reuse larger parts of existing program code, e.g. complete method bodies.

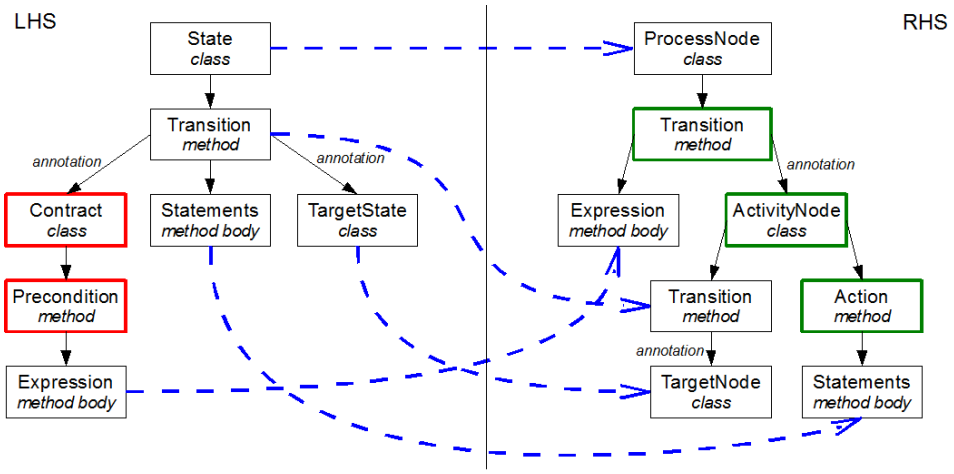


Figure 3: Simplified graph transformation rule for transforming states into process nodes. Nodes deleted from the syntax graph are marked in red while newly created nodes are marked in green. Some of the preserved nodes are renamed during transformation. Note, how contents from the original state node are moved to a newly created activity node, while contents from the original extra contract node are moved to the existing process node.

3.3 Generalization of the Approach

In order to gain reusability these rules can be grouped into five different categories. The main criterion is whether they are concerend with elements of the embedded model before transformation (source model), elements of the embedded model after transformation (target model), arbitrary source code or graph elements that are not relevant for the syntax of program code.

The first category contains rules that act both on elements of the source model as well as of the target model like the rule shown above. These rules can hardly be reused in a general approach since they are specific for a transformation between a specific pair of model types.

The second category contains rules that are responsible for deleting elements from the source model. They can possibly be reused if a transformation starts from this type of source model and has to delete these elements, independent from the target model. Similary, the third category contains rules that create elements from the target model without considering the source model. They can possibly be reused if a transformation has to create the same type of target model.

The fourth category contains rules that work on arbitrary source code elements that are not related to embedded models directly. They might be useful in general, even without working with embedded models. The same applies to the fifth category, containing rules that work on the structure of syntax graphs itself.

Besides rules from the first category, all other rules are reusable at least in some other transformations, thus the approach can easily be generalized. Building a larger library of rules would allow to define different transformations between embedded models, e.g. from process models back to state machine models, by combining rules in the right way.

4 Related Work

Approaches exist that try to relate program code to higher-level specifications. When these specification are formal models, round-trip engineering [SK04] concepts can be applied. Informal specifications can inferred from program code by detecting patterns [PSRN05, Shi07]. Although specifications can be extracted from program code based on design patterns [NWZ01, DLDvL08, WBHS07, NKG+07, Mef06, GSJ00, MCL04, MEB05], all of these approaches still require manual effort or are based on heuristics and are thus error-prone. Approaches to formalize design patterns [SH04, Mik98, MDE97] work at a lower level of abstraction and are not related to abstract specifications.

Approaches which consider the program code itself as a model [VHB+03, HJG08, Vol06], for example using model checking techniques, also work with program code semantics at a low level of abstraction and do not consider abstract specifications.

Integrating models in code has also been studied in [BM06]. Compared to our approach, the references to model elements are generic and not related to specific properties of formal models.

The main issue discussed in this presentation refers to model transformation. This area of research is very active and well explored. A good overview of bidirectional transformations can be found in [CFH+09]. In addition, the field of program transformation is addressing similar goals. In most cases program transformation is performed in form of refactorings which is a different case compared to our approach here: In refactoring approaches a program is (locally) restructured in order to avoid bad code smells. Here we transfer a given program which has a specific structure into an equivalent program with another specific structure.

5 Conclusion

In this paper we described an approach which reports on the benefits of tight integration between program code and its specification in form of a given class of models. The approach transforms a software system based on a state machine model into an equivalent program code again with a model based on the class of process models. This could be then the basis for further developments. The tool chain to support this kind of high level transformation builds on graph transformation and a graph representation of the programs which contains additional model information in form of annotations as available in Java. The actual set of transformation rules is surprisingly small. Of course, there are the usual

preparation and house keeping transformations which provide the necessary build up and cleaning process. The actual transformation is done by just a few transformation rules. Of course, it can be argued that this is due to the small conceptual distance between state machines and process models. But it is also clear from the discussion on views and viewpoints in the literature that each view is a legitimate and useful way in its own right to provide a specification of a system or a system part.

Tools have been implemented to support the actual transformations for multiple purposes, as we described in our previous publications. The objective to recover specifications unambiguously can be used to transform between models as presented in this contribution, but also for visual design and verification. Since embedded models are based on static program code structures that are accessible by means of reflection at run time, they can also be extracted from running systems and transformed into abstract representations, for example for monitoring.

If the approach is applied to model classes being more apart than the two classes used in this paper we envisage still a great part of the transformation process being supported automatically. The parts which have to be supported manually is very small and has clear interfaces to the rest of the system. Thus specific support can be created for these manual steps as well.

Support for additional model classes is currently in development. This will extend the approach to structural models of the systems and widens the support for more runtime related monitoring, refurbishment and evolution of software systems.

References

[AGG] AGG website. `http://tfs.cs.tu-berlin.de/agg/`.

[Ant07] Michal Antkiewicz. Round-trip engineering using framework-specific modeling languages. In *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 927–928, New York, NY, USA, 2007. ACM.

[BG09] Moritz Balz and Michael Goedicke. Embedding Process Models in Object-Oriented Program Code. In *Proceedings of the First Workshop on Behavioural Modelling in Model-Driven Architecture (BM-MDA)*, 2009.

[BM06] Thomas Büchner and Florian Matthes. Introspective Model-Driven Development. In *Software Architecture, Third European Workshop, EWSA 2006, Nantes, France, September 4-5, 2006*, volume 4344 of *Lecture Notes in Computer Science*, pages 33–49. Springer, 2006.

[BSG08] Moritz Balz, Michael Striewe, and Michael Goedicke. Embedding State Machine Models in Object-Oriented Source Code. In *Proceedings of the 3rd Workshop on Models@run.time at MODELS 2008*, pages 6–15, 2008.

[BSG09] Moritz Balz, Michael Striewe, and Michael Goedicke. Embedding Behavioral Models into Object-Oriented Source Code. In *Software Engineering 2009. Fachtagung des GI-Fachbereichs Softwaretechnik, 2.-6.3.2009 in Kaiserslautern*, 2009.

[CFH+09] Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger. Bidirectional Transformations: A Cross-Discipline Perspective—GRACE meeting notes, state of the art, and outlook. In *ICMT2009 - International Conference on Model Transformation, Proceedings*, LNCS. Springer, 2009. To appear.

[DLDvL08] Pierre Dupont, Bernard Lambeau, Christophe Damas, and Axel van Lamsweerde. The QSM Algorithm and its Application to Software Behavior Model Induction. *Applied Artificial Intelligence*, 22(1-2):77–115, 2008.

[DM95] François-Nicola Demers and Jacques Malenfant. Reflection in logic, functional and object-oriented programming: a short comparative study. In *In IJCAI '95 Workshop on Reflection and Metalevel Architectures and their Applications in AI*, pages 29–38, 1995.

[EBM07] Ghizlane El Boussaidi and Hafedh Mili. A model-driven framework for representing and applying design patterns. In *COMPSAC '07: Proceedings of the 31st Annual International Computer Software and Applications Conference*, pages 97–100, Washington, DC, USA, 2007. IEEE Computer Society.

[GGX] GGX-Toolbox website. `http://www.s3.uni-duisburg-essen.de/research/ggx-toolbox.html`.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java™ Language Specification, The 3rd Edition*. Addison-Wesley Professional, 2005.

[GSJ00] Alain Le Guennec, Gerson Sunyé, and Jean-Marc Jézéquel. Precise Modeling of Design Patterns. In *UML 2000 - The Unified Modeling Language, Advancing the Standard, Third International Conference, York, UK, October 2-6, 2000, Proceedings*, pages 482–496, 2000.

[HJG08] Gerard J. Holzmann, Rajeev Joshi, and Alex Groce. Model driven code checking. *Automated Software Engineering*, 15(3-4):283–297, 2008.

[LPY97] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, Oct 1997.

[MCL04] Jeffrey K. H. Mak, Clifford S. T. Choy, and Daniel P. K. Lun. Precise Modeling of Design Patterns in UML. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 252–261, Washington, DC, USA, 2004. IEEE Computer Society.

[MDE97] Theo Dirk Meijler, Serge Demeyer, and Robert Engel. Making Design Patterns Explicit in FACE. *ACM SIGSOFT Software Engineering Notes*, 22(6):94–110, 1997.

[MEB05] Hafedh Mili and Ghizlane El-Boussaidi. Representing and Applying Design Patterns: What Is the Problem? In Lionel C. Briand and Clay Williams, editors, *MoDELS*, volume 3713 of *Lecture Notes in Computer Science*, pages 186–200. Springer, 2005.

[Mef06] Klaus Meffert. Supporting Design Patterns with Annotations. In *ECBS '06: Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems*, pages 437–445, Washington, DC, USA, 2006. IEEE Computer Society.

[Mik98] Tommi Mikkonen. Formalizing Design Patterns. In *ICSE '98: Proceedings of the 20th international conference on Software engineering*, pages 115–124, Washington, DC, USA, 1998. IEEE Computer Society.

[Nec02] George C. Necula. Proof-Carrying Code. Design and Implementation. In *Proof and System Reliability*, pages 261–288, 2002.

[NKG+07] Oscar Nierstrasz, Markus Kobel, Tudor Girba, Michaele Lanza, and Horst Bunke. Example-Driven Reconstruction of Software Models. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR) 2007*, pages 275–286, 2007.

[NWZ01] Jörg Niere, Jörg P. Wadsack, and Albert Zündorf. Recovering UML Diagrams from Java Code using Patterns. In *Proceedings of the 2nd Workshop on Soft Computing Applied to Software Engineering, Enschede, The Netherlands (J.H. Jahnke and C. Ryan, eds.)*, 2001.

[PSRN05] Ilka Philippow, Detlef Streitferdt, Matthias Riebisch, and Sebastian Naumann. An approach for reverse engineering of design patterns. *Software and Systems Modeling*, 4(1):55–70, February 2005.

[Sch04] Don Schwarz. Peeking Inside the Box: Attribute-Oriented Programming with Java 1.5. *ONJava.com*, June 2004. `http://www.onjava.com/pub/a/onjava/2004/06/30/insidebox1.html`.

[SD05] Giovanna Di Marzo Serugendo and Michel Deriaz. Specification-Carrying Code for Self-Managed Systems. In *IFIP/IEEE International Workshop on Self-Managed Systems and Services*, 2005.

[SH04] Neelam Soundarajan and Jason O. Hallstrom. Responsibilities and Rewards: Specifying Design Patterns. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 666–675, Washington, DC, USA, 2004. IEEE Computer Society.

[Shi07] Nija Shi. *Reverse Engineering of Design Patterns from Java Source Code*. PhD thesis, University of California, Davis, 2007.

[SK04] Shane Sendall and Jochen Küster. Taming Model Round-Trip Engineering. In *Proceedings of Workshop on Best Practices for Model-Driven Software Development*, 2004.

[Sun04] Sun Microsystems, Inc. JSR 175: A Metadata Facility for the Java™Programming Language, 2004. `http://jcp.org/en/jsr/detail?id=175`.

[VHB+03] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model Checking Programs. *Automated Software Engineering Journal*, 10(2), 2003.

[Vol06] Nic Volanschi. A Portable Compiler-Integrated Approach to Permanent Checking. In *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 103–112, Washington, DC, USA, 2006. IEEE Computer Society.

[WBHS07] Neil Walkinshaw, Kirill Bogdanov, Mike Holcombe, and Sarah Salahuddin. Reverse Engineering State Machines by Interactive Grammar Inference. In *WCRE '07: Proceedings of the 14th Working Conference on Reverse Engineering*, pages 209–218, Washington, DC, USA, 2007. IEEE Computer Society.

Repository-Dienste für die modellbasierte Entwicklung

Udo Kelter
Fachbereich Elektrotechnik und Informatik
Universität Siegen
kelter@informatik.uni-siegen.de

Abstract: Viele langlebige Systeme existieren in mehreren Varianten, die parallel weiterentwickelt werden müssen. Hierzu werden unterstützende Repository-Dienste benötigt, neben dem klassischen Mischen von Dokumenten auch Historienanalysen. Bei Systemen, die mit modellbasierten Methoden (weiter-) entwickelt werden, ergeben sich besondere Probleme, weil auch die Modelle mitversioniert und in die Analysen einbezogen werden müssen. Dieser Workshopbeitrag beschreibt dieses Problemfeld genauer und skizziert Lösungsansätze, die im Rahmen des SiDiff-Projekts entwickelt wurden.

1 Einführung

Große Systeme, die über viele Jahre existieren und während dieser Zeit immer wieder erweitert und umstrukturiert werden, werfen große Wartungsprobleme auf. Gewartet werden müssen typischerweise nicht nur die aktuelle Version des Systems, sondern auch ältere Releases, die noch bei Kunden im Einsatz sind. Neben historisch entstandenen Varianten können auch aus produktbezogenen Gründen Varianten entstehen. Im Endeffekt müssen mehrere Systemvarianten, die zwar erhebliche Gemeinsamkeiten, aber auch signifikante Unterschiede aufweisen können, parallel weiterentwickelt werden.

Ein heute stark favorisierter Ansatz zur Reduktion der Entwicklungs- und Wartungsaufwände ist die modellbasierte Systementwicklung [10], also die Generierung von Teilen des Systems aus Modellen. In diesem Kontext werden auch domänenspezifische Sprachen diskutiert, die meist eine vorhandene Modellierungssprache erweitern[1]. Insb. die adaptive Wartung infolge von neuen Versionen unterliegender Systeme (DBMS, BS u.a.) kann so erleichtert werden. Die eingesetzten Modelltypen hängen natürlich von den Merkmalen der Anwendungsdomäne ab. Häufig eingesetzt werden Datenmodelle, meist eine Variante der UML-Klassendiagramme. Für eingebettete Systeme werden vor allem Zustandsmodelle und Funktions- bzw. Aktivitätsmodelle eingesetzt. Die Anwendungsdomänen unterscheiden sich auch darin, wie große Teile eines Systems aus den Modellen generiert werden können: im Idealfall alles, oft müssen jedoch umfangreiche Teile manuell entwickelt werden.

[1] Die Frage, ob die notwendigen komplexen Generierungsframeworks mehr Wartungsprobleme schaffen als lösen, ist für dieses Papier nicht relevant. Festgehalten werden kann, daß domänenspezifische Sprachen die hier diskutierten technologische Herausforderungen eher vergrößern.

Die modellbasierte Systementwicklung hat die Konsequenz, daß die Modelle zum integralen Teil der Software werden. Die Modelle müssen immer völlig konsistent sein mit manuell entwickeltem Code, Konfigurationsdaten und sonstigen Ressourcen, die oft in XML-Dateien gespeichert werden. Alle zusammengehörigen Dokumente – hierzu gehören natürlich auch Testdaten, Dokumentationen und sonstige Begleitdokumente – müssen gemeinsam versioniert werden.

Die Versionierung von Software wird klassischerweise durch Repository-Systeme wie CVS oder SVN unterstützt. Diese sind allerdings für Texte ohne spezielle Struktur konzipiert worden und arbeiten mit strukturierten Dokumenten, namentlich Modellen, nicht zufriedenstellend. Im Kern sind die üblichen Repository-Dienste auch für Modelle erforderlich; Systeme, die dies leisten, bezeichnen wir i.f. als **Modell-Repositories**. Modell-Repositories müssen natürlich nicht nur Modelle, sondern auch beliebige andere Dokumente integriert mitverwalten können.

Wir konzentrieren uns i.f. auf solche Modell-Repositories, die Funktionen anbieten, die die Weiterentwicklung langlebiger Software mit langen Versionshistorien unterstützen.

2 Modell-Repositories

Hauptfunktionen von Modell-Repositories. Zur Unterstützung von Entwicklungsprozessen werden mit hoher Priorität folgende Dienste bzw. Funktionen von Repositories benötigt. Diese bauen auf Basisfunktionen zur Verwaltung von Versionsgraphen und anderer administrativer Daten auf. Man kann alle Funktionen in einem einzigen System realisieren oder einige in Form autarker Analysewerkzeuge, solche Implementierungsentscheidungen interessieren uns an dieser Stelle nicht.

1. das *Vergleichen und Mischen* von Modellen: In großen Projekten ist arbeitsteilige Entwicklung und das gemeinsame Bearbeiten von Dokumenten unvermeidlich. Das exklusive Sperren von Dokumenten führt zu praktischen Problemen, daher hat sich in der Praxis weitgehend die Strategie durchgesetzt, nicht zu sperren, sondern automatisch zu mischen (3-Wege-Mischen).
2. *Suchfunktionen*, die “gleiche” Modellelemente oder -Fragmente in unterschiedlichen Varianten finden: Wir unterstellen hier, daß bei langlebigen Systemen mehrere Releases bei Anwendern installiert sind. Wenn in irgendeinem der Releases ein kritischer Fehler gefunden wird, muß für alle anderen Releases geprüft werden, ob der Fehler dort auch auftritt. In einen Fehler sind in der Regel mehrere Modellelemente involviert. Dieses Modellfragment kann in anderen Releases identisch oder in modifizierter Form auftreten. In diesem Zusammenhang ist es auch wichtig zu wissen, in welcher Version – die nicht notwendig ein Release ist – der Fehler entstanden ist und in welchem Kontext die damaligen Änderungen standen. Generell ist es für das Verständnis des Systems oft hilfreich zu wissen, welche Teile warum und zusammen mit welchen anderen Teilen eingeführt wurden und wie diese Systemteile weiterentwickelt wurden.
3. *historienbasierte Modellanalysen*: Ein System, das oft geändert wird, degeneriert insofern, als die Qualität der Entwurfsentscheidungen immer suboptimaler wird. Im Endef-

fekt sinkt die Verstehbarkeit des Systems und damit auch die Wartbarkeit, ferner wird die Einschätzung, wie aufwendig bzw. risikoreich weitere Änderungen sind, immer schwieriger. Der Qualitätsverlust muß durch Maßnahmen zur Strukturverbesserung kompensiert werden. Für die Planung solcher Reengineering-Maßnahmen benötigt man Analysefunktionen, die kritischsten Defizite des Systems zu finden helfen. Derartige Analysefunktionen werden auch im Rahmen des Softwarequalitätsmanagements für die Bewertung von Entwicklungsprozessen, zur Vorbereitung von Audits etc. benötigt.

Ein Referenzmodell für Repository-Dienste. Die vorstehenden Funktionen werden im Prinzip direkt von Entwicklern genutzt. Intern weisen sie begriffliche Überschneidungen und Abhängigkeiten auf. Es liegt nahe, ein Repository-System in dementsprechende Subsysteme zu zerlegen. Im folgenden Referenzmodell werden die konzeptuellen Abhängigkeiten und mit den Konzepten direkt korrespondierende Dienste in Form von Schichten dargestellt:

Schicht 0: Dokumentverwaltung
Diese Schicht beinhaltet Dienste zur Speicherung von Dokumenten beliebigen Typs. Eingeschlossen ist die Verwaltung von Nachfolger-Beziehungen zwischen Revisionen, Benutzern und sonstigen administrativen Daten. Diese Schicht wird man in der Regel durch ein etabliertes Repository-Produkt realisieren.

Schicht 1: Differenz- und Ähnlichkeitsberechnung von Modellen
Dieser Schicht liegen Begriffsdefinitionen für die Ähnlichkeit von einzelnen Modellelementen bzw. Modellfragmenten zugrunde. Diese Definitionen gehen direkt in die Berechnung von Modelldifferenzen ein. Allerdings gilt bei manchen Ähnlichkeitsbegriffen auch die Umkehrung. Daher kann man die Berechnung von Ähnlichkeiten und Differenzen nicht trennen. Aus Dokumentdifferenzen kann man direkt Differenzmetriken ableiten, die typischerweise in der Differenz angegebene Korrespondenzen zwischen Modellelementen oder Änderungsoperationen zählen, ggf. gewichtet und/oder selektiert.

Im Gegensatz zu Schicht 0 ist diese Schicht *abhängig vom Modelltyp*, d.h. pro Modelltyp sind eigene Implementierungen oder zumindest Anpassungen erforderlich. Dies gilt auch für die aufbauenden Schichten.

Schicht 2a: Mischfunktionen
Eine Mischung basiert in der Regel auf einer vorher bestimmten Differenz, da die gemeinsamen Teile nur einmal in das Ergebnis übernommen werden. Die speziellen Teile der zu mischenden Dokumente können unverträglich sein. Zentrale Begriffe dieser Ebene sind daher Konflikte und Strategien zur Konfliktbehandlung.

Schicht 2b: Historienanalysen
Diese Schicht realisiert die oben erwähnten Suchfunktionen und historienbasierte Modellanalysen. Sie baut direkt auf Schicht 1 auf und liegt daher parallel zu den Mischfunktionen. Neben der Definition von Suchfunktionen sind hier Verfahren zur Vorverarbeitung und Indexierung von Versionshistorien angeordnet.

3 Technologische Herausforderungen

Dieser Abschnitt diskutiert den Stand der Technik in den vorstehende benannten Schichten des Referenzmodells. Auf Schicht 0 gehen wir nicht ein, denn hier ist etablierte Technologie verfügbar.

3.1 Differenzberechnung

Die entscheidende interne Funktion ist hier Bestimmung korrespondierender Dokumentteile. Für textuelle Dokumente sind hinreichend gute und effiziente Algorithmen bekannt, für graphstrukturierte Modelle ist das Problem deutlich komplizierter. Persistente Darstellungen sind keine geeignete Basis, sondern nur abstrakte Syntaxbäume. Besonders schwierig ist die Bestimmung von Korrespondenzen bei Modelltypen, in denen wichtige Modellelemente keine markanten lokalen Eigenschaften haben, sondern deren Nachbarschaft entscheidend ist. Ein zusätzliches Problem bei langlebigen Systemen sind neue Sprachversionen, die zu veränderten Metamodellen führen.

Der in [6] publizierte Vergleich von Algorithmen zeigt die Spannbreite der aktuell bekannten Lösungen und die jeweiligen Kompromisse auf:

- Verbreitet sind Verfahren, die auf Basis persistenter Identifizierer von Modellelementen arbeiten. Sie sind sehr effizient und leicht implementierbar, basieren aber auf sehr einschränkenden Voraussetzungen und Annahmen, die bei langlebigen, großen Systemen praktisch nicht erfüllbar sind. Sie bieten wenig bzw. keine Unterstützung für die Konfliktbehandlung und ähnlichkeitsbasierte Suchverfahren, weil die Semantik der Dokumenttypen nicht bekannt ist.
- Auf der anderen Seite stehen dedizierte Algorithmen für einzelne Sprachen bzw. Dokumenttypen, die besonders hochwertige Vergleichs- bzw. Mischergebnisse liefern, dafür aber relativ ineffizient sind und einen sehr hohen (um nicht zu sagen prohibitiven) Implementierungsaufwand verursachen, weil sie für jeden Modelltyp weitgehend neu entwickelt werden müssen.
- Frameworks wie das System SiDiff [5] zielen mittels einer “Sprache” zur Spezifikation von Ähnlichkeiten auf einen tragfähigen Kompromiß zwischen Laufzeiteffizienz, Qualität der Ergebnisse und Implementierungsaufwand der Algorithmen.

Qualitativ gute Algorithmen sind aktuell nur verfügbar für Klassendiagramme (Datenmodelle) in diversen Varianten, insb. für reverse engineerte Java-Quellprogramme, ferner für einfachere Varianten von Aktivitätsdiagramme und Zustandsautomaten. Für andere Modelltypen und domänenspezifischen Sprachen ist wenig oder nichts verfügbar. Weiterer Forschungs- und Entwickungsbedarf besteht hinsichtlich der Gestaltung der Metamodelle, der Qualitätsbeurteilung bzw. Optimierung von Differenzen und der Evolution der Metamodelle.

3.2 Mischfunktionen

Beim Mischen von Dokumenten können Fehler entstehen, und zwar hinsichtlich kontextfreier bzw. kontextsensitiver Syntax, Programmierstil und Semantik. Paare von Modellteilen (bzw. die sie erzeugenden Änderungen), die solche Fehler erzeugen, stehen in Konflikt zueinander. Mischfunktionen haben daher zwei wesentliche Teilfunktionen, nämlich Konflikterkennung und Konfliktbehandlung, also Mischentscheidungen. Beide Teilfunktionen hängen von der Semantik des Modelltyps ab und sind schwieriger zu realisieren, wenn

- Modelle dieses Typs komplexe Konsistenzkriterien aufweisen und Modelleditoren nur Modelle verarbeiten können, die einen hohen Korrektheitsgrad einhalten;
- eventuelle falsch positive Mischentscheidungen von nachfolgenden Entwicklungsschritte nicht oder nur mit hohem Aufwand erkannt werden.

Der Stand der Technik kann hier als rudimentär bezeichnet werden. Für viele Modelltypen gibt es keine Mischwerkzeuge, viele vorhandene Mischwerkzeuge unterstützen nur das 2-Wege-Mischen auf Syntaxbaumsdarstellungen und bieten nur sehr wenig Unterstützung.

3.3 Suchfunktionen

Suchfunktionen verallgemeinern die paarweise Ähnlichkeit, die schon beim Vergleichen von zwei Modellen benötigt wurde, auf beliebige Revisions- und Varianten-Ketten oder allgemeine Sammlungen von Modellen. Die Existenz bzw. Qualität von Suchfunktionen hängt daher direkt von den Funktionen ab, die Ähnlichkeiten berechnen. Einzelne Lösungsansätze werden in [2, 13, 14] diskutiert, von einem flächendeckenden Angebot praxiserprobter Lösungen ist man aber noch weit entfernt.

3.4 Analysefunktionen

Hauptzweck dieser Funktionen ist, die Qualität eines Systems zu beurteilen und insb. Systemteile (also u.a. Modellfragmente) zu finden, an denen Strukturverbesserungen notwendig sind. Systemteile mit einer geringen Größe sind in diesem Sinne leichter handhabbar, weil man in vielen Fällen die Defekte formal beschreiben kann (z.B. zu große Klassen oder Zyklen in benutzt-Beziehungen); auf dieser Basis kann man Suchverfahren implementieren, die die entsprechenden Stellen finden. Ferner ist aufgrund der geringen Größe der Aufwand für die Reparatur gering, namentlich wenn sie durch Refactorings, ggf. in Verbindung mit Design-Patterns, z.T. automatisierbar ist bzw. durch Werkzeuge unterstützt wird.

Strukturverbesserungen in größeren Systemteilen werfen deutlich mehr Probleme auf: die Definition, wann ein Defekt vorliegt, ist nicht formalisierbar, und vielfach wird nicht alleine der Zustand einer Version zur Beurteilung herangezogen, sondern Merkmale der

Änderungshistorie, d.h. es wird vom Entwicklungsprozeß auf die Qualität des Produkts geschlossen. Beispielsweise sind Defizite in Systemteilen, in denen wiederholt größere Änderungen stattfanden, wahrscheinlicher. Das Auffinden von suspekten Systemteilen ist eher als ein Information-Retrieval-Problem anzusehen, bei dem es darum geht, unter den vielen möglichen Verbesserungsmaßnahmen diejenigen mit dem größten Nutzen und den geringsten Kosten herauszufinden. Wegen der höheren Umbaukosten können nämlich i.d.R. nur wenige derartige Maßnahmen durchgeführt werden.

Visualisierung von Historien. Die geforderten Analysen ganzer Versionshistorien stehen vor dem Problem der Informationsüberflutung: einzelne Versionen sind i.d.R. schon sehr umfangreich, das Datenvolumen steigt infolge der Versionen um 1 - 2 Größenordnungen.

Viele Analyseverfahren nutzen daher Metriken, einzelne Versionen werden also nicht mehr in allen Details dargestellt, sondern auf ihre Metrikwerte reduziert. “Auffällige” Versionen bzw. Vorkommnisse in der Versionshistorie können nur noch anhand der Metrikwerte bzw. der numerischen Differenzen der Metrikwerte erkannt werden.

Auf Metriken basieren auch fast alle Methoden zur Visualisierung von Historien. Die bekannten Methoden zur Visualisierung von großen naturwissenschaftlichen Datenmengen versagen aber bei Modellen weitgehend, weil hier die Grundstruktur des Datenraums nicht ein homogenes 3D-Gitter ist, sondern durch die wesentlich komplizierteren Strukturen in Modellen geprägt ist. Es sind diverse Vorschläge für 2- oder 3-dimensionale Visualisierungen von Versionshistorien gemacht worden, die teilweise auf einer 2-dimensionalen Darstellung einzelner Versionen basieren, u.a. die Evolution Matrix [7], die auf polymetrischen Sichten [8] basiert, Evo Spaces [16], die sich optisch an Stadtbilder anlehnen, Gevol [3], das Evolution Radar [1] und weitere Systeme.

Ein genereller Nachteil der vorstehenden Ansätze ist, daß die gewohnte graphische Darstellung der Modelle, in der die Systemstrukturen gut dargestellt werden, nicht mehr eingesetzt wird. Wegen der geometrischen Eigenschaften ist es sogar prinzipiell fraglich, ob man die gewohnten Darstellungsformen überhaupt für Historien einsetzen kann; sie stoßen bei großen Modellen ohnehin an ihre Grenzen und sind nicht für die Darstellung von Historien konzipiert worden. Veränderungen an den Strukturen eines Systems zählen indes zu den interessantesten Veränderungen.

4 Lösungsansätze zur Visualisierung von Modell-Historien

Dieser Abschnitt skizziert einige Lösungsansätze, die für die Visualisierung von Modell-Historien im Kontext des SiDiff-Projekts [11] entwickelt wurden.

4.1 Metriken von Differenzen statt Differenzen von Metriken

Übliche Metriken für Modelle sind Zählungen von Strukturelementen, z.B. die Zahl der Attribute einer Klasse in einem Klassendiagramm oder die Zahl der ausgehenden Transitionen eines Zustands in einem Zustandsmodell. Wenn nun die ein Attribut durch ein anderes ersetzt wird oder eine Transition durch eine andere, ändern sich die Metrikwerte nicht, obwohl signifikante Änderungen stattgefunden haben. Anders gesagt sind die numerischen Differenzen der Metrikwerte nur ein unzuverlässiger Indikator für den Umfang der Änderungen.

Die naheliegende Lösung besteht darin, zunächst eine vollständige (korrekte) Differenz [12] zwischen den Versionen zu berechnen. Aus dieser Differenz geht hervor, welche Editieroperationen zum Nachvollziehen der Veränderungen notwendig sind. Metriken, die sich auf Differenzen beziehen und z.B. die darin enthaltenen Operationen zählen, bezeichnen wir als **Differenzmetriken**. Differenzmetriken sind offensichtlich viel genauere Indikatoren für den Umfang der Änderungen als Differenzen von Metrikwerten.

Differenzmetriken haben den Vorteil, daß Typen von Änderungen hinsichtlich ihres Risikos kategorisiert werden können. Ferner können tabellarische oder graphische Darstellungen von Änderungshistorien einzelne Metriken isoliert darstellen (m.a.W. sollten Analysewerkzeuge dies erlauben).

Differenzmetriken und Werkzeuge, die diese unterstützen, müssen spezifisch für einzelne Modelltypen entwickelt werden, da jeder Modelltyp eigene Editieroperationen und Konsistenzkriterien hat. Das reine Graphiksystem eines Werkzeugs kann weitgehend unabhängig von den Modelltypen entwickelt werden (s. z.B. [4]), muß also nicht für jeden Modelltyp neu entwickelt werden. Das relevanteste Problem ist auch hier wieder die Differenzberechnung (vgl. Abschnitt 3.1).

4.2 3D-Darstellungen und Animation

3-dimensionale Darstellungen von Versionshistorien können grob eingeteilt werden in solche, die Strukturen der Modelle direkt anzeigen, auf dieser Basis natürlich auch Veränderungen der Strukturen, und andere, die i.w. nur Metrikwerte anzeigen.

Anzeige von Modellstrukturen. Ein Beispiel für die erste Kategorie ist der StructureChangesView im Werkzeug Evolver [4, 15], s. Bild 1. Diese Ansicht adressiert vor allem strukturelle Änderungen, z.B. wenn Klassen ihre Assoziationen zu anderen Klassen ändern. Die Darstellung besteht aus mehreren hintereinanderliegenden “Scheiben”, von denen jede eine Version darstellt. Jede Versionsdarstellung besteht aus Würfeln, die z.B. Klassen eines Klassendiagramms oder Zustände eines Zustandsdiagramms repräsentieren. Linien zwischen diesen Würfeln stellen Beziehungen, Transitionen o.ä. dar. Die Grunddarstellung kann mit diversen Metriken angereichert werden, z.B. kann je eine Metrik auf die Höhe, Breite und Farbe der Quader und die Dicke und Farbe der Linien abgebildet werden.

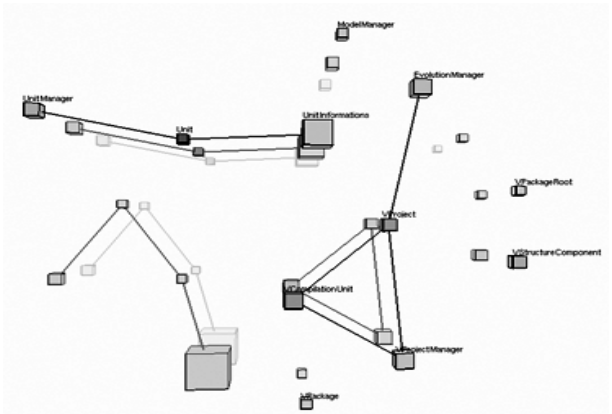


Abbildung 1: StructureChangesView im Werkzeug Evolver

Die Kameraposition kann beliebig zwar in Realzeit verändert werden (“Ego-Shooter”), normalerweise ist sie vor der vordersten Scheibe, so auch in Bild 1. Nur diese vorderste Version kann man also unbehindert erkennen, die dahinterliegenden Versionen werden von den davorliegenden teilweise verdeckt. Die vorneliegende Version ist die am meisten interessierende; um diese von den anderen optisch abzuheben, werden die hinteren Versionen immer transparenter gezeichnet. Die Knoten der Graphen werden vereinfacht dargestellt; würde man alle Details einer Klasse oder eines Zustands darstellen, wären diese nicht mehr lesbar.

Diese Darstellung ist gut geeignet, um bestimmte Typen von Änderungen zu erkennen, allerdings hat sie durchaus Limitationen:

– Die Zahl der angezeigten Entitäten muß klein sein. Für eine erste Gesamtübersicht über ein unbekanntes System ist diese Darstellung daher wenig geeignet, sie ist eher nach einer Einschränkung der Menge der zu untersuchenden Entitäten sinnvoll. Gute Möglichkeiten zur Selektion der angezeigten Entitäten sind daher sehr wichtig.
– Es kann nur eine beschränkte Zahl von Versionen sinnvoll angezeigt werden, ca. 5 Versionen sind noch gut erkennbar, ab ca. 10 Versionen wird die Darstellung trotz transparenter Darstellung unbrauchbar. Die Zahl der anzeigten Versionen sollte in Realzeit veränderbar sein, ebenso die vorne angezeigte Version. Wünschenswert ist ferner eine Funktion, die aus der gesamten Revisionskette besonders interessante Versionen anhand bestimmter Kriterien selektiert.

Metrikbasierte Darstellungen. Ein Beispiel für die zweite o.g. Kategorie ist der EvolutionView im Werkzeug Evolver, s. Bild 2. Diese zeigt für jede Version und jede Entität eine Säule. Die Höhe einer Säule ergibt sich anhand einer Metrik, angewandt auf diese Entität in dieser Version. Bei der Kameraposition, die in Bild 2 gewählt ist, verläuft die “Zeit” von hinten nach vorne. Von links nach rechts sind die angezeigten Entitäten angeordnet. In Bild 2 ist eine der Entitäten selektiert und alle zugeordneten Säulen über alle Versionen hinweg sind dunkler gezeichnet.

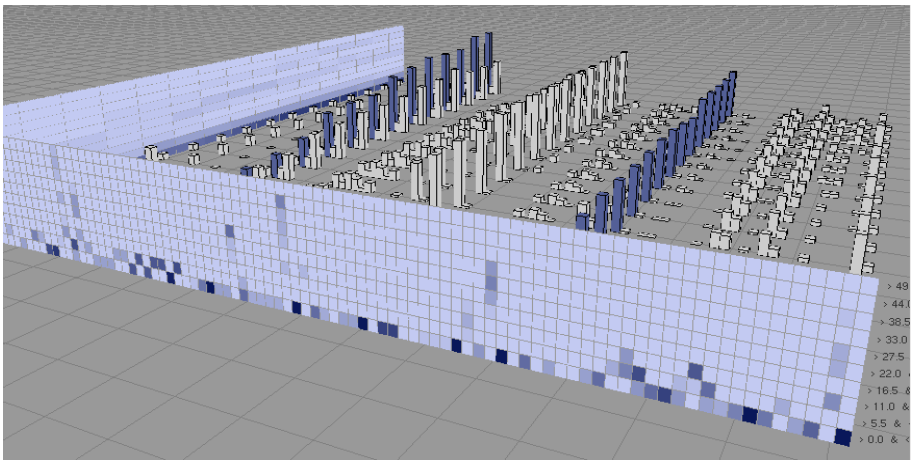


Abbildung 2: EvolutionView

An den Seiten befinden sind zusätzlich noch zwei Spektrographen: diese zeigen die Häufigkeitsverteilungen der Metrikwerte an. Hierzu wird der Wertebereich der Metrik in ca. 10 Intervalle eingeteilt, entsprechend viele kleine übereinanderliegende Rechtecke stehen auf einer Spektrographen-Wand zur Verfügung. Mit einer Farbcodierung wird die Zahl der Säulen, deren Größe im jeweiligen Intervall liegt, angezeigt. Die in Bild 2 vorne sichtbare Spektrographen-Wand zeigt die Häufigkeitsverteilungen der Metrikwerte *pro Entität* über die Systemversionen hinweg an, die seitlich sichtbare Wand die Häufigkeitsverteilungen *pro Systemversion*.

Animationen. Die beiden vorigen Darstellungsformen haben die zeitliche Reihenfolge, die durch eine Folge von Revisionen eines Systems entsteht, auf eine Dimension eines 3-dimensionalen graphischen Objekts abgebildet. Ein völlig anderer Ansatz besteht darin, die zeitliche Reihenfolge durch eine Animation darzustellen. Basis kann eine geeignete 2- oder 3-dimensionale Darstellung einzelner Versionen sein. Dies muß allerdings so gewählt sein, daß die Bewegungen gut erkennbar sind, also nicht zu geringfügig und nicht zu heftig sind.

Ein Beispiel findet sich im EvolutionView des Werkzeugs Evolver, s. Bild 3, das nur *ein* Bild einer Animation zeigt. Komplette Animationen können auf der WWW-Seite des Projekts [4] als Video angesehen werden. Entitäten werden hier durch Ellipsen dargestellt, die kreisförmig um einen Mittelpunkt angeordnet sind. Jeweils eine Metrik wird dargestellt durch (a) den Abstand der Ellipsen vom Mittelpunkt, (b) die Größe der Ellipse und (c) die Richtung der Längsachse der Ellipse. Änderungen in diesen Metriken führen zu entsprechenden mehr oder wenig schnellen Bewegungen der Ellipsen; die Fähigkeiten des menschlichen Sehsystems können hier besonders gut ausgenutzt werden.

Zusätzlich kann *eine* der Entitäten ausgewählt werden, deren Beziehungen zu anderen Entitäten dargestellt werden.

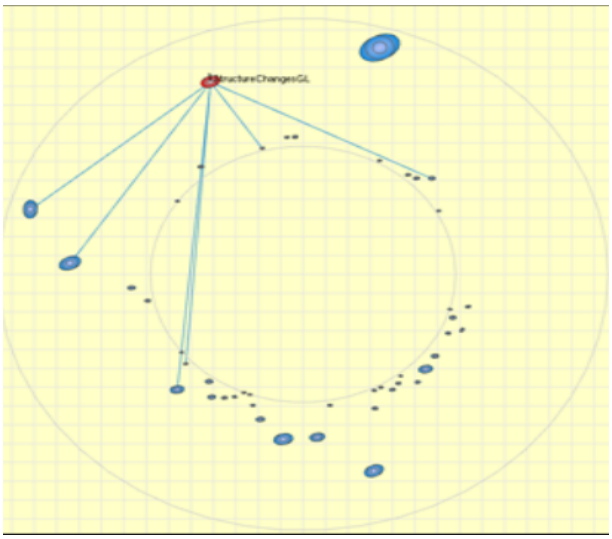


Abbildung 3: AnimationView

Integration und Evaluation der Darstellungsformen. Die vorstehenden Darstellungsformen haben jeweils eigene Stärken und Schwächen und müssen in einer integrierten Form verfügbar sein, in der man nahtlos zwischen diesen und weiteren Darstellungen, auch von Einzelversionen, wechseln kann.

Erste kontrollierte Evaluationen der oben vorgestellten Darstellungsformen zeigten, daß die EvolutionView aus Anwendersicht den größten Nutzen brachte, gefolgt von der AnimationView. Am schlechtesten schnitt die StructureChangesView ab.

5 Resümee

Die modellbasierte Systementwicklung erfordert für Modelle die gleichen Repository-Dienste, die man bei textuellen Dokumenten gewohnt ist. In der Praxis und hinsichtlich der technologischen Grundlagen ist man hiervon noch weit entfernt.

Das aus Sicht der Praxis drängendste Problem stellen Misch- und Vergleichswerkzeuge dar. Für viele Modelltypen sind keine brauchbaren Werkzeuge verfügbar, die verfügbaren Werkzeuge unterstützen nur das zeitaufwendige manuelle Mischen und bieten nur beschränkte Unterstützung bei der Konflikterkennung.

Analysefunktionen, die die Weiterentwicklung von Systemen mit vielen Revisionen bzw. Varianten unterstützen, existieren nur als Forschungsprototypen. Hier ist noch viel Raum für Verbesserungen vorhanden, sowohl bei der Entwicklung weiterer Darstellungsformen als auch bei der Integration verschiedener Darstellungsformen und Optimierung hinsichtlich der praktischen Nutzung.

Literatur

[1] d'Ambros, Marco; Lanza, Michele; Lungu, Mircea: Visualizing Integrated Logical Coupling Information; in: Proc. International Workshop on Mining Software Repositories 2006 (MSR 2006); ACM Press; 2006

[2] Bildhauer, Daniel; Horn, Tassilo; Ebert, Jürgen: Similarity-Driven Software Reuse; p.31-36 in: Proc. 2009 ICSE Workshop on Comparison and Versioning of Software Models; IEEE Catalog Number CFP0923G; 2009

[3] Collberg, Christian; Kobourov, Stephen; Nagra, Jasvir; Pitts, Jacob; Wampler, Kevin: A System For Graph-based Visualization of the Evolution of Software; p.77ff in: Proc. 2003 ACM Symposium on Software Visualization SoftVis'03; ACM; 2003

[4] Evolver: Analyzing Software Evolution with Animations and 3D-Visualizations (Project homepage); `http://pi.informatik.uni-siegen.de /projects/evolver`; 2009

[5] Kelter, Udo; Wehren, Jürgen; Niere, Jörg: A Generic Difference Algorithm for UML Models; p.105-116 in: Software Engineering 2005. Fachtagung des GI-Fachbereichs Softwaretechnik, 8.-11.3.2005, Essen; LNI 64, GI; 2005

[6] Kolovos, Dimitrios S.; Ruscio, Davide Di; Pierantonio, Alfonso; Paige, Richard F.: Different Models for Model Matching: An Analysis Of Approaches To Support Model Differencing; p.1-6 in: Proc. 2009 ICSE Workshop on Comparison and Versioning of Software Models; IEEE; 2009

[7] Lanza, M.: Recovering Software Evolution Using Software Visualization Techniques; p.37-42 in: Proc. 4th Intl. Workshop Principles Software Evolution IWPSE; ACM; 2001

[8] Lanza, Michele; Ducasse, Stéphane: Polymetric Views - A Lightweight Visual Approach To Reverse Engineering; IEEE Trans. Softw. Eng., 29:9, p.782ff; 2003

[9] Lungu, Mircea; Lanza, Michele: Softwarenaut: Exploring Hierarchical System Decompositions; p.351-354 in: Proc. Conference on Software Maintenance and Reengineering (CSMR '06), Washington, DC, USA, 2006; IEEE Computer Society; 2006

[10] Miller, Joaquin; Mukerji, Jishnu (eds.): MDA Guide Version 1.0.1; OMG, Document Number: omg/2003-06-01; 2003-06-12; `http://www.omg.org/docs/omg/03-06-01.pdf`

[11] SiDiff Differenzwerkzeuge; http://www.sidiff.org; 2008

[12] Treude, Christoph; Berlik, Stefan; Wenzel, Sven; Kelter, Udo: Difference Computation of Large Models; p.295-304 in: 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Sep 3 - 7, Dubrovnik, Croatia; 2007

[13] Wenzel, Sven; Kelter, Udo; Hutter, Hermann: Tracing Model Elements; p.104-113 in: 23rd IEEE International Conference on Software Maintenance (ICSM 2007), October 2-5, 2007, Paris, France; IEEE ; 2007

[14] Wenzel, Sven; Kelter, Udo: Analyzing Model Evolution; p.831-834 in: Proc. 30th International Conference on Software Engineering, Leipzig, Germany, May 1018, 2008 (ICSE'08); ACM Press; 2008

[15] Wenzel, Sven; Koch, Jens; Kelter, Udo; Kolb, Andreas: Evolution Analysis with Animated and 3D-Visualizations; p.475-478 in: Proc. 25th IEEE International Conference on Software Maintenance (ICSM 2009), 2009, Edmonton, Canada; IEEE; 2009

[16] Wettel, R.; Lanza, M.: Visual exploration of large-scale system evolution; p.219-228 in: Proc. 15th Working Conference on Reverse Engineering (WCRE), Washington DC, USA; IEEE Computer Society; 2008

[17] Wu, J.; Holt, J.; Hassan, A.: Exploring Software Evolution Using Spectrographs; p.80-89 in: Proc. Working Conference on Reverse Engineering (WCRE 2004); 2004

KAMP: Karlsruhe Architectural Maintainability Prediction

Johannes Stammel
stammel@fzi.de
Forschungszentrum Informatik (FZI), Karlsruhe, Germany

Ralf Reussner
reussner@kit.edu
Karlsruhe Institute of Technology (KIT), Germany

Abstract: In their lifetime software systems usually need to be adapted in order to fit in a changing environment or to cover new required functionality. The effort necessary for implementing changes is related to the maintainability of the software system. Therefore, maintainability is an important quality aspect of software systems.

Today Software Architecture plays an important role in achieving software quality goals. Therefore, it is useful to evaluate software architectures regarding their impact on the quality of the program. However, unlike other quality attributes, such as performance or reliability, there is relatively less work on the impact of the software architecture on maintainability in a quantitative manner. In particular, the cost of software evolution not only stems from software-development activities, such as re-implementation, but also from software management activities, such as re-deployment, upgrade installation, etc. Most metrics for software maintainability base on code of object-oriented designs, but not on architectures, and do not consider costs from software management activities. Likewise, existing current architectural maintainability evaluation techniques manually yield just qualitative (and often subjective) results and also do concentrate on software (re-)development costs.

In this paper, we present *KAMP*, the Karlsruhe Architectural Maintainability Prediction Method, a quantitative approach to evaluate the maintainability of software architectures. Our approach estimates the costs of change requests for a given architecture and takes into account re-implementation costs as well as re-deployment and upgrade activities. We combine several strengths of existing approaches. First, our method evaluates maintainability for concrete change requests and makes use of explicit architecture models. Second, it estimates change efforts using semi-automatic derivation of work plans, bottom-up effort estimation, and guidance in investigation of estimation supports (e.g. design and code properties, team organization, development environment, and other influence factors).

1 Introduction

During its life cycle a software system needs to be frequently adapted in order to fit in its changing environment. The costs of maintenance due to system adaptations can be very high. Therefore, system architects need to ensure that frequent changes can be done as easy as possible and as proper. This quality attribute of software systems is commonly known as adaptability placed within the wider topic of maintainability.

With respect to [ISO90] we define maintainability as *"The capability of a software product to be modified. Modifications may include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications"*. This *capability of being modified* in our point of view can be quantified by means of the effort it takes to implement changes in a software system. If changes can be done with less effort for one software system alternative than for another it indicates that the maintainability for the first one is better.

An important means for making software engineering decisions is the software architecture, which according to [IEE07] represents *"the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution."* It is commonly agreed, that the quality of a software system highly depends on its software architecture. One specific benefit of architectures is that they provide documentation for different activities, also beyond software implementation, such as software management or cost-estimation [PB01]. Software architects nowadays specify software architectures using formal architecture models, like [BKR07]. Ideally, it should already be possible to predict the resulting system's quality on basis of these models. The Palladio [BKR07] approach for instance allows an early performance prediction for component-based software architectures. In the same way, maintainability is highly influenced by the architecture and should be supported within an early design-time prediction. This observation is already considered by qualitative architecture evaluation techniques, such as ATAM or SAAM [CKK05]. However, these approaches mostly produce very subjective results as they offer little guidance through tool support and highly rely on the instructors experience [BLBvV04], [BB99]. Moreover, they only marginally respect follow-up costs, e. g. due to system deployment properties, team organization properties, development environment and generally lack meaningful quantitative metrics for characterizing the effort for implementing changes.

This paper presents as a contribution *KAMP*, the Karlsruhe Architectural Maintainability Method, a novel method for the quantitative evaluation of software architectures based on formal architecture models. One specific benefit of KAMP is that it takes into account in a unified manner costs of software development (re-design, re-implementation) and software management (re-deployment, upgrade installation). Moreover, KAMP combines a top-down phase where change requests are decomposed into several change tasks with a bottom-up phase where the effort of performing these change tasks is estimated. The top-down phase bases on architectural analyses, while the second phase utilises existing bottom-up cost estimation techniques [PB01]. Our approach evaluates the maintainability of an architecture by estimating the effort required to perform certain change requests for a given architecture. By doing this it also differs from classical cost estimation approaches which do not systematically reflect architectural information for cost estimation. The approach is limited to adaptive and perfective maintainability. This includes modifications or adaptations of a system in order to meet changes in environment, requirements and functional specifications, but not the corrections of defects, as also mentioned by the definition of maintainability we previously gave. *Paper Structure:* In Section 2 we summarize related work. In Section 3 we introduce the Karlsruhe Architectural Maintainability Prediction approach. Section 4 gives conclusions and points out future work.

2 Related Work

2.1 Scenario-Based Architecture Quality Analysis

In literature there are several approaches which analyse quality of software systems based on software architectures. In the following paragraphs we discuss approaches which make explicitly use of scenarios. There are already two survey papers ([BZJ04], [Dob02]) which summarize and compare existing architecture evaluation methods.

Software Architecture Analysis Method (SAAM) [CKK05] SAAM was developed in 1994 by Rick Kazman, Len Bass, Mike Webb and Gregory Abowd at the SEI as one of the first methods to evaluate software architectures regarding their changeability (as well as to related quality properties, such as extendibility, portability and reusability). It uses an informally described architecture (mainly the structural view) and starts with gathering change scenarios. Then via different steps, it is tried to find interrelated scenarios, i.e., change scenarios where the intersection of the respective sets of affected components is not empty. The components affected by several interrelated scenarios are considered to by critical and deserve attention. For each change scenario, its costs are estimated. The outcome of SAAM are classified change scenarios and a possibly revised architecture with less critical components.

The Architecture Trade-Off Analysis Method (ATAM) [CKK05] ATAM was developed by a similar group for people from the SEI taking into account the experiences with SAAM. In particular, one wanted to overcome SAAM's limitation of considering only one quality attribute, namely, changeability. Much more, one realised that most quality attributes are in many architectures related, i.e., changing one quality attribute impacts other quality attributes. Therefore, the ATAM tries to identify trade-offs between different quality attributes. It also expands the SAAM by giving more guidance in finding change scenarios. After these are identified, each quality attribute is firstly analysed in isolation. Than, different to SAAM, architectural decisions are identified and the effect (sensitivity) of the design decisions on each quality attribute is tried to be predicted. By this "sensitivity analysis" one systematically tries to find related quality attributes and trade-offs are made explicit. While the ATAM provides more guidance as SAAM, still tool support is lacking due to informal architectural descriptions and the influence of the personal experience is high. (Therefore, more modern approaches try to lower the personal influence, e.g., POSAAM [dCP08].) Different to our approach, change effort is not measured as costs on ATAM.

The Architecture-Level Prediction of Software Maintenance (ALPSM) [BB99] ALPSM is a method that solely focuses on predicting software maintainability of a software system based on its architecture. The method starts with the definition of a representative set of change scenarios for the different maintenance categories (e.g. correct faults or adapt to changed environment), which afterwards are weighted according to the likelihood of occurrence during the systems's lifetime. Then for each scenario, the impact of implementing it within the architecture is evaluated based on component size estimations (called scenario scripting). Using this information, the method finally allows to predict

the overall maintenance effort by calculating a weighted average of the effort for each change scenario. As a main advantage compared to SAAM and ATAM the authors point out that ALPSM neither requires a final architecture nor involves all stakeholders. Thus, it requires less resources and time and can be used by software architects only to repeatedly evaluate maintainability. However, the method still heavily depends on the expertise of the software architects and provides little guidance through tool support or automation. Moreover, ALPSM only proposes a very coarse approach for quantifying the effort based on simple component size measures like LOC.

The Architecture-Level Modifiability Analysis (ALMA) [BLBvV04]
The ALMA method represents a scenario-based software architecture analysis technique specialized on modifiability and was created as a combination of the ALPSM approach [ALPSM] with [Lassing1999a]. Regarding the required steps, ALMA to a large extend corresponds to the ALPSM approach, but features two major advantages. First, ALMA supports multiple analysis goals for architecture-level modifiability prediction, namely maintenance effort prediction, risk estimation and comparison of architecture alternatives. Second, the effort or risk estimation for single change scenarios is more elaborated as it explicitly considers ripple effects by taking into account the responsible architects' or developers' expert knowledge (bottom up estimation technique). Regarding effort metrics, ALMA principally allows for the definition of arbitrary quantitative or qualitative metrics, but the paper itself mainly focuses on lines of code (LOC) for expressing component size and complexity of modification (LOC/month). Moreover, the approach as presented in the paper so far only focuses on modifications relating to software development activities (like component (re-)implementation), but does not take into account software management activities, such as re-deployment, upgrade installation, etc.

2.2 Change Effort Estimation

Top-Down Effort Estimation Approaches in this section estimate efforts in top-down manor. Although they are intended for forward engineering development projects, one could also assume their potential applicability in evolution projects. Starting from the requirement level, estimates about code size are made. Code size is then related somehow to time effort. There are two prominent representatives of top-down estimation techniques: *Function Point Analysis (FPA)* [IFP99] and *Comprehensive Cost Model (COCOMO) II* [Boe00]. COCOMO-II contains three approaches for cost estimation, one to be used during the requirement stage, one during early architectural design stage and one during late design stage of a project. Only the first one and partially the second one are top-down techniques. Although FPA and COCOMO-II-stage-I differ in detail, their overall approach is sufficiently similar to be treated commonly in this paper. In both approaches, the extent of the functionality of a planned software system is quantified by the abstract unit of function points (called "applications points" in COCOMO). Both approaches provide guidance in counting function points given an informal requirements description. Eventually, the effort is estimated by dividing the total number of function points by the productivity of the development team. (COCOMO-II-stage-I also takes the expected degree of software reuse

into account.) In particular COCOMO-II in the later two stages takes additional information about the software development project into account, such as the degree of generated code, stability of requirements, platform complexity, etc. Interestingly, architectural information is used only in a very coarse grained manner (such as number of components). Both approaches require a sufficient amount of historical data for calibration. Nevertheless, it is considered hard to make accurate predictions with top-down estimations techniques. Even Barry Boehm (the author of COCOMO) notes that hitting the right order of magnitude is possible, but no higher accuracy[1].

Bottom-Up Effort Estimation – Architecture-Centric Project Management [PB01] (ACPM) is a comprehensive approach for software project management which uses the software architecture description as the central document for various planning and management activities. For our context, the architecture based cost estimation is of particular interest. Here, the architecture is used to decompose planned software changes into several tasks to realise this change. This decomposition into tasks is architecture specific. For each task the assigned developer is asked to estimate the effort of doing the change. This estimation is guided by pre-defined forms. Also, there is no scientific empirical validation. But one can argue that this estimation technique is likely to yield more accurate prediction as the aforementioned top-down techniques, as (a) architectural information is used and (b) by asking the developer being concerned with the execution of the task, personal productivity factors are implicitly taken into account. This approach is similar to KAMP by using a bottom-up estimation technique and by using the architecture to decompose change scenarios into smaller tasks. However, KAMP goes beyond ACPM by using a formalized input (architectural models must be an instance of a predefined meta-model). This enables tool-support. In addition, ACPM uses only the structural view of an architecture and thus does not take software management costs, such as re-deployment into account.

3 The Karlsruhe Architectural Maintainability Prediction Approach

The Karlsruhe Architectural Maintainability Prediction (KAMP) approach enables software architects to compare architecture alternatives with respect to their level of difficulty to implement a specific change request. For each alternative the effort it takes to implement a change request is estimated. An important measure for maintainability is the change effort necessary for implementation of a change. Therefore, the capability of change effort estimation is very important in the context of maintainability prediction. Our method shows how change effort estimation can be embedded within an architectural maintainability prediction methodology and predicts the maintainability using change requests. This is because an architecture is not able to treat all change requests with the same level of ease. An architecture should be optimized in a way that frequent change requests can be implemented easier than less frequent ones. Our method makes explicit use of meta-modelled software architecture models. From software architecture models our method derives important inputs for change effort estimation in a semi-automatic way. Additionally, we use a bottom-up approach for estimation of change efforts. Thus, our method incorporates

[1] http://cost.jsc.nasa.gov/COCOMO.html

developer-based estimates of work effort. In order to get a higher precision of estimates our method helps in investigation of estimation supports. We do this by explicitly considering several kinds of influence factors, i.e. architectural properties, design and code properties, team organization properties, and development environment properties. *Use Case: Evaluate Single Change Request* One use case is to compare two architecture alternatives given a single change request. The method helps answering the question which alternative supports better the implementation of the given change request. *Use Case: Evaluate Multiple Change Requests* If there are several change requests at hand which occur with different frequencies the method helps answering questions like: Is there a dependency between given change requests? Is there a trade-off situation, where only one change request can be considered? Which architecture alternative provides best for implementation of a given set of change requests?

3.1 Properties of KAMP

Compared to the approaches presented in the related works section above, our approach has the following specific benefits: (a) a higher degree of automation, (b) lower influence of personal experience of the instructor, (c) stronger implicit consideration of personal developer productivity. Different to other approaches, we use architectural models, which are defined in a meta model (the Palladio Component Model). While this it no means to an end, the use of well defined architectural models is necessary to provide automated tool-support for architectural dependency analysis. Such tools are embedded into a general guidance through the effort estimation process.

3.2 Maintainability Analysis Process

As Figure 1 shows the maintainability prediction process is divided into the following phases: Preparation, Analysis, Result Interpretation. Each phase is described below. *Running Example:* In order to get a better understanding of the approach we provide a running example, which considers a client-server business application where several clients issue requests to a server in order to retrieve customer address information stored in a database. There are 100 Clients deployed in the system.

Preparation Phase In the model preparation phase, the software architect at first sets up a *software architecture description* for each architectural alternative. For this we use a meta-modelled component-based architecture model, which can be handled within an tool chain. *Example:* According to our running example the architects create an architecture model of the client-server architecture. They identify two architecture alternatives. In *Architecture Alternative 1 (AA1)* the clients specify SQL query statements and use JDBC to send them to the server. The server delegates queries to the database and sends results back to the client. In *Architecture Alternative 2 (AA2)* client and server use a specific IAddress Interface which allows clients to directly ask servers for addresses belonging to

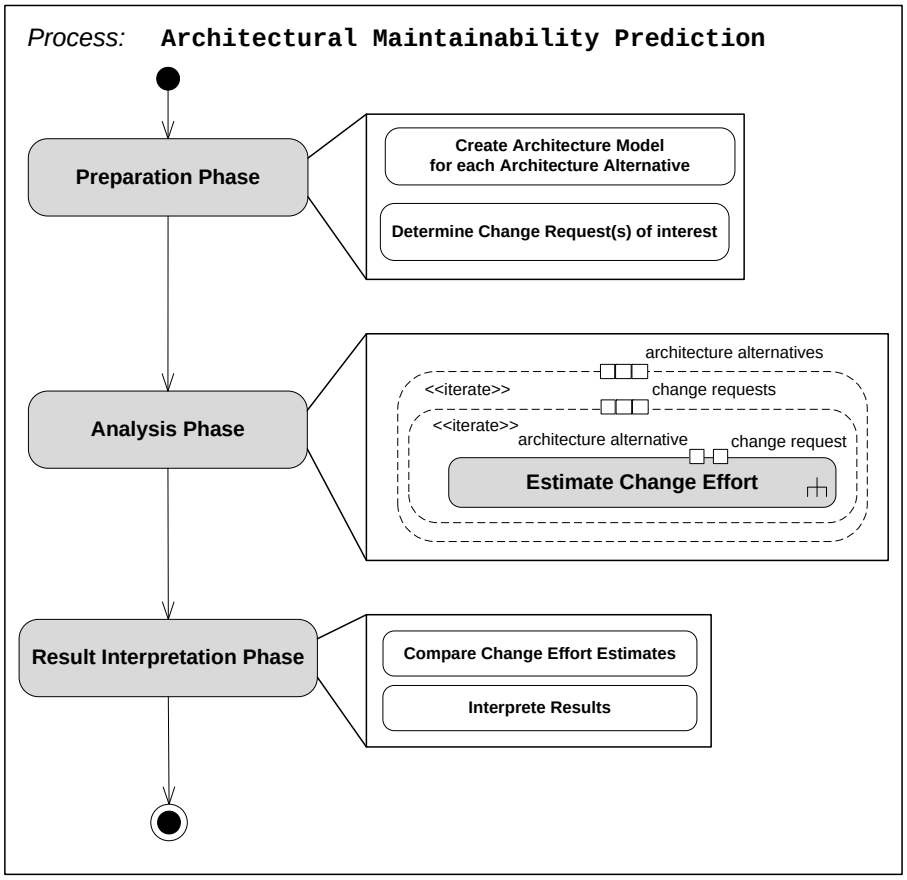


Figure 1: Architectural Maintainability Prediction Process

a certain ID. The server transforms requests into SQL query statements and communicates with the database via JDBC. The architecture model according to Figure 2 consists of a three components (Client, Server, Database). Several Clients are connected to one Server. Server and Database are connected via Interface Ports implementing JDBC Interface. The database schema is considered as a Data Type in the architecture model. Since there are two alternatives the architect creates two models. In alternative *AA1* Client component and Server component are connected via interface ports implementing JDBC Interface. In alternative *AA2* these interface ports use a IAddress Interface instead.

As a second preparation step the architect describes the considered *change requests*. A description of change request contains 1) a name, 2) an informal description of change cause, and 3) a list of already known affected architecture elements. *Example:* Independently from the system structure the architects have to design the database schema. Since they can not agree on a certain database schema they want to keep the system flexible enough

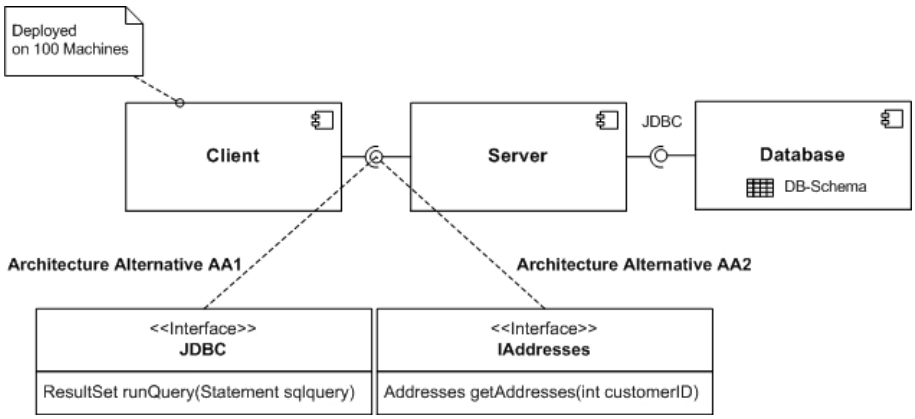


Figure 2: Running Example Architecture Overview

to handle changes to database schema. In the following paragraphs we show how the architects can use our approach to identify the better alternative with respect to expected changes to database schema. Hence, the following change request description is given: *Name:* CR-DBS, *Change Cause:* Database Schema needs to be changed due to internal restructuring, *List of known affected architecture elements:* Data Type "DB-Schema".

Maintainability Analysis Phase In the maintainability analysis phase for each architectural alternative and each change request a *Change Effort Estimation* is done. The process of change effort estimation is shown in Section 3.4. *Example:* The architects in our example want to analyse which architecture alternative (*AA1* or *AA2*) needs less effort to implement Change Request *CR-DBS*.

Result Interpretation Phase Finally the process summarises results, compares calculated change efforts and enriches them with qualitative information and presents them to the software architect.

3.3 Quality Model

Before we go on with the effort estimation process in Section 3.4 it is necessary to introduce the underlying maintainability quality model and used metrics. In general, maintainability characteristics have a rather qualitative and subjective nature. In literature maintainability is usually split up into several sub-characteristics. For example in [ISO90] the quality model divides maintainability into analysability, stability, changeability, testability and maintainability compliance. Unfortunately the given definitions of these terms are rather abstract and therefore not directly applicable. Thus, we first use a systematic way to derive maintainability characteristics and metrics. In order to get a quality model with adequate metrics which provide consequently for specific analysis goals the Goal-Question-Metrics method (GQM) [BCR94] is applied. In this approach, in the first step,

a set of analysis goals is specified by characteristics like analysis purpose, issue, object, and viewpoint as explained here: *1) Purpose:* What should be achieved by the measurement? *2) Issue:* Which characteristics should be measured? *3) Object:* Which artefact will be assessed? *4) Viewpoint:* From which perspective is the goal defined? (e.g., the end user or the development team). The next step is to define questions that will, when answered, provide information that will help to find a solution to the goal. To answer these questions quantitatively every question is associated with a set of metrics. It has to be considered that not only objective metrics can be collected here. Also metrics that are subjective to the viewpoint of the goal can be listed here. Regarding the GQM method we specify the following goal for maintainability analysis: *1) Purpose:* Comparison of Architectural Alternative AA_i and AA_j, *2) Issue:* Maintainability, *3) Object:* Service and Software Architecture with respect to a specific Change Request CR_k, *4) Viewpoint:* Software Architect, Software Developer. The following questions and sub-questions are defined according to the maintainability definitions above.

Question 1:	How much is the **maintenance effort** caused by the architectural alternative AA_i for implementing the Change Request CR_k?
Question 1.1:	How much is the **maintenance workload** for implementing the Change Request CR_k?
Question 1.2:	How much **maintenance time** is spent for implementing the Change Request CR_k?
Question 1.3:	How much are the **maintenance costs** for implementing the Change Request CR_k?

Based on the questions above we divide Maintenance Effort Metrics into Maintenance Workload Metrics, Maintenance Time Metrics, Maintenance Cost Metrics.

Maintenance Workload Metrics Maintenance Workload Metrics represent the amount of work associated with a change. To be more specific we consider several possible *work activities* and then derive counts and complexity metrics according to these work activities. Figure 3 shows how work activity types are found. A work activity is composed of a basic activity which is applied to an artefact of the architecture. Basic activities are *Add*, *Change*, and *Remove*. The considered architecture elements are *Component*, *Interface*, *Operation*, and *Datatype*. This list is not comprehensive, but can be extended with further architecture elements. Usually when we describe changes we refer to the *Implementation* of elements. In the case of *Interface* and *Operation* it is useful to distinguish *Definition* and *Implementation* since there is usually only one definition but several implementations which cause individual work activities. Some architecture meta-models use the concept of *Interface Ports* to bind an interface to a component. From the perspective of work activities an *Implementation of Interface* is equal to an *Interface Port*. In Figure 3 there is also a (incomplete) list of resulting work activity types. The following metrics are defined based on Work Activity Types.

Number of work activities of type WAT_i: This metric counts the number of work activities of type WAT_i. *Complexity annotations for work activity* WA_i: At this point several types of complexity annotations can be used, e. g. number of resulting activities, complexity

Basic Activities	Architecture Elements	Work Activity Set
Add	Component	Add Component
Change	Interface	Change Component
Remove	Operation	Remove Component
	Datatype	Add Interface
	...	...

Figure 3: Work Activity Types

of affected elements in source code, number of affected files, number of affected classes, number of resulting test cases to be run, number of resulting redeployments.
Maintenance Time Metrics These metrics describe effort spent in terms of design and development time. The following metrics are proposed in this category: *Working time for activity* WA_i: This metric covers the total time in person months spent for working on activity WA_i. *Time until completion for activity* WA_i: This metric describes the time between start and end of activity WA_i. *Total time for work plan* WP_i: This metric describes the time between start and end of work plan WP_i.
Maintenance Cost Metrics This subcategory represents maintenance efforts in terms of spent money. *Development costs for activity* WA_i */ work plan* WP_i: The money paid for development in activity WA_i or of work plan WP_i.

3.4 Change Effort Estimation Process

The architect has to describe how the change is going to be implemented. Based on the architecture model he points out affected model artefacts. Our approach proposes several starting points and guided refinements for detection of affected model elements. Starting points represent affected model element which the architect can *directly* identify. Three typical starting points are: 1) an already known Data Type change, 2) an already known Interface Definition change, 3) an already known Component change. Regarding our running example the architect identifies the following direct changes to the architecture: *Change of Datatype Implementation "DB-Schema"*. Directly identified changes are described as work activities in a work plan. A work plan is a hierarchical structured collection of work activities. We now stepwise refine the work plan into small tasks. This means we identify resulting changes and describe high-level changes on a lower level of abstraction. The idea is the effort of such low-level activities can be identified easier and with higher accuracy than high-level coarse grained change requests, in particular, as the latter do not include any architectural informations. There are several types of relations between architectural elements. These relations help to systematically refine change requests and work plans. In the following, these different types of relations are defined, their role in work plan refinement is discussed and examples are given. Thereby, the actual relations depend of the architectural meta model used. However, in one way or another such relation types are present in many architectural meta models. In the following, we present relations

which are present in the Palladio Component (Meta-) Model (PCM) and the Q-ImPrESS Service Architecture Meta-Model (SAMM). **Include- / contains-relations:** Architectural elements which are contained in each other, are in a contains-relationship. This means, that any change of the inner element implies a change of the outer element. Also, any work activity of the outer element can be refined in a set of work activities of the inner elements. *Examples:* A System consists of Components. A Component has Interface Ports (i.e. Implementation of Interface). An Implementation of Interface contains Implementations of Operations. In our running example a change to the database schema also implies a change to database component. Hence, we get another work activity: *Change Component Impl. of "Database" Component*. **References-relations:** Architectural elements which reference (or use) other elements are related by a reference-relation. Such relations are the base for architectural change propagation analyses. Such analyses allow to find potential follow-up changes. This means, that the using entity which references a used entity is potentially affected by a change of the used entity. The direction of the change propagation is reversed to the direction of the reference-relation. This implies, that on a model level one needs to navigate the backward direction of reference-relations. *Examples:* An Interface Port references a Definition of Interface. A Definition of Operation uses a Data Type. A Definition of Interface uses transitively a Data Type. In our running example a change to Data Type "DB-Schema" also affects the JDBC Interface and all Interface Ports which implement the JDBC interface. In both alternatives (*AA1* and *AA2*) the Server component has a JDBC-implementing interface port. Hence, we get the additional work activity: Change Interface Port "JDBC" of Component "Server". In alternative *AA1* client components also implement the JDBC interface. Hence, another work activity is identified: Change Interface Port "JDBC" of Component "Client". By using kinds of relations we systematically identify work activities. For each work activity additional complexity annotations are derived. This can comprise architecture properties (e. g. existing architecture styles or patterns), design / code properties (e. g. how many files or classes are affected), team organization properties (e. g. how many teams and developers are involved), development properties (e. g. how many test cases are affected) system management properties (e. g. how many deployments are affected). If those complexity annotations are present in the architecture model they can be gathered with tool-support. *Examples:* In our architecture meta-model we have a deployment view which specifies deployment complexity of components. In our running example client components are deployed on 100 machines. Thus, a change activity affecting Client components also implies a redeployment on 100 component instances. After work activities are identified and workload metrics are calculated the architect has to assign time effort estimates for all work activities. Following a bottom-up approach the architect presents low-level activities to respective developers and asks them to give estimates. The results of the change estimation process are 1) the work plan with a detailed list of work activities and annotated workload metrics, time effort estimates and costs, and 2) aggregated effort metric values.

Conclusions and Future Work In this paper, we presented a quantitative architecture-based maintainability prediction method. It estimates the effort of performing change requests for a given software architecture. Costs of software re-development *and* costs of software management are considered. By this, KAMP takes a more comprehensive

approach than competing approaches. KAMP combines the strength of a top-down architecture based analysis which decomposes the change requests into smaller tasks with the benefits of a bottom-up estimation technique. We described the central change effort estimation process in this paper. By extrapolating from the properties of our approach compared to other qualitative approaches, we claim the benefits of lower influence of personal experience on the prediction results accuracy and a higher scalability through a higher degree of automation. By using bottom-up estimates we claim the personal productivity is implicitly reflected. However, it is clear that we need to empirically validate such claims. Therefore, we plan as future work a larger industrial case study and several smaller controlled experiments to test the validity of these claims.

Acknowledgements – This work was funded in the context of the Q-ImPrESS research project (http://www.q-impress.eu) by the European Union under the ICT priority of the 7th Research Framework Programme.

References

[BB99] P. Bengtsson and J. Bosch. Architecture level prediction of software maintenance. *Software Maintenance and Reengineering, 1999. Proc. of the Third European Conference on*, pages 139–147, 1999.

[BCR94] V. Basili, G. Caldeira, and H. D. Rombach. *Encyclopedia of Software Engineering, chapter The Goal Question Metric - Approach.* Wiley, 1994.

[BKR07] S. Becker, H. Koziolek, and Ralf H. Reussner. Model-based Performance Prediction with the Palladio Component Model. In *WOSP '07: Proc. the 6th Int. Works. on Software and performance*, pages 54–65, New York, NY, USA, Febr. 2007. ACM.

[BLBvV04] P. Bengtsson, N. Lassing, J. Bosch, and H. van Vliet. Architecture-level modifiability analysis (ALMA). *Journ. of Systems and Software*, 69(1-2):129 – 147, 2004.

[Boe00] Barry W. Boehm, editor. *Software cost estimation with Cocomo II*. Prentice Hall, Upper Saddle River, NJ, 2000.

[BZJ04] M.A. Babar, L. Zhu, and R. Jeffery. A framework for classifying and comparing software architecture evaluation methods. *Software Engineering Conference, 2004. Proceedings. 2004 Australian*, pages 309–318, 2004.

[CKK05] Paul Clements, Rick Kazman, and Mark Klein. *Evaluating software architectures*. Addison-Wesley, 4. print. edition, 2005.

[dCP08] David Bettencourt da Cruz and Birgit Penzenstadler. Designing, Documenting, and Evaluating Software Architecture. Technical Report TUM-INFO-06-I0818-0/1.-FI, Technische Universität München, Institut für Informatik, jun 2008.

[Dob02] E. Dobrica, L.; Niemela. A survey on software architecture analysis methods. *Transactions on Software Engineering*, 28(7):638–653, Jul 2002.

[IEE07] ISO/IEC IEEE. Systems and software engineering - Recommended practice for architectural description of software-intensive systems. *ISO/IEC 42010 IEEE Std 1471-2000 First edition 2007-07-15*, pages c1–24, 15 2007.

[IFP99] IFPUG. *Function Point Counting Practices Manual*. International Function Points Users Group: Mequon WI, 1999.

[ISO90] ISO/IEC. Software Engineering - Product Quality - Part 1: Quality. *ISO/IEC 9126-1:2001(E)*, Dec 1990.

[PB01] Daniel J. Paulish and Len Bass. *Architecture-Centric Software Project Management: A Practical Guide*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

Verteilte Evolution von Softwareproduktlinien: Herausforderungen und ein Lösungsansatz

Klaus Schmid

Institut für Informatik, Universität Hildesheim,
Marienburger Platz 22, D-31141 Hildesheim
schmid@sse.uni-hildesheim.de

Die Evolution einzelner Systeme ist bereits relativ komplex, doch alle diese Probleme findet man potenziert in einer Produktlinienentwicklung. Zusätzlich zu den Einzelsystemproblemen treten weitere Schwierigkeiten hinzu. In diesem Beitrag gehen wir auf die Herausforderungen der Produktliniensituation ein und legen unser Augenmerk vor allem auf die Situation verteilter Evolution. Wir stellen einen möglichen Ansatz zur Lösung vor.

1 Einleitung

Im Laufe der letzten 10-15 Jahre ist die Entwicklung von Software in Form einer *Produktlinie* für die meisten Unternehmen zum Normalfall der Softwareentwicklung geworden. Das heißt, eine Menge von Systemen wird in integrierter Weise mit Hilfe gemeinsamer Komponenten entwickelt. Dies geschieht vor allem, um die Entwicklungszeit zu verkürzen und die Kosten zu verringern [LSR07, CN01]. Betrachtet man also das Problem der Softwareevolution und insbesondere der Legacy Systeme aus dem Blickwinkel der softwareentwickelnden Organisationen, so macht es Sinn das Problem im Zusammenhang mit der *Evolution von Produktlinien* insgesamt zu betrachten.

Das Problem der Produktlinienevolution ist aus den folgenden Gründen eine echte Obermenge der Evolution von Einzelsystemen:

- Es umfasst alle Problematiken der Evolution von Einzelsystemen, den jedes einzelne System kann jeweils zum Legacy-System werden.
- Änderungen, die sich die Kunden der einzelnen Systeme wünschen, können sich stark widersprechen, müssen aber in einer Produktlinie zusammengeführt werden. Dies führt zu einer wesentlichen Komplexitätserhöhung.
- Die Lebensdauer einer Produktlinie ist – da sie viele Produkte umfasst – meist deutlich länger als die eines einzelnen Produkts.

Auf Grund dieser Herausforderungen gibt es aktuell noch keine zufriedenstellenden Lösungen für die Produktlinienevolution.

Als eine weitere Schwierigkeit komm bei der Produktlinienentwicklung hinzu, dass einzelne Produkte häufig einen Plattformcharakter haben. Das heißt, Kunden der Produkte führen selbst Anpassungen durch, die dann wiederum zu einer besonderen Produktvariante führen. Dabei entsteht die Schwierigkeit, dass das produktlinienentwickelnde Unternehmen diese Produkte meist nicht kennt, die aufbauenden Produkte sollten aber auch bei einer Aktualisierung der zugrundliegenden Produktlinienelemente intakt bleiben.

Der weitere Beitrag ist wie folgt strukturiert. Der nächste Abschnitt führt kurz die Grundbegriffe für Produktlinienentwicklung ein und stellt das Problem der Evolution näher da. In Abschnitt 3 skizzieren wir unseren Lösungsansatz und stellen wesentliche Anforderungen dar, die er mit sich bringt. Eine abschließende Diskussion bietet Abschnitt 4.

2 Produktlinienevolution

In diesem Abschnitt führen wir die wesentlichen Begriffe der Produktlinienentwicklung ein und diskutieren die Herausforderungen der Produktlinienevolution genauer.

2.1 Produktlinienentwicklung

Die Grundidee einer Produktlinienentwicklung ist es eine Menge von Systemen gemeinsam aus einer Menge von Assets zu entwickeln, wobei eine möglichst hohe Wiederverwendung erreicht werden soll. Dies kann einerseits erreicht werden indem die Zusammensetzung der Komponenten für die einzelnen Systeme unterschiedlich ist. Zum anderen können die Komponenten selbst wieder parametrisierbar sein, und schließlich gibt es die Möglichkeit produktspezifischer Komponenten. Um diese Flexibilität zu ermöglichen spielt in einer Produktlinienentwicklung die Softwarearchitektur eine wesentliche Rolle. Sie beschreibt eine konfigurierbare Softwarearchitektur; die einzelnen Systeme instanziieren diese, um zur jeweiligen Produktarchitektur zu gelangen. Die Produktlinienarchitektur wird daher auch als Referenzarchitektur bezeichnet.

Von besonderer Bedeutung ist in einer Produktlinienentwicklung das Variabilitätsmodell. Dieses Modell beschreibt, wie die verschiedenen Produkte variieren können, und damit welche Eigenschaften von der Produktlinie insgesamt unterstützt werden. Das Variabilitätsmodell nennt dabei die möglichen Ausprägungen von Eigenschaften und definiert insbesondere Abhängigkeiten zwischen diesen Eigenschaften. Verschiedene Arten von Variabilitätsmodellen sind gebräuchlich. Am häufigsten werden sogenannte Featuremodelle [KC+90, RSP03, CHE05a] verwendet, aber auch Entscheidungsmodelle [SJ04] werden genutzt. Da die Darstellung von Featuremodellen – insbesondere in der Form von Featurebäumen – sich besonders gut zur Erläuterung eignet, nutzen wir diese Darstellungsweise hier. Es gibt verschiedene Varianten auch dieser Notation. Hier nutzen wir eine Variante, die sich weitestgehend auf FODA [KC+90] stützt (vgl. Abbildung 1). Dieses Modell unterstützt die Dekomposition einer Produktlinie in ihre Bestandteile, die Auszeichnung sogenannter optionaler Features (diese können

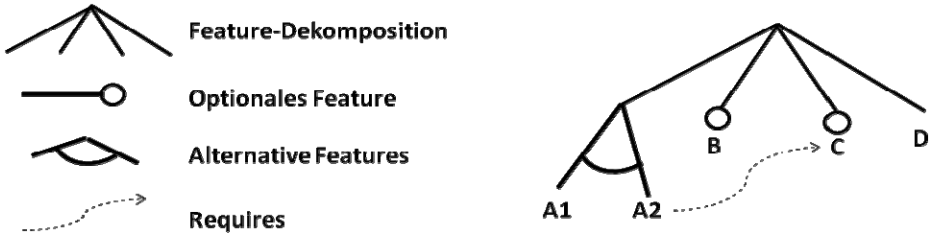


Abbildung 1: Notation und Beispiel für einen Featurebaum

Bestandteil eines Produkts sein, müssen es aber nicht), sowie die Repräsentation von Alternativen. Außerdem wird dabei die Abhängigkeit von Eigenschaften (requires) dargestellt. Abbildung 1 stellt beispielhaft eine Produktlinie dar, in der jedes Produkt die Eigenschaft A1 oder A2 (aber nicht beide) hat, Produkte die Eigenschaft B bzw. C haben können, aber nicht müssen. Die Eigenschaft D ist verpflichtend. Die Abhängigkeit zwischen A2 und C drückt aus, dass alle Produkte, die A2 enthalten auch C enthalten müssen. Weiterhin können zu Features Parameterwerte gehören. Diese werden aber in dieser Notation nicht graphisch dargestellt. Dies ist eine sehr einfache Variante der Featuremodellierung; auch wollen wir hier nicht auf semantische Formalisierungen (bspw. [SHT06]) eingehen, um die konzeptionelle Diskussion hier nicht unnötig kompliziert zu machen.

2.2 Evolution von Produktlinien

Im Vergleich zu vielen anderen Aspekten der Produktlinienentwicklung gibt es bisher noch relativ wenige Arbeiten zur Evolution von Produktlinien. Eine ausführliche Diskussion möglicher Evolutionsschritte einer Produktlinie haben wir bereits an anderer Stelle gegeben [SE07], wir wollen aus Platzgründen daher hier nur die wesentlichen Ergebnisse zusammenfassen. Auslöser einer Produktlinienevolution (Change Requests) kann man beispielsweise nach folgenden Kriterien kategorisieren:

- *Granularität der Änderungen:* wie viele Produkte werden von den Änderungen betroffen?
    - *Anforderungsebene:* einzelne Anforderungen werden verändert
    - *Produktebene:* die Menge der zu unterstützenden Produkte wird verändert
    - *Produktlinienebene:* Änderungen haben Auswirkungen auf ganze Produktgruppen
- *Arten von Änderungen:* für jede der vorhergehenden Ebenen lassen sich nun Änderungsoperationen benennen.
    - *Hinzufügen* von Anforderungen oder Produkten

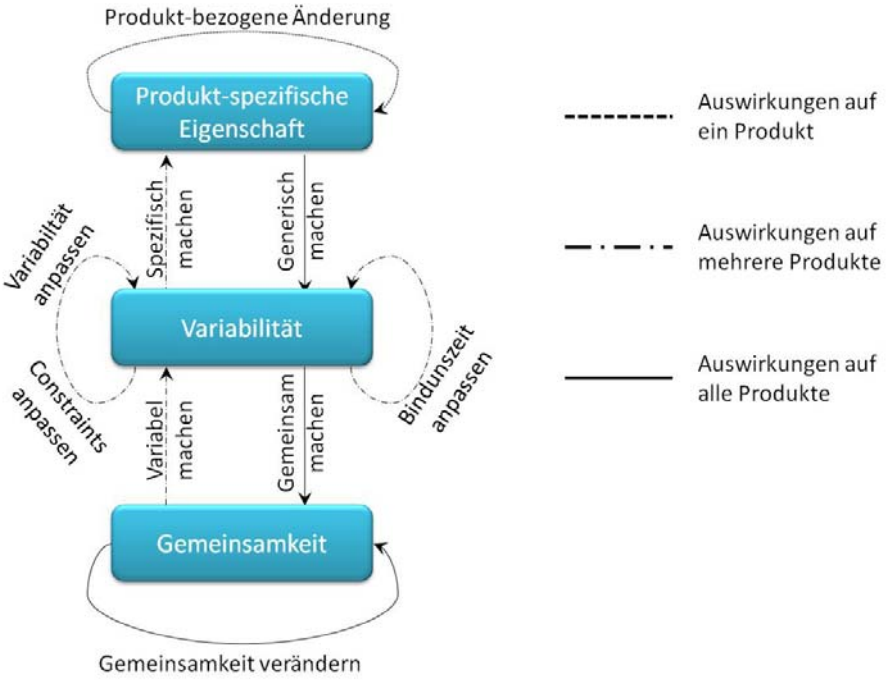


Abbildung 2: Übersicht über mögliche Evolutionsschritte für Produktlinien [SE07].

- *Entfernen* von Anforderungen oder Produkten
- *Modifizieren* von Anforderungen oder Produkten

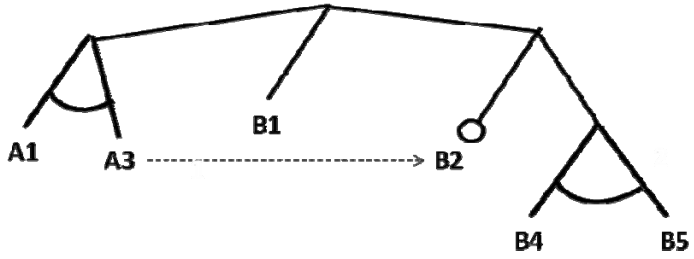
Um die detaillierten Auswirkungen und Möglichkeiten zu verstehen (so macht es bspw. einen Unterschied ob eine neue Alternative eingeführt wird oder zu einer bestehenden Alternative eine neue Möglichkeit hinzugefügt wird), ist eine genaue Betrachtung notwendig.

- Zu den aufgeführten Änderungstypen kommen noch Spezialformen, wie beispielsweise die Veränderung der Auswahlmöglichkeiten, die Änderung von Abhängigkeiten, usw.

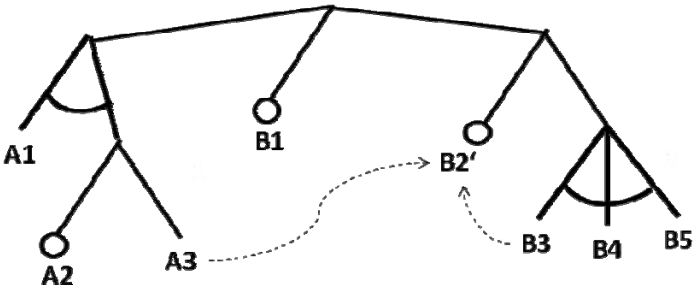
Eine Übersicht gibt Abbildung 2. Eine detaillierte Diskussion ist in [SE07] gegeben.

2.3 Ein Beispiel zur Produktlinienevolution

Als Grundlage zur Diskussion von Ansätzen zur Produktlinienevolution wollen wir hier ein einfaches Beispiel einführen. Das entsprechende Featuremodell findet sich in Abbildung 3. In Abbildung 3a ist ein einfaches Featuremodell für eine Produktlinie



(a) Ausgangssituation



(b) Endsituation

Abbildung 3: Ein einfaches Evolutionsbeispiel: Anfangs- und Endsituation

dargestellt, die zu einem Zeitpunkt t_1 die Ausgangssituation darstellt. Nehmen wir jetzt folgende Änderungsanforderungen an:

- *Kunde 1* hätte gerne eine Erweiterung der Produktlinie um das Feature A2. Damit es nur diesem Kunden zur Verfügung gestellt werden kann, wird dieses als optionales Feature realisiert. (Wir behandeln hier zur Vereinfachung produktspezifische /kundenspezifische Funktionalität wie Variabilitäten.)

- *Kunde 2* ist selbst in der Lage Anpassungen durchzuführen. Dies kann beispielsweise der Fall sein, da ihm die Realisierung (soweit sie ihn betrifft) zur Verfügung steht. Er erweitert die Alternative B4, B5 um eine weitere Ausprägung B3. Diese erfordert jedoch die das Feature B2. Da diese Erweiterung vom Kunden selbst durchgeführt wird, wird sie (und die Abhängigkeit) dem Hersteller eventuell nie bekannt. Eine entsprechende Situation gibt es beispielsweise in der Entwicklung für Standardsoftware.

Nun gibt es weitere Änderungen:

- Zum Zeitpunkt t_2 wird B1 aus einem gemeinsamen Feature in ein optionales Feature überführt. Nach Aussage des Variabilitätsmodells sollte dies keinerlei Auswirkungen auf die anderen Features haben – und insbesondere auf die möglichen Varianten. Da jedoch zum Zeitpunkt t_1 B1 noch als verpflichtend angenommen wurde, besteht eine gewisse Wahrscheinlichkeit, dass evtl. Abhängigkeiten nicht dokumentiert sind. Auch erfordert die Integration von

Variabilitätsmechanismen weitergehende Implementierungsänderungen, die ebenfalls andere Features beeinflussen können.

- Zum Zeitpunkt t_3 wird schließlich B2 in B2´ überführt. Dies kann bspw. auf Grund von Fehlerkorrekturen oder auf Basis ergänzender Anforderungen notwendig werden. Da A3 abhängig ist von B2 kann dies natürlich Auswirkungen auf A3 haben. Vor allem wenn diese nutzerseitig sichtbar sind, kann dies dazu führen, dass dies nicht gewünscht wird. Schwieriger noch ist es für die Ergänzung B4, da das Unternehmen, welches für die Produktlinien-infrastruktur verantwortlich ist, nichts von B3 weiß, ist eine Überprüfung des Gesamtresultats nicht möglich. Trotzdem ist unter allen Umständen zu vermeiden, dass B3 nicht mehr funktionstüchtig ist.

In der Praxis lassen sich angesichts dieser Schwierigkeiten nun verschiedene Ansätze zur Produktlinienevolution beobachten:

- Der einfachste Weg ist: für jeden Kunden wird eine Kopie der Infrastruktur erzeugt. Dies vermeidet Abhängigkeiten zwischen Änderungen für unterschiedliche Kunden. Jedoch hat dies zur Folge, dass jede Infrastrukturänderung, die für alle Kunden relevant ist, mehrfach realisiert werden muss. Aus diesem Grund wird dieser Ansatz manchmal auch als nicht zum Produktlinienkonzept kompatibel aufgefasst. Im Fall von Kunde 2 wäre auch nach wie vor offen, ob seine eigenen Ergänzungen (B3) noch funktionstüchtig sind, wenn er eine Produktbasis erhält, die B2´ enthält.

- Ein weiterer Ansatz ist der Versuch alle Evolution in der Produktlinie zu realisieren. Dies hat jedoch auch wesentliche Folgen. So können Ergänzungen wie die von Kunde 2 nicht mehr durchgeführt werden, ohne dass dies dem Hersteller bekannt ist. Dies heißt auch, dass sobald ein Kunde eine Fehlerkorrektur in einer Funktionalität übernehmen will, er in allen Features die aktuellste Version übernehmen muss.

Dies sind nur Beispiele: insgesamt haben heutige Evolutionsänsätze eine Menge von Schwierigkeiten. Beispiele sind:

- Kunden möchten manche Erweiterungen und Veränderungen der Infrastruktur nicht mitmachen. Diese können bspw. durch Änderungen für andere Kunden oder durch allgemeine Evolutionsentscheidungen auftreten. Diese Trennung ist nicht möglich.

- Die Integration in die Infrastruktur ist komplexer als eine Einzelsystem-änderung. Diese zusätzliche Zeit steht manchmal nicht zur Verfügung; eine Anpassung muss manchmal speziell für einen Kunden erfolgen.

- Kunden haben spezielle Anpassungen für ihr System erhalten, diese sollen natürlich auch bei aktualisierten Versionen noch unterstützt werden.

- Die Entwicklung findet verteilt statt (evtl. Firmenübergreifend). Dadurch gibt es keine einheitliche Sicht auf die Produktlinie.

- ..

Dies sind nur einige Beispiele für Herausforderungen im Bereich der Produktlinien-evolution.

3 Lösungsansatz

In diesem Abschnitt wollen wir die Grundlagen für einen Evolutionsansatz, der die oben beschriebenen Probleme mindert oder sogar löst skizzieren. Wesentliche Charakteristika dabei sind:

- Eine Abbildung der Produktlinienevolution auf Produktlinienvariabilität wird vorgenommen. Das heißt, das Variabilitätsmodell deckt alle Varianten ab, auch die von früheren Plattformversionen. Variabilitäten können erst entfernt werden, wenn keine entsprechenden Produkte mehr im Feld sind.

- Um das Problem der Zeitverzögerung durch die Integration in die Infrastruktur zu lösen, kann zuerst lediglich das Variabilitätsmodell integriert werden, während die Realisierung als getrennte Version verbleibt. In weiteren Schritten kann dann eine tiefere Integration erfolgen.[1]

- Verschiedene Kunden (oder Entwicklungspartner) haben eventuell nur Zugriff auf einzelne Ausschnitte des Gesamtmodells (inklusive des Variabilitäts-modells). Trotzdem betrachten wir alle Bestandteile eines solchen verteilten Variabilitätsmodells als ein (virtuelles) Variabilitätsmodell.

- In der Aktualisierung von Systemen werden minimale Änderungen angestrebt (ein Kunde erhält ein Update, das nur die Änderungen enthält, die für den Kunden notwendig sind). Das heißt auf Artefaktebene sollen kleine (nicht notwendigerweise formal minimale) Change-Sets identifizierbar sein.

[1] Dies kann dann als Integration über das Versionsmanagement gesehen werden, ein Ansatz der von Produktlinienwerkzeugen wie GEARS [GEARS] gar als einzigem Ansatz unterstützt wird. Jedoch wird dort die spätere Integration nicht explizit unterstützt.

3.1 Charakteristika des Ansatzes

Eine solche Vorgehensweise erfordert es zu einer einzelnen Variation die notwendigen Artefakte zuzuordnen. Nur so können die einzelnen Auslieferungen auf die jeweils notwendigen Artefakte beschränkt werden. Eine solche Menge von ein Feature (bzw. eine Featureänderung realisierenden Artefakten) bezeichnen wir als *Featurebundle*. Dabei ist es notwendig die Verbindung zwischen den Variabilitäten und den jeweiligen Lebenszyklusmodellen zu beachten, bzw. zu etablieren. Dies wird in Abbildung 4 dargestellt: das Variabilitätsmodell tritt als Konfigurationsmodell auf und beschreibt die Anpassung der weiteren Modelle, um jeweils spezifische Produkte abzuleiten. So werden Instanzen in einheitlicher Weise beschrieben, bei denen Anforderungen, Architektur, Implementierung und Tests jeweils zu einer bestimmten Systeminstanz gehören.

Wollen wir nun eine Modularisierung beschreiben, um zu ermöglichen, dass Kunden ihre eigenen Anpassungen durchführen (oder diese von Dienstleistern durchführen lassen), so ist es notwendig, dass Variabilitätsmodell und Artefaktmodelle (Anforderungen, .., Test) in gleicher Weise modularisiert werden. Wir definieren also:

Ein *Featurebundle* besteht aus:

- Einem Variabilitätsmodellfragment
- Allen dazugehörigen (also durch das Variabilitätsmodellfragment) beeinflussten Anforderungen
- Allen dazugehörigen Architekturelementen
- Allen dazugehörigen Implementierungselementen (bspw. neue Realisierungen für die durch die Architekturelemente betroffenen Komponenten)
- Allen dazugehörigen Qualitätssicherungsmodelle (u.a., Tests)

Das heißt ein Featurebundle beschreibt einen Querschnitt durch die Produktlinie, der alle zu einem Feature gehörenden Artefaktelemente umfasst.

Ein *Featureset* fasst alle *Featurebundles*, die zu einem Zeitpunkt für einen Kunden relevant sind, zusammen. Eine solche Menge von Änderungen kann auch nur relativ zu einer bestimmten Version einer Produktlinie interpretiert werden. Diese Version bezeichnen wir deshalb auch als *Basis*. Ein Featureset beschreibt damit eine Ergänzung oder Veränderung der Produktlinie um zuvor nicht vorhandene Eigenschaften. Dabei können neue Variabilitäten nur zu dem Zweck eingeführt werden, um die für einen Kunden relevanten Eigenschaften abzudecken.

Um die Bedürfnisse verteilter Entwicklung im Allgemeinen und der kundenspezifischen Entwicklung durch Dritte im Besonderen abzudecken, gehen wir davon aus, dass das Gesamtmodell im Sinne der Kombination von Basis vereinigt mit der Menge aller Bundles in keiner Entwicklungsorganisation vollständig vorliegen muss. Derartige Verteilungsaspekte spielen heute bereits in der praktischen Produktlinienentwicklung eine große Rolle, doch wurde dies im Bereich der Variabilitätsmodellierung weitestgehend ignoriert. Nur wenige Arbeiten berücksichtigen die Möglichkeit zur Fragmentierung von Featuremodellen – und selbst diese gehen davon aus, dass zu einem bestimmten Zeitpunkt eine vollständige Integration des Variabilitätsmodells hergestellt wird (bspw. [DN+08]). In Bezug auf das in Abschnitt 2.3 dargestellte Beispiel würde dies bedeuten, dass die Ergänzung, die von Kunde 2 eigenständig durchgeführt wird als eigenständiges Variabilitätsmodellfragment vorliegt. Dieses würde dann beschreiben: es findet eine Ergänzung einer bestimmten Alternative um eine Ausprägung B3 statt. Diese ist dabei abhängig von B2, welches aber nicht weiter definiert wird, da es nicht von der Kundenorganisation realisiert wird. Wenn nun ein aktualisiertes Featureset des Herstellers beim Kunden eintrifft, dann kann eine Integration stattfinden und das resultierende Modell auf Konsistenz geprüft werden.

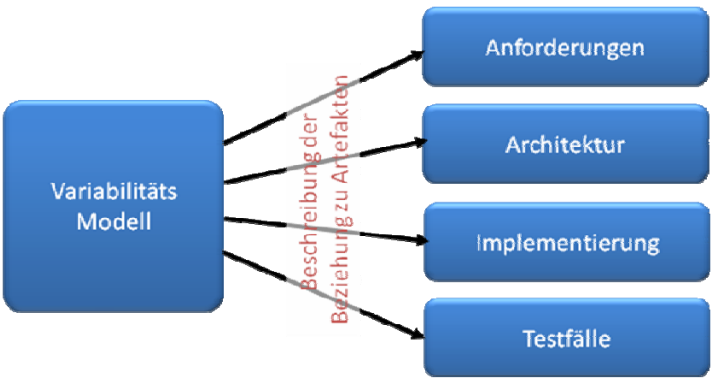


Abbildung 4: Beziehung zwischen Variabilitätsmodell und Entwicklungsartefakten

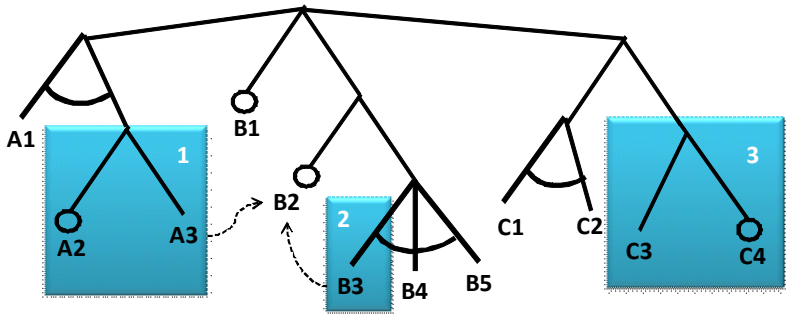


Abbildung 5: Variabilitätsmodell mit Feature Bundles

3.2 Modularisierung des Variabilitätsmodells

Bei der oben beschriebenen Vorgehensweise ist eine Modularisierung des Variabilitätsmodells wesentlich, da nur so eine Unabhängigkeit einzelner Änderungen voneinander sichergestellt werden kann. Im Grunde geht es hier um die gleiche Problematik, die wir auch aus anderen Bereichen der Realisierung kennen: nur durch eine geeignete Modularisierung kann eine leichte Änderbarkeit erreicht werden. Dies gilt um so mehr, wenn die Evolution (wie im Beispiel) verteilt erfolgen muss. Um die Modularisierung von Variabilitätsmodellen weiter zu untersuchen, sind zwei Fragen besonderes relevant:

- Welche Arten der Zerlegung von Variabilitätsmodellen sollen unterstützt werden?
- Wie lässt sich eine Modularisierung (im Sinne einer Kapselung) der Variabilitätsmodelle herstellen?

Die Kapselung und Zerlegung von Teilkonfigurationsmodellen ist heute in der Praxis bereits bei Installations- und Updatewerkzeugen gebräuchlich. Jedoch wurde dies bisher nur beschränkt theoretisch behandelt [SK09]. Da einzelne Systembestandteile von unterschiedlichen Entwicklungsorganisationen geliefert werden, wird dies oft nicht als Produktlinienentwicklung wahrgenommen, jedoch kann ein solches Konfigurationsproblem weitestgehend einer verteilten Produktlinienentwicklung gleich gesetzt werden.

Neben einem eigenen Namensbereich ist bei einer Modularisierung vor allem wichtig Import- und Exportschnittstellen zu definieren. Eine Import-Schnittstelle definiert welche Teile des Basismodells bzw. darunterliegender Featurebundels genutzt werden sollten, eine Exportschnittstelle definiert die möglichen Variabilitäten, sowie die Punkte an denen zusätzliche Erweiterungen möglich sind (wie dies bspw. Eclipse mittels Extensionpoints tut). Dies erlaubt eine beliebige Kombination mehrerer Teilfeaturemodell, wobei die Konsistenz überprüfbar bleibt.

3.3 Modularisierung der Artefaktmodelle

Die wohl größere Herausforderung stellt die Modularisierung der Artefaktmodelle dar. Entsprechend Abbildung 4 sollten alle Artefakte ebenso modularisiert werden, wie das Variabilitätsmodell. Die Modularisierung von Implementierungen und von Architekturen ist heutzutage vergleichsweise gut verstanden. Die Modularisierung von Anforderungen wurde bisher jedoch noch weniger behandelt. Zwar werden Anforderungen regelmäßig in Teilprojekten und –aufgaben zerlegt, jedoch entspricht diese Form der Zerlegung meist nicht direkt einer architektonischen Zerlegung.

4 Diskussion

In diesem Aufsatz sind wir auf die Problematik der Evolution von Softwareproduktlinien eingegangen. Während alle „klassischen“ Probleme der Systementwicklung auftreten, kommen zusätzliche Probleme durch die Abhängigkeiten zwischen den Produkten hinzu. Auf diesen Aspekten lag hier ausschließlich der Fokus. Wir haben in diesem Aufsatz vor allem versucht einige Herausforderungen der Produktlinienevolution herauszuarbeiten und ein Konzept zur verbesserten Evolution darzustellen. Dieses Konzept beruht insbesondere darauf, dass es das Variabilitätsmodell selbst zur Grundlage des Evolutionsmanagements macht.

Viele Aspekte dieses Konzepts müssen durch weitergehende Forschungsarbeiten noch bearbeitet werden. Als wesentliche Fragen seien hier nur beispielhaft genannt:

- Wie lassen sich Variabilitätsmodelle geeignet in Module zerlegen, so dass eine Kombination einfach möglich ist und eine gute Kapselung gegeben ist.
- Welche Anforderungen an den Aufbau (Semantik) solcher Module sind zu stellen, um die Kombinierbarkeit von Modulen sicher zu stellen.
    - Lokale Identifikation von globalen Erfüllbarkeitsproblemen
    - Richtlinien zur Erstellung von Variabilitätsmodulen
- Wie lässt sich eine entsprechende Modularisierung auf (PlugIn-)Architekturen abbilden, so dass eine gute Evolvierbarkeit gegeben ist.
- Wie lassen sich Modularisierungen von Anforderungen gestalten. Wie sehen Schnittstellen solcher Module aus?
- Entsprechend die Modularisierung von Tests

Dies deutet nur einen kleinen Ausschnitt der Vielzahl von Fragestellungen an, die im Zusammenhang mit einem solchen Ansatz zu lösen sind. Im Rahmen der Diskussion haben wir auch versucht aufzuzeigen, dass die Nutzung von Produktlinienkonzepten wie Variabilitätsmodellierung, Meta-Variabilität, usw. helfen kann eine systematische Evolution von Produktlinien auch im verteilten Fall zu unterstützen.

Danksagung

Der Autor möchte Hr. Dr. Hübner, Hr. Keunecke und Hr. Brummermann für anregende Diskussion und die Darlegung der grundsätzlichen Evolutionsproblematik danken.

Literaturverzeichnis

[CHE05a] K. Czarnecki, S. Helsen, and U. Eisenecker, *Formalizing cardinality-based feature models and their specialization*, in Software Process: Improvement and Practice, vol. 10, no. 1, pp. 7–29, 2005.

[CHE05b] K. Czarnecki, S. Helsen, and U. Eisenecker, *Staged configuration through specialization and multi-level configuration of feature models*, Software Process Improvement and Practice, vol. 10, no. 2, pp. 143–169, 2005.

[CN01] P. Clements and L. Northrop, Software Product Lines: Practices and Patterns, Addison-Wesley, 2001.

[DN+08] D. Dhungana, T. Neumayer, P. Gruenbacher, R. Rabiser, *Supporting the Evolution of Product Line Architectures with Variability Model Fragments*, Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008), pp. 327-330, 2008.

[GEARS] www.biglever.com

[KC+90] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, 1990.

[LSR07] F. van der Linden, K. Schmid, and E. Rommes. *Product Lines in Action*. Springer Verlag, 2007.

[RSP03] M. Riebisch, D. Streitferdt, and I. Pashov, Modeling variability for object-oriented product lines. in ECOOP Workshops, LNCS 3013. Springer, pp. 165–178, 2003.

[SE07] K. Schmid, H. Eichelberger. *A Requirements-Based Taxonomy of Software Product Line Evolution*, Proceedings of the Third International ERCIM Symposium on Software Evolution (Software Evolution 2007), Oktober 2007.

[SE08] K. Schmid, H. Eichelberger. *Model-Based Implementation of Meta-Variability Constructs: A Case Study using Aspects*, Second International Workshop on Variability Modeling of Soft-ware-Intensive Systems, ICB-Research Report No. 22, ISSN 1860-2770, Seiten 63-71, 2008.

[SJ04] K. Schmid, I. John. A Customizable Approach to Full Lifecycle Variability Management, Science of Computer Programming, Vol. 53, No. 3, Seite 259-284, 2004.

[SK09] K. Schmid, C. Kröher. *An Analysis of Existing Software Configuration Systems*. Third International Workshop on Dynamic Software Product Lines (DSPL 2009) at the Software Product Line Conference, 2009.

[SHT06] P. Schobbens, P. Heymans, J. Trigaux, *Feature Diagrams: A Survey and a Formal Semantics*. 14th Requirements Engineering Conference (RE'06), pp. 139-148, 2006.

Qualitätssichernde Koevolution von Architekturen eingebetteter Systeme

Malte Lochau und Ursula Goltz
Institut für Programmierung und Reaktive Systeme
TU Braunschweig
{lochau|goltz}@ips.cs.tu-bs.de

Abstract: Eingebettete Software-Systeme sind heute allgegenwärtig und müssen zukünftig auch in Hinblick auf eine stetig steigende Lebensdauer und Wartbarkeit entworfen werden. Entsprechend müssen diese Systeme zunehmend evolvierbar sein, d.h. sowohl Fähigkeiten zur situationsbedingten Adaption im Betrieb aufweisen, als auch flexibel an sich ändernde Anforderungen anpassbar sein. Um die Komplexität von Auswirkungen von Anpassungen an diesen Systemen beherrschbar zu halten und Erosionseffekte zu verhindern, sind abstrahierende Architektur-Modelle eine wesentliche Grundlage für die Planung und strukturierte Durchführung von Änderungen im gesamten Lebenszyklus. Gleichzeitig kann durch Evolutions-begleitende Architektur-Evaluationen die nachhaltige Erfüllung von Qualitätskriterien wie Echtzeiteigenschaften, Sicherheit und Verfügbarkeit sichergestellt werden. Wir beschreiben die Einbettung von Maßnahmen zur fortgesetzten Qualitätssicherung in einen koevolutionären Betriebs- und Wartungsprozess und die hierfür erforderliche Konsistenz der Architektur-Spezifikation mit dem laufenden System. Der Ansatz wird anhand einer Fallstudie, einer adaptiven Robotersteuerung mit harten Echtzeitanforderungen, illustriert.

1 Einführung

1.1 Architekturen evolvierender Systeme

Die Software ist der zunehmend kritische Faktor für den Erfolg eingebetteter Systeme. Dabei entsteht ein steigender Anteil der Aufwände und Kosten für derartige Systeme durch Wartungs-, Anpassungs- und Weiterentwicklungstätigkeiten, also der *Evolution* der Software [EGG+09, Mas07], die gerade bei eingebetteten Systemen von der Umgebung und dem Anwendungskontext geradezu erzwungen wird [Leh96, Par94].

Der Architektur einer Software kommt zukünftig eine entscheidende Rolle zu, damit ein Software-System evolvierbar ist und bleibt. Dies gilt sowohl für kleinere Anpassungen wie Refactorings [RH06] oder dem Austausch von Komponenten, als auch für „revolutionäre“ Eingriffe, angefangen beim kompletten Reengineering von (Teil-) Systemen bis hin zur Migration ganzer Systeme auf neue Plattformen [Mas07]. Darüber hinaus müssen moderne Systeme zunehmend adaptiv sein, d.h. sich selbständig zur Laufzeit strategisch an neue Bedingungen anpassen können [SG07, SGM09]. Um die Komplexität der Evolution zukünftiger Systeme beherrschbar zu halten, muss bereits im Rahmen des Ent-

wurfs und der Modellierung die Anpassbarkeit als Kriterium bzw. Eigenschaft explizit berücksichtigt werden. Dies kann nicht allein durch den Einsatz moderner Technologien und Paradigmen und der damit verbundenen "Verheißungen", wie z.B. dem Einsatz Service-orientierter Entwurfsprinzipien, erzielt werden. Vielmehr sind neben Domänenspezifischen Lösungsmustern vor allem übergeordnete Prinzipien einer modularen, konfigurierbaren Architektur-Konzeption wesentlich für eine flexible Anpassbarkeit [BCK03].

Gerade an eingebettete Systeme werden aber auch weitere Qualitätsanforderungen gestellt, wie z.B. Sicherheit, Verfügbarkeit und Echtzeitfähigkeit. Im Kontext kontinuierlicher Evolution besteht somit auch die Notwendigkeit, ein fortlaufendes Qualitätsmanagement begleitend zum gesamten Lebenszyklus zu definieren. Dennoch können nicht alle erdenklichen zukünftigen Richtungen möglicher Weiterentwicklungen und Anpassungen beim Entwurf von System-Architekturen antizipiert werden [EGG+09]. Gerade für Systeme, die sich durch eine besonders hohe Langlebigkeit auszeichnen, können zukünftige, Lebenszyklus-übergreifende Entwicklungen nur schwer abgeschätzt werden, wie z.B. sich ändernde Anforderungen, Erosionen in der Umgebung oder der technischen Infrastruktur sowie die Portierung auf andere Plattformen. Methodiken zur planvollen Evolution von Software-Systemen, die auf strukturierten, Architektur-basierten Vorgehensweisen beruhen, müssen geeignete Dokumentationsansätze für die Nachverfolgbarkeit unternommener Anpassungen beinhalten. Dabei muss nicht nur die korrekte Funktionsfähigkeit des Systems gewährleistet bleiben, sondern gleichzeitig die Sicherung weiterer Qualitätseigenschaften im Betrieb möglich bleiben.

In diesem Beitrag beschreiben wir, inwieweit Architektur-Modelle nicht nur für den Entwurf und Implementierung sowie die Planung von Evolutionen eingebetteter Systeme von entscheidender Bedeutung sind, sondern auch die Grundlage für ein Lebenszyklusübergreifendes Anforderungs- und Qualitätsmanagement bilden können. Begleitend zur stetigen Anpassung des Systems kann durch modellbasierte Architektur-Evaluation eine fortlaufende Bewertung und Sicherung zu erfüllender Qualitätskriterien im Betrieb erfolgen. Die kontinuierliche *Koevolution* von Systemspezifikation und Systemimplementierung erfordert die Integration von Methodiken zur Ermittlung konkreter Architekturausprägungen des laufenden Systems in den Wartungsprozess. Somit können Verfahren sowohl für die *proaktive* als auch *reaktive* Planung von Architektur-Entscheidungen und -Evolutionen unter Abwägung aller relevanten Qualitätsziele etabliert werden. Zudem können Anpassungen geeignet dokumentiert und nachverfolgt werden, um die Auswirkungen unternommener Evolutionsschritte in nachfolgende Entscheidungen einfließen zu lassen.

1.2 Qualitätssicherung durch Architektur-Evaluation

Eine frühzeitige Bewertung der zu erwartenden Qualität eines Systems durch Architektur-Evaluation [BCK03] ist ein unverzichtbarer Bestandteil des Entwurfsprozesses für eingebettete Systeme. Wie in [FGB+07] für den Anwendungsbereich automotiver Systeme beschrieben wurde, ermöglicht ein strukturierter Evaluationsansatz die Integration von Bewertungstechniken für sämtliche Qualitätskriterien des zu entwickelnden Systems. Der

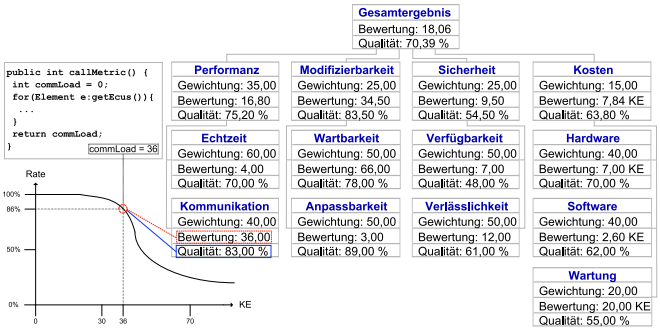


Abbildung 1: Evaluationsstruktur zur Architektur-Bewertung

prinzipielle Aufbau einer Evaluationsstruktur ist exemplarisch in Abbildung 1 dargestellt, für eine detaillierte Diskussion des Ansatzes verweisen auf [LMD+09, LMS+09]. Um zu einer Aussage über die zu erwartende Gesamtqualität des Systems auf Grundlage der Architektur zu gelangen, werden zunächst die je nach Projektzielen relevanten Einzelkriterien durch geeignete *Qualitätsattribute* charakterisiert und bewertet. Für die konkrete Bewertung dieser Kriterien hinsichtlich einer Architektur-Variante werden einheitliche *Bewertungstechniken*, wie z.B. Metriken oder Expertenbefragungen, definiert. Gerade Metriken können automatisiert auswertbar als integraler Bestandteil einer Komponente zur Anpassungsplanung implementiert werden können. Von besonderem Interesse für langlebige Systeme sind Bewertungstechniken zur Modifizierbarkeit der Architektur, bestehend aus a priori festzulegenden Anpassungsszenarien. Um aus den Bewertungsregebnissen der Einzelkriterien eine Gesamtqualität zu ermitteln, werden *Qualitätsraten* zur Normalisierung definiert. Diese ordnen Bewertungsergebnissen je nach Projektzielen und Constraints, z.B. Kostenbudgets oder QoS, eine Güte als Prozentzahl zu. K.O.-Kriterien in Form von harten Grenzwerten für Qualitätskriterien dürfen dabei nicht verletzt werden, um die Realisierbarkeit nicht zu gefährden. Die Integration zu einer Gesamtqualität erfolgt durch Hierarchisierung in Unter- und Oberkriterien mit entsprechenden Gewichtungen gemäß der Priorität der Qualitätsziele. Die Evaluationsstruktur kann dann zur Analyse, Optimierung und Dokumentation von Architektur-Varianten und -Entscheidungen dienen.

Bisher wurde dieser Ansatz zur Bewertung konzeptioneller Architektur-Varianten begleitend zum Entwurf automotiver Steuergerätenetzwerke, also vor der Implementierung und dem Betrieb des Systems, betrachtet. Für eingebettete Systeme mit evolutionsfähigen Architekturen wird zunehmend auch die Notwendigkeit bestehen, den Erhalt von Qualitätseigenschaften nicht nur initial, sondern auch begleitend zu den fortlaufenden Änderungen im Betrieb sicherzustellen.

1.3 Fallstudie: Architektur einer adaptiven Robotersteuerung

Für die nachfolgenden Überlegungen beziehen wir uns exemplarisch auf die „Parallel RObot Software Architecture - eXtended“ (PROSA-X) [SG07] als Beispiel für ein evo-

lutionsfähiges, verteiltes eingebettetes System mit hohen Qualitätsanforderungen. Abb. 2 zeigt die Prosa-X-Architektur zur Integration der Steuerungsmodule von Parallelrobotern. Die Steuerung der verwendeten Parallelkinematiken muss mit hoher Präzision erfolgen sowie harte Anforderungen bezüglich Echtzeit, Zuverlässigkeit und Sicherheit erfüllen. Darüber hinaus muss der Architektur-Aufbau flexible Anpassungen der Komponentenstruktur ermöglichen. Darüber hinaus wurde das System um eine *Self-Manager*-Komponente (Analyse-PC) zur Realisierung von Self*-Eigenschaften erweitert [SG07, SGM09]. Das System ist so in der Lage, ohne Eingriffe von außen adaptiv im Betrieb auf unterschiedliche Situationen, wie z.B. Knotenausfälle auf der Hardware-Plattform, wenn möglich unter Erhalt der Echtzeiteigenschaften zu reagieren.

2 Qualitätssichernde Koevolution von Architekturen eingebetteter Systeme

2.1 Evolution von Software-Systemen

Die Identifikation und Durchführung notwendiger Änderungsmaßnahmen an im Betrieb befindlichen (eingebetteten) Software-Systemen kann auf zwei Arten erfolgen:

Methodische Anpassungen: Eingriffe „von Hand“ als Teil der Systemwartung. Aufwand und Auswirkungen der Anpassungen variieren dabei von einfachen Rekonfigurationen oder dem Austausch von Komponenten, bis zu komplexen Umstrukturierungen der Gesamtarchitektur und dem Deployment. Ein sorgfältig strukturierter Architektur-Entwurf mit stabilen Komponentenschnittstellen und Konfigurationsparametern kann zur Anpassbarkeit von Systemen beitragen. Allerdings erhöht jede Maßnahme zur Auslegung auf potentielle Änderungen zumeist die Komplexität der Architektur. Zudem müssen gerade eingebettete Systeme trotz durchzuführender Anpassungen durchgängig verfügbar sein, was durch eine strategische Abfolge systematischer Anpassungsschritte erreicht werden kann.

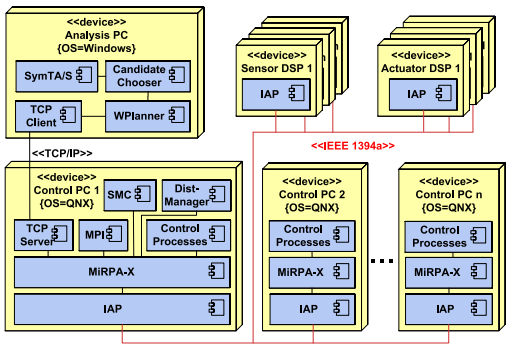


Abbildung 2: Architektur PROSA-X für Robotersteuerungen

Adaption: Die Fähigkeit des Systems, sich selbst zur Laufzeit durch geeignete Strategien an bestimmte Bedingungen oder Kontexte anzupassen, wie z.B. zur Kompensation ausfallender Systemressourcen hinsichtlich Verfügbarkeits-, Sicherheits- und Echtzeitanforderungen. Die Implementierung adaptiven Verhaltens kann z.B. durch parametrisierbare Architektur-Artefakte, also der Spezifikation des Änderungspotentials im Architektur-Modell erfolgen. Im Betrieb kann nun entweder bei Bedarf zwischen zuvor festgelegten Modi umgeschaltet, oder durch Online-Planung je nach Situation eine optimale Parametrisierung ermittelt werden.

Die in Abschnitt 1.3 beschriebene adaptive Architektur PROSA-X verfügt über eine spezielle Komponente (*Self-Manager*), die je nach Betriebszustand durch Rekonfiguration von Betriebsparametern Self*-Eigenschaften realisiert [SG07].

Für beide Ansätze sind Architektur-Modelle die Basis für die Planung und Durchführung der Änderungen. Gerade weil die Architektur das stabilste Element eines Software-Systems über den gesamten Lebenszyklus darstellen sollte, gelten Anpassungen an der Architektur aber als schwierig und kostspielig. Deshalb erfolgen Modifikationen am System häufig in nicht-optimaler Form, d.h. ohne die notwendige Evolution der Architektur, und führen gerade deshalb zur nachhaltigen „Schädigung“ (*Erosion*) der Architektur. Die so kontinuierlich anwachsende Entropie der Systeme führt dann unter anderem zu abnehmender Wartbarkeit und Performanz [Mas07]. Zudem werden durch diese „Wucherungen“ die Abgrenzungen des Systems zur Umgebung zunehmend unklar.

2.2 Architektur-Modellierung und -Rekonstruktion

Der Architektur-Entwurf ist das Bindeglied zwischen der Anforderungsanalyse und dem technischen Design, auf dessen Grundlage neben der korrekten Implementierung der geforderten Funktionalität zugleich Maßnahmen zur Qualitätssicherung in den Entwurfsprozess integrieren werden können [FGB+07, LMD+09, LMS+09].

2.2.1 Architektur-Modelle eingebetteter Systeme

Für Abstraktionskonzepte von Architektur-Spezifikationen, insbesondere zur strukturellen Dekomposition, existieren eine Vielzahl unterschiedlichster Modellierungsansätze und Formalismen [RH06]. Wir beziehen uns vor allem auf Component-Connector-Modelle [BCK03], also die explizite Darstellung grundlegender, häufig rein logischer System-Artefakte und den Abhängigkeiten zwischen diesen Elementen.

Eingebettete Systeme sind zunehmend Software-intensiv und realisieren Steuerungs- und Regelungsaufgaben durch Kommunikations-orientierte Integration verschiedenster, hochspezialisierter Komponenten auf häufig massiv verteilten und heterogen strukturierten Hardware-Plattformen. Standardisierte Schichtenarchitekturen erlauben eine flexible Verteilung von Software- und Hardware-Komponenten, sodass beliebige Architektur-Varianten bezüglich (Wieder-) Verwendung, Anordnung, Verteilung und Verknüpfung dieser Komponenten möglich sind und so großen Spielraum für Optimierungen bieten. Ein Problem

bei der Entwicklung dieser Systeme besteht u.a. im konzeptionellen „Bruch“ beim Übergang von der abstrakten Architektur hin zur möglichst effizienten, d.h. Hardware-nahen Implementierung der Einzelfunktionen. Für die gezielte Planung und Durchführung von Evolutionen langlebiger eingebetteter Systeme bilden Architektur-Modelle eine wichtige Entscheidungsgrundlage, da sie nicht nur Abhängigkeiten zwischen betroffenen Systemteilen aufzeigen, sondern auch durch Modularisierung gezielt die Austauschbarkeit von Teilkomponenten unterstützen.

Architektur-Entwürfe konzentrieren sich in der Regel auf die Spezifikation statischer Strukturen, jedoch ist dies für weitergehende Zwecke, wie z.B. für Systemanalysen zur Anpassung bzw. Rekonfiguration, oft unzureichend, sodass deshalb eine Kombination mit Verhaltensmodellen notwendig ist [RB02]. Um zukünftig beim Entwurf von System-Architekturen potenzielle Evolutionen bereits während der Entwicklung zu berücksichtigen, muss die geplante Variabilität des Systems explizit mitmodelliert werden. Derartige *evolutionsfähige Architekturen* können dann als Grundlage für die Planung und Durchführung von Evolutionsschritten und für die Dokumentation der Evolutionsgeschichte dienen und ermöglichen ein Variabilitätsmanagement vergleichbar mit den Konzepten der *Produktlinien*-Ansätze [CN07]. Durch vordefinierte Variationspunkte werden mögliche Freiheitsgrade für die Anpassbarkeit des Systems bereits als Teil des Architektur-Modells festgelegt. Je nach Änderungsszenario kann die Evolution dann durch Anpassung betroffener Artefakte erfolgen, und das in sämtlichen Phasen des Lebenszyklus. Hierfür sind Architektur-Metamodelle notwendig, die neben der Zuordnung von Variabilitätspunkten auch die Integration von zusätzlichem Wissen über das System erlauben. Auf diese Weise kann sowohl auf Änderungen funktionaler, als auch nicht-funktionaler Anforderungen zielgerichtet reagiert werden.

2.2.2 Architektur-Sichten eingebetteter Systeme

Die Definition von *Architektur-Sichten* trägt beim Entwurf von Architekturen eingebetteter Systeme maßgeblich zur Übersichtlichkeit und Komplexitätsbeherrschung bei. Ausgehend von der domänenspezifischen Anforderungsspezifikation (inkl. Qualitätskriterien) in der Applikationssicht können sowohl die hochspezialisierte, modularisierte Software in der Funktionsarchitektur-Sicht, als auch die anwendungsspezifische Hardware-Plattform (technische Sicht) zunächst getrennt betrachtet werden, und erst im letzten Schritt durch möglichst günstige Verteilung der Software-Komponenten auf vorhandene Recheneinheiten unter Beachtung des resultierenden Kommunikationsaufkommens flexibel in einer System-Architektur integriert werden. Analog zu [RB02] unterscheiden wir im Folgenden zudem zwischen Architektur-Sichten zur Spezifikations- und Laufzeit:

Konzeptionelle Architektur: Die Spezifikation beschreibt abstrakte strukturelle Einheiten des Systems mitsamt ihren Abhängigkeiten. Je nach Modellierungsansatz können konzeptionelle Architekturen unterschiedlichste Artefakte beinhalten. Beispielsweise kann die Struktur des Softwaresystems auf Grundlage einer Component-Connector-Abstraktion beschrieben und durch Schnittstellen- und Prozess-Definitionen verfeinert werden. Beim Entwurf adaptiver Systeme können zudem im Vorfeld gezielt Konfigurationsparameter eingeplant werden.

Konkrete Architektur: Zur Laufzeit repräsentieren die im System agierenden, operationellen Einheiten die aktuelle Architektur-Ausprägung. Dazu gehören Artefakte wie laufende Prozesse (Tasks), deren Kommunikation untereinander sowie Werte von Konfigurationsparametern. Diese Instanzen der Elemente der konzeptionellen Architektur variieren je nach Zustand des Systems im betrachteten Zeitpunkt.

Eine Divergenz zwischen konzeptioneller und konkreter Architektur ist in der Regel unvermeidbar. Spätestens im Betrieb ergeben sich Erosions-bedingte Inkonsistenzen, z.B. durch unzureichend dokumentierte Evolutionen. Darüber hinaus ist insbesondere für Altsysteme in der Regel keine oder eine nur unvollständige Spezifikation der Architektur vorhanden. Für eine strukturierte Planung von Architektur-Evolutionen ist somit eine *Rekonstruktion* der konzeptionellen Architektur auf Grundlage der durch die aktuelle Systemausprägung gegebenen konkreten Architektur notwendig.

Auch für die Architektur-Evaluation können passende Architektur-Sichten definiert werden, die für die jeweiligen Bewertungstechniken relevanten Artefakte enthalten. Für die Modellierung evolvierender Architekturen können zudem Sichten definiert werden, die eine explizite Spezifikation von Variabilitätsparametern unterstützen. Für die Bewertung von Echtzeiteigenschaften in PROSA-X ist z.B. die Verteilung und Kommunikation von Prozessen im System von entscheidender Bedeutung, wobei eine Reihe von Randbedingungen zu beachten sind. Eine Sicht zur Planung und Bewertung von Prozessmigrationen repräsentiert diese Verteilungs- und Kommunikationsstruktur durch eine entsprechende Graphendarstellung [SGM09].

2.2.3 Architektur-Rekonstruktion

Die Kenntnis der Architektur eines Systems ist Grundvoraussetzung für ein tiefer gehendes Verständnis der internen Systemstrukturen zur Analyse und Evolutionsplanung. Für langlebige Systeme mit hohen Anteilen an Legacy-Komponenten liegen allerdings häufig nur unvollständige bzw. inkonsistente Architektur-Modelle vor, in denen Änderungen am System häufig nicht adäquat dokumentiert wurden. Zudem beinhalten Spezifikationen oftmals nur für die Qualitätsbewertung unzureichende statische Aspekte der Systemstruktur. Es gibt eine Vielzahl von Ansätzen, um für ein bestehendes System Architektur-Eigenschaften zu rekonstruieren, über die nachfolgend ein kurzer Überblick gegeben werden soll.

Rekonstruktion: Durch „heuristisches“ Reverse Engineering wird versucht, höhere Abstraktionsebenenen, wie z.B. Component-Connector-Strukturen und andere Pattern, aus dem Quellcode eines bestehenden Software-Systems ohne adäquate Spezifikation zu ermitteln. Eine wesentliche Problematik besteht hier im „mismatch“ zwischen abstrakten Architektur-Konzepten und den Artefakten von Programmiersprachen, also der Definition eines geeigneten *correspondence model* zwischen Code und Architektur-Modell. Bei der Hardware-nahen Programmierung eingebetteter Systeme gestaltet sich die Identifikation von High-Level-Komponenten als besonders schwierig. Ansätze zum Architektur-Recovery sind in der Regel nur (semi-) automatisierbar, bedürfen also geführter Entscheidungen von Expertenseite.

Extraktion: Eine Erweiterung der Rekonstruktionsansätze basiert auf der Einbettung von Architektur-Modellen als *Metawissen* in den Programmcode, wodurch eine präzisere Identifikation und Zuordnung von Architektur-Artefakten möglich wird. Speziell objektorientierte Programmierparadigmen weisen eine enge konzeptionelle Verwandtschaft zu Komponenten-orientierten Architektur-Modellen auf. Die Einbettung von Verhaltensmodellen werden z.B.. [BSG09] und [SVB+03] beschrieben. Weiterhin können hier Programmierumgebungen mit Reflection-Funktionen angeführt werden, von denen zur Laufzeit Informationen über die eigene Programmstruktur abgefragt werden können.

Monitoring: Für eine beliebig detaillierte Ermittlung von dynamischen Informationen können schließlich eigene Observer-Komponenten innerhalb des Systems zum *Tracking* von Laufzeitinformationen, z.B. dem Erstellen von Last- und Kommunikationsprofilen, verwendet werden, was insbesondere ein wesentlicher Bestandteil adaptiver Systeme ist. Auf dieser Grundlage können auch entsprechend aussagekräftige Metriken zur Validierung vorhergehender Qualitätsbewertungen definiert werden.

Rekonstruktion und Extraktion eignen sich vor allem zur Ermittlung statischer Aspekte und Strukturen. In Kombination mit Monitoring-Ansätzen können diese um dynamische Eigenschaften ergänzt werden. In erster Näherung ergeben die genannten Verfahren zunächst konkrete Architektur-Sichten, der Übergang zu einem konzeptionellen Modell bedarf anschließend weiterer Abstraktionsschritte.

2.3 Architektur-Koevolution unter Erhalt von Qualitätseigenschaften

Die Integration der beschriebenen Verfahren erfordert Vorgehensmodelle, die den gesamten Lebenszyklus eingebetteter Systeme vom initialen Entwurf, der Implementierung und Inbetriebnahme, bis zur Wartung und Anpassung im Betrieb, umfassen. Die Konsistenzsicherung zwischen Anforderungsspezifikation, Architektur-Modell und Systemimplementierung verlangt einen umfassenden Ansatz zur Koevolution, d.h. der fortgesetzten Synchronisation des laufenden Systems und der Spezifikation. Neben der Planung und Dokumentation möglicher Evolutionen kann so gleichzeitig die Sicherung der geforderten Qualitätseigenschaften auf Grundlage der aktuellen Architektur-Ausprägung erfolgen. Der prinzipielle Aufbau eines entsprechend zyklischen Vorgehensmodells ist in Abbildung 3 skizziert. Das dargestellte Roundtrip Engineering ist notwendig, um die beiderseitige Synchronisation zwischen Spezifikation und laufendem System sicherzustellen, da (1) Erosionen im laufenden System Anpassungen an der Architektur-Spezifikation erzwingen, und (2) Änderungen der Anforderungen Anpassungen am laufenden System erzwingen. In beiden Fällen bildet die Architektur-Spezifikation die „Brücke“ zwischen Anforderungen und Implementierung: (1) als Entwurf der initialen (konzeptionellen) Systemstruktur, (2) zur modellbasierten Repräsentation der wesentlichen Aspekte des (konkreten) Systemzustandes, und damit (3) als Grundlage für die Planung, Durchführung und Dokumentation von Anpassungen und Adaptionen. Sowohl beim ersten Entwurf, als auch bei der fortgesetzten Evolution der System-Architektur kann die Sicherstellung geforderter Qualitätseigenschaften durch wiederholte Architektur-Evaluation erfolgen.

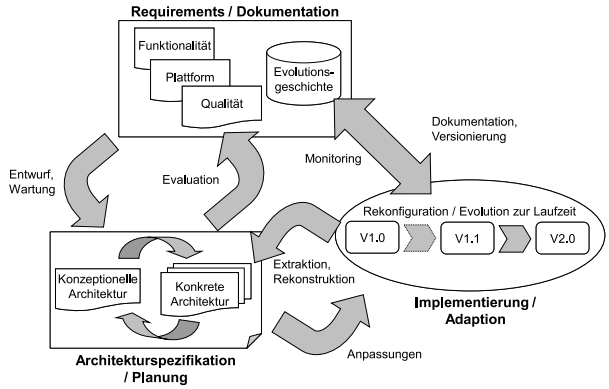


Abbildung 3: Koevolution im gesamten Lebenszyklus

Grundlage bzw. Auslöser von Anpassungen sind sich stetig ändernde Anforderungen an das System bezüglich: (1) der Funktionalität, (2) der (technischen) Plattform, wozu prinzipiell auch die Systemumgebung gezählt werden kann sowie (3) Qualitätsanforderungen. Die Planung dieser Anpassungen auf Grundlage evolvierender Architekturen kann sowohl anhand der konkreten, als auch der konzeptionellen Architektur erfolgen. Erstere kann insbesondere für „leichtgewichtige“ Adaptionen, wie z.B. der Prozessmigration von Bedeutung sein, wohingegegen das konzeptionelle Architektur-Modell für die Planung von Umstrukturierungen größeren Umfangs, wie z.B. der Integration neuer Komponenten, dienen kann. In diesem Zusammenhang kann bereits im Vorfeld sowie Evolutions-begleitend durch das Qualitätskriterium „Modifzierbarkeit“ [BCK03] die Architektur auf potenzielle Anpassungsszenarien hin ausgelegt und bewertet werden. Neben der *globalen* Koevolution zwischen Anforderungen, Architektur-Modellen und der Implementierung, besteht auch die Notwendigkeit, die Konsistenz von konkreter und konzeptioneller Architektur sicherzustellen. Die konkrete Architektur ergibt sich als *Snapshot* aus Systemzustand und Laufzeitartefakten im betrachteten Betriebszeitpunkt und kann, wie in Abschnitt 2.2.3 beschriebenen, ermittelt werden. Ergibt sich für die so ermittelte konkrete Architektur zu einem bestimmten Zeitpunkt, dass sie z.B. durch eine Folge von Adaptionen strukturell zu weit von der konzeptionellen Architektur entfernt ist und nicht mehr als dessen Ausprägung gelten kann, ist auch eine Evolution, d.h. Rekonstruktion der konzeptionellen Architektur notwendig. Umgekehrt führt die Planung von Evolutionen auf Grundlage der konzeptionellen Architektur zu einer Diskrepanz mit der konkreten Architektur, die gerade die für diese Anpassungen notwendigen Änderungen im laufenden System impliziert. In beiden Fällen können die aktualisierten Architektur-Modelle zur erneuten Architektur-Evaluation und somit fortlaufenden Qualitätssicherung verwendet werden, bzw. als Entscheidungsgrundlage in den Planungsprozess von Anpassungen und Adaptionen einbezogen und durch Monitoring-Ansätze ergänzt werden. Die laufenden Anpassungen der Implementierung können je nach *Grad* der hieraus entstehenden Evolution zu unterschiedlich stark ausgeprägten Erosionen der konkreten, und schließlich auch der konzeptionellen Architektur führen. Diese „Versionssprünge“ stellen die wesentlichen Bezugspunkte bei der Dokumentation der Evolutionsgeschichte für das Änderungs-, Anforderungs- und

Qualitätsmanagement langlebiger Systeme dar. Somit sind Architektur-Entscheidungen als Bestandteil der Versionsverwaltung und des *Requirements Tracing* mitsamt der Evaluationsstruktur für die Qualitätskriterien nachverfolgbar und können auch als Anhaltspunkte für spätere Anpassungen erneut herangezogen werden. Vor der Durchführung von Änderungen am System sind neben der Überprüfung von Qualitätskriterien auch entsprechende Techniken zum Erhalt der funktionalen Korrektheit einzusetzen, wie beispielsweise die Kombination kontinuierlicher Refactorings und Tests aus dem Bereich der agilen Methoden [RH06]. Für eine nähere Diskussion der in diesem Bereich bestehenden Probleme und Herausforderungen verweisen wir u.a. auf [RB02].

Evolutionsfähige Architekturen: Für die Planung, Durchführung und Dokumentation von Lebenszyklus-belgeitenden Evolutionen auf Grundlage von Architektur-Modellen bedarf es entsprechender Ansätze, die an dieser Stelle kurz skizziert werden sollen. Wie bereits in [EGG+09] diskutiert wurde, müssen potenzielle Evolutionen integraler Bestandteil der Systemspezifikation werden. Wissen über das System in Form von Konfigurationsparametern für Anpassungen, Annotationen zur Metrikauswertung sowie Instrumentierungen für das Monitoring des Laufzeitverhaltens müssen hierfür als fester Besatandteil des Metamodells vorgesehen werden. Je nach Art und Umfeld des Systems können so entsprechende Profile für Anpassungsklassen definiert werden. Das in [Par94] propagierte *Design for Change* basiert beispielsweise auf der Definition von Änderungsklassen, für die je nach Eintrittswahrscheinlichkeit einer Änderung eine Auswirkungsanalyse für betroffene Artefakte und deren Isolation ermöglicht wird. Die eigentliche „technische“ Durchführung von Anpassungen ist von vielerlei Aspekten abhängig [RH06] und kann sowohl durch eine Abfolge kleiner Einzelschritte, als auch in einen „Big Bang“ erfolgen. So basieren beispielsweise Architektur-Refactorings auf konsistenzerhaltenden Transformationsvorschriften zwischen Artefakten des Architektur-Metamodells und entsprechender Pattern in der Implementierung. Bei der Migration ganzer (Teil-) Systeme sind hingegen vor allem Faktoren wie verwendete Schichten-Architekturen und Virtualisierungs-Komponenten von Bedeutung.

Qualitätssicherung von Evolutionen: Obwohl für die Qualitätsbewertung bestehender evolvierender Systeme im Gegensatz zur Entwurfs-begleitenden Architektur-Evaluation prinzipiell sämtliche Implementierungs- und Laufzeitdetails verfügbar sind, bietet sich auch hier die Verwendung von schwerpunktmäßig auf Architektur-Abstraktionen basierender Bewertungstechniken an, um die Komplexität nicht zu groß werden zu lassen. Bei der Weiterverwendung einer während der Entwurfsphase des Systems für die Qualitätsanforderungen definierten Evaluationsstruktur aus Abbildung 1 zur fortlaufenden Überprüfung der Qualität evolvierender Systeme ergeben sich folgende Fragestellungen:

1. Welche Qualitätskriterien sind im Betrieb im Vergleich zur Entwurfsphase relevant? Kriterien, die harte Randbedingungen für den korrekten Betrieb des Systems darstellen, wie z.B. Sicherheit und Echtzeit, sind im gesamten Lebenszyklus einzuhalten. Kriterien, die schwerpunktmäßig für Entscheidungen während des Entwurfs von Bedeutung sind, wie z.B. Kosten, sind bei der Bewertung von Architektur-Evolutionen hingegen zweitrangig. Ein Spezialfall bilden Kriterien wie Modifzierbarkeit, die nicht nur fortlaufend evaluiert werden sollten, sondern insbesondere durch neue Modifikationsszenarien, die zum Zeitpunkt des Entwurfs noch gar nicht bekannt waren, erweitert werden können. Somit werden

auch die Bestandteile der Evaluationsstruktur selbst eine stetige Evolution erfahren.

2. Welche Anpassungen werden an den Bewertungstechniken vorgenommen? Neben der konzeptionellen Architektur können nun auch Artefakte der konkreten Architektur sowie weitere Anreicherungen um Laufzeitinformationen durch Monitoring für die Präzisierung der vormals nur auf Grundlage des Architektur-Entwurfs definierten Bewertungstechniken herangezogen werden. Zudem können Erfahrungswerte aus der Evolutionsgeschichte in die Bewertung einfließen.

Die methodische Integration der Architektur-Evaluation gemäß Abbildung 3 kann auf zwei Arten erfolgen: (1) reaktiv, d.h. schon beim Entwurf des Systems werden für bestimmte Betriebsmodi feste Anpassungspläne definiert und vor der Anwendung durch Architektur-Evaluation „abgesichert“, um so K.O.-Kriterien auszuschließen, oder (2) proaktiv, d.h. die Planung von Anpassungen erfolgt „online“ und die Qualitätsattribute gehen als Planungsparameter in die Entscheidung ein, um einen möglichst guten Trade-off zu ermitteln. Diese Planungen können als Reaktion auf Änderungen von (funktionalen) Anforderungen oder Laufzeitbedingungen erfolgen sowie zur Optimierung der Qualitätseigenschaften selbst. Der enge Zusammenhang zwischen Evolution und Evaluation wird vor allem an Kriterien wie Modifzierbarkeit deutlich: Diese sind einerseits Gegenstand der Bewertungsszenarien und zugleich Ziel der Anpassungen.

Fallstudie: Rekonfiguration unter Erhalt von Echtzeiteigenschaften

Durch das Einsatzgebiet von PROSA-X werden hohe Anforderungen an die Sicherheit, Verfügbarkeit, Performanz, etc. gestellt. Um insbesondere die Erfüllung harter Echtzeitanforderungen sicherzustellen, sind unter Umständen Anpassungen im laufenden Betrieb notwendig, z.B. bei Änderungen an der Ausführungsplattform (Knotenausfall) oder der Betriebslast (Prozessanzahl). In diesen Fällen kann der *Self-Manager* zur Laufzeit eine neue Konfiguration durch Umverteilung von Prozessen auf Netzwerkknoten ermitteln, um den Erhalt der Echtzeiteigenschaften weiterhin zu gewährleisten. Ein exemplarisches Adaptionsszenario ist in Abbildung 4 dargestellt. Die Planung der Umverteilung erfolgt auf Grundlage einer extrahierten, konkreten Architektur-Sicht, bestehend aus einem durch Monitoring ermittelten Kommunikationsgraphen der laufenden Prozesse. Durch eine Kombination verschiedener Heuristiken zur Graphpartitionierung wird eine möglichst ausgewogene Neuverteilung der Prozesse auf vorhandene Knoten geplant (im Beispiel drei Kontroll-PCs), d.h. (1) eine gleichmäßige Auslastung der einzelnen Prozessorknoten, und (2) eine möglichst geringe Buslast angestrebt. Die Adaption der konkreten Architektur erfolgt durch Prozessmigration im laufenden System. Gleichzeitig kann das angepasste Architektur-Modell zur Prüfung der übrigen Qualitätskriterien herangezogen werden, z.B. die Verletzung von Sicherheitsanforderungen, falls durch die Migration die notwendige Isolation zweier Prozes-

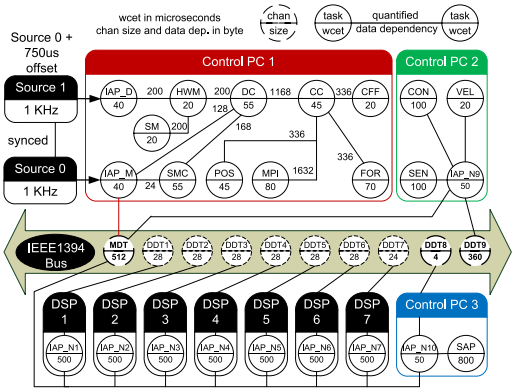


Abbildung 4: Prozess-Migration in PROSA-X

se mit unterschiedlichem Sicherheitslevel aufgehoben würde. Schließlich kann die resultierende Evaluationsstruktur zur Dokumentation der vollzogenen Evolution als Bestandteil des Qualitäts- und Versionsmanagements dienen.

Literatur

[BCK03] L. Bass, P. Clements und R. Kazman. *Software Architecture in Practice*. Addison-Wesley Longman, Amsterdam, 2. Auflage, 2003.

[BSG09] M. Balz, M. Striewe und M. Goedicke. Embedding Behavioral Models into Object-Oriented Source Code. In *Software Engineering*, Seiten 51–62, 2009.

[CN07] P. Clements und L. M. Northrop. *Software Product Lines: Practices and Patterns*. Addison Wesley, 6. Auflage, 2007.

[EGG+09] G. Engels, M. Goedicke, U. Goltz, A. Rausch und R. Reussner. Design for Future - Legacy-Probleme von morgen vermeidbar? In *Informatik Spektrum*. Springer, 2009.

[FGB+07] B. Florentz, U. Goltz, J. Braam, R. Ernst und T. Saul. Architekturevaluation: Qualitätssicherung in frühen Entwicklungsphasen. In *8. Symposium AAET 2007*, Seiten 238 – 248, 2007.

[Leh96] M. Lehman. Laws of Software Evolution revisited. In *Software Process Technology*, Seiten 108–124. 1996.

[LMD+09] M. Lochau, T. Müller, S. Detering, U. Goltz und T. Form. Architektur-Evaluation von AUTOSAR-Systemen: Adaption und Integration. In *Tagungsband des 1. Elektronik automotive congress, München*, 2009.

[LMS+09] M. Lochau, T. Müller, J. Steiner, U. Goltz und T. Form. To appear: Optimierung von AUTOSAR-Systemen durch automatisierte Architektur-Evaluation. In *VDI-Berichte, 14. Internationale Konferenz Elektronik im Kraftfahrzeug*, 2009.

[Mas07] D. Masak. *Legacysoftware: Das lange Leben der Altsysteme*. Springer, 2007.

[Par94] D. L. Parnas. Software Aging. In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, Seiten 279–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[RB02] A. Rausch und M. Broy. Evolutionary Development of Software Architectures. In *Technology for Evolutionary Software Development, a Symposium organised by NATO's Research & Technology Organization (RTO)*, 2002.

[RH06] R. Reussner und W. Hasselbring. *Handbuch der Software-Architektur*. Dpunkt, 2006.

[SG07] J. Steiner und U. Goltz. Engineering Self-Management into Legacy Systems. In *Proceeding for 3rd International Conference on Self-Organization and Autonomous Systems in Computing and Communications (SOAS 2007)*, Jgg. 2 of *System and Information Science Notes*, Seiten 114–117, 2007.

[SGM09] J. Steiner, U. Goltz und J. Maaß. Dynamische Verteilung von Steuerungskomponenten unter Erhalt von Echtzeiteigenschaften. In *6. Paderborner Workshop Entwurf mechatronischer Systeme*, 2009.

[SVB+03] R. Sekar, V. Venkatakrishnan, S. Basu, S. Bhatkar und D. DuVarney. Model-carrying code: A practical approach for safe execution of untrusted applications, 2003.

Using Framework Introspection for a Deep Integration of Domain-Specific Models in Java Applications

Thomas Büchner and Florian Matthes
Fakultät für Informatik
Technische Universität München
{buechner,matthes}@in.tum.de

Abstract:
Domain-specific models and languages are an attractive approach to raise the level of abstraction in software engineering. In this paper, we first analyze and categorize the semantic dependencies that exist between domain-specific models and their generated implementations via frameworks and customization code in a target programming language. We then demonstrate that framework introspection allows a deeper integration of domain-specific models into statically and polymorphically typed object-oriented languages like Java. Using the example of an introspective persistence and query framework for Java, we demonstrate how programmer productivity and software quality can be improved substantially. Since the Java IDE captures the semantic dependencies between the Java application and its embedded domain-specific model(s), it is able to provide programmers with powerful consistency checking, navigation, refactoring, and auto-completion support also for the domain-specific models. Our introspective approach works for whitebox and blackbox frameworks and is particularly suited for the integration of multiple domain-specific models (e.g. data model, interaction model, deployment model) in one application. Due to space limitations, these benefits [2] are not covered in detail in this paper. The paper ends with a discussion of future work on how introspective models can improve the maintenance of long-lived business applications.

1 Introduction

To cope with the increasing complexity and constant change of business applications, new abstractions have been developed which are intended to increase programmer productivity and software quality: Statically and polymorphically typed object-oriented programming languages like Java or C# provide a powerful basic abstraction. Today, they are supported with rich IDEs that provide programmers with powerful consistency checking, navigation, refactoring, and auto-completion support. Based on these languages and tools, frameworks provide architectural abstraction. In order to solve a concrete problem, a framework has to be customized. Modern software systems usually utilize several frameworks, for example for persistence management, web-based interaction or distributed computing. Developers of a complex system have to understand both, the frameworks and their customizations.

Model-driven development tries to raise the level of abstraction of the framework customization process. Customizations are represented as models of a domain-specific lan-

guage. Model-driven hereby means, that there is a transformation between the models and the concrete customization artifacts [6].

As a consequence, there exist artifacts on two different levels of abstraction (framework core and handwritten customizations vs. models). Keeping these artifacts over the lifetime of business applications consistent is the key challenge to be met by model-driven approaches [12]. The central question of this paper is how to better realize and integrate domain-specific languages. We put special emphasis on the issue of integration. Analogous to the approach of Proof Carrying Code [13] we enable Java programs to provide model information through introspection.

This paper is organized as follows: We first review related work on DSLs and roundtrip engineering (Section 2). In Section 3 we analyze and categorize semantic dependencies that exist between domain-specific models and their generated implementations via frameworks and customization code in a target programming language and identify three integration requirements specific to model-driven development. Section 4 gives an overview of our approach to introspective model-driven development (IMDD) based on framework introspection [2] which supports introspective blackbox and whitebox frameworks. Due to space limitations, we only explain in Section 5 how our introspective modeling framework (IMF) provides introspection for whitebox frameworks. A specific example of whitebox framework introspection is the persistence and query framework described in Section 6. Using this example, we illustrate how model-core and model-code integration is achieved. Section 7 compares our approach with popular generative model-driven approaches and highlights its benefits in terms of programmer productivity and software quality. The paper ends with a discussion of future work on how introspective models can facilitate the realization and maintenance of long-lived business applications.

2 Related Work

Specifying the behavior of computer systems using domain-specific abstractions has a long tradition in computer science [3]. One way to implement a DSL is to tailor an existing base language into a DSL. These kinds of DSLs are called *embedded* or *internal* DSLs [5]. An advantage of this approach is that the whole programming environment available to the base language can be reused. The main disadvantage is, that mainstream statically typed object-oriented programming languages are not designed to be syntactically extensible and do not allow for the creation of powerful internal languages.

The prevalent way of implementing a DSL using a statically typed object-oriented language as a base language is building an *external* DSL [11]. As already introduced, we see DSLs as means to specify solutions on a higher level of abstraction in combination with frameworks [4]. Therefore, building an external DSL means building a transformation, which generates customizations from models. This is called *generative* model-driven development. In such a process a metamodel is created, which represents the extension points of the framework to be customized. Additionally, transformation rules which control how to transform the models, have to be created. Based on the metamodel, the framework user

creates a model which solves the problem at hand. This model will then be transformed automatically into customization artefacts.

As already mentioned, the lower level artifacts (framework core, handwritten customizations) have to be maintained as well as the higher level models. This leads to the wish for *roundtrip engineering*, which means that artifacts on both levels of abstraction should be editable and changes to one of them should lead to an immediate synchronization of the affected one. Realizing roundtrip engineering in a generative model-driven process is a challenging task [9].

An approach similar to the one introduced in this paper proposes the use of *Framework-Specific Modeling Languages* [14] that are defined on top of existing object-oriented frameworks. This approach tries to facilitate roundtrip engineering by defining transformations between the levels of abstraction in both directions.

The approach presented in this paper is based on *introspective* frameworks, in which the customization points are annotated explicitly and the customizations follow a constrained programming model. This enables the extraction of models as transient views and makes roundtrip engineering easily achievable.

3 Integration of Domain-Specific Models

In this paper, we call two elements *integrated*, if there exists a semantic dependency between these two, this dependency is stated explicitly [10], and it can be inferred easily by a tool.

Source code of a statically typed programming language is integrated, in that for example the connection between declarations and usages of methods and fields can be inferred at compile-time. Over the last years, the new class of post-IntelliJ-IDEs made excessive use of this property to increase developer productivity and changed the way developers perceive source code [19]. Integration enables features like navigation, search for references, code assist, and refactoring.

As we want to realize domain-specific languages in an *integrated* way, we first identify three integration requirements specific to modeling approaches. As a first requirement, all artifacts related to the DSL should be integrated with the underlying framework core. These frameworks are usually written in an object-oriented programming language. We call this requirement *model-core* integration.

In another typical scenario, concepts defined in a model of a domain-specific language have to be referenced in handwritten code. This occurs because in most cases it is not possible to completely specify a complex system using a declarative DSL. In this case we require the domain-specific language to be integrated with the code of the base language. This is called *model-code* integration in the following. There are two aspects of model-code integration, which differ regarding the location of the handwritten code. In one case, the code which references the model is part of the customizations of the framework. In this case, only parts of the framework can be customized declaratively using a DSL. In

another scenario the code which accesses the model belongs to customizations of another framework. In both cases, the model artifact should be integrated with the handwritten code to improve the consistency of the overall system.

Many problems are solved using several frameworks in cooperation. In such a case, an additional requirement is the integration of different domain-specific languages with each other, which we call *model-model* integration.

Which benefits arise from a modeling approach, which realize these three integration requirements? The main benefit is the automatic checking and assurance of consistency between the artifacts involved. Since the connection between the artifacts in an integrated scenario is stated explicity, tools can help ensuring consistency. This improves the quality of the overall system. Another benefit as already mentioned is tool support for productivity enhancements like navigation, search for references, refactoring, and input assistance.

The prevailing generative model-driven approaches lack integration, since the relationships between the modeling artifacts and the underlying system are not stated explicitly. The underlying reason for this problem is the lack of *symbolic integration* [11] between the artefacts involved. For instance it is not possible to navigate automatically from metamodel artifacts to the extension points of the framework core, reflected by them. The DSL is not integrated with the framework it is expected to customize (no model-core integration). The model-code and model-model integration requirements are not met either by generative model-driven approaches. As a consequence of this lack of integration it takes a lot of manual work to keep all artifacts consistent.

4 Introspective Model-Driven Development

In [1] we proposed a bottom-up approach to model-driven development, which we call *introspective* model-driven development (IMDD). The main idea of IMDD is the construction of frameworks that can be analyzed in order to obtain the metamodel for customizations they define. The process in which the metamodel is retrieved is called *introspection*. The term introspection stems from the latin verb *introspicere*: to look within. Special emphasis should be put on the distinction between introspection and *reflection* in this context. We use both terms as they have been defined by the OMG [16] (see table 1).

In analogy to the definition of reflective, *introspective* describes something that supports introspection. An introspective framework supports introspection in that its metamodel can be examined.

The whole process of introspective model-driven development is schematically shown in Figure 1. The process is divided into the well known core development phase and the application development phase. The first result of the core development phase is an introspective framework. An introspective framework supports introspection by highlighting all declaratively customizable extension points through annotations [17]. This enables the extraction of the metamodel by *metamodel introspection*. It is important to understand, that the metamodel is not an artifact to be created by the framework developer, but rather can be retrieved at any point in time from the framework.

Table 1: Term Definitions

introspection	A style of programming in which a program is able to examine parts of its own definition.
reflection	A style of programming in which a program is able to alter its own execution model. A reflective program can create new classes and modify existing ones in its own execution. Examples of reflection technology are metaobject protocols and callable compilers.
reflective	Describes something that uses or supports reflection.

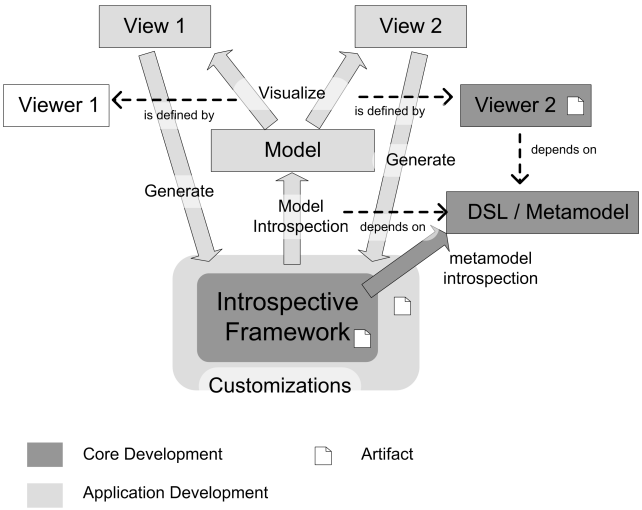


Figure 1: Introspective Model-Driven Software Development

The central artifact of the application development phase are the customizations to be made by the framework user. In IMDD it is possible to analyze these artifacts and to obtain their model representation. This is called *model introspection*. The model is an instance of the retrieved metamodel and can be visualized by different viewers (i.e. visualization tools). We implemented out of the box viewers which can visualize an introspective model in a generic way. In some cases it is desirable to develop special viewers which visualize the model in a specific way (e.g. as a UML model). This will be done by framework developers in the core development phase. The manipulation of the model can be either done by using the views or by manipulating the customization artifacts directly. In both cases an updated customization artifact leads to an updated model and subsequently to an updated view. As a result of this, the model and the views are always synchronized with the actual implementation and can never "lie". This kind of visualization is called *roundtrip visualization* [15].

Generative model-driven development and IMDD differ in the direction the transformation between the model and the customization artifacts takes place. There are similarities between our approach and that of internal DSLs. In both approaches, the models are represented in terms of the base language. The difference comes in how the user of the DSL perceives and manipulates models. Using an internal DSL, the user directly edits statements of the base language, whose syntax is tailored to the particular domain. Because of the inflexibility of statically typed object-oriented programming languages to be tailored syntactically, we have to visualize the customization artifacts on a higher level of abstraction.

The main idea of introspective model-driven development is the direct extraction of the model and the metamodel from the framework artifacts which represent them. There are two categories of frameworks: blackbox frameworks and whitebox frameworks. They differ in the way adaptation takes place. Due to space limitations we only give an overview of how introspection of whitebox frameworks works. Introspective blackbox frameworks are discussed in [1].

5 Whitebox Introspection

The customization of whitebox frameworks is done by providing implementations of abstract classes of the framework core in the base programming language. More specifically, the framework user specifies the desired behavior by implementing methods. These methods are called *hook methods* and represent the extension points of the framework [7]. Regarding introspective whitebox frameworks there are two kinds of hook methods – introspective and non-introspective hook methods. Customization code of introspective hook methods must use a constrained subset of the expressiveness of the base language. We call this subset an *introspective programming model*. Programming using an introspective programming model is of declarative nature and enables the extraction of the model. In contrast, the implementation of a non-introspective hook method can use the full expressiveness of the imperative base language.

The main idea of whitebox introspection is to annotate introspective hook methods in the framework core and to analyze the introspective method implementations. The analysis of the structure of the introspective methods results in the metamodel of the framework core, and the analysis of the method implementations leads to a model of the provided adaptations.

The conceptual idea of whitebox introspection described so far is very generic. To verify the idea, we identified important introspective methods and programming models. In [1], we introduced some basic introspective methods and their programming models. They form a meta-metamodel of whitebox introspection, in that they enable a concrete whitebox framework to draw on these methods. We also implemented generic tool support, which creates an introspective model for introspective whitebox frameworks. Our tools are based on the Eclipse IDE, which is available under an Open Source license and is easily extensible because of its plugin architecture. Specifically, we rely heavily on the Eclipse JDT subproject [20], which provides access to an abstract syntax tree representation of the source code.

On top of Eclipse we built a framework which supports introspection in a general way. This framework is called *Introspective Modeling Framework* – IMF and provides basic abstractions for analyzing source code, representing the models, and visualizing them. Based on IMF there are tools which support blackbox and whitebox introspection with generic visualization. As previously mentioned, it is sometimes desirable to customize the way the introspective model is created and visualized (see Figure 1). This can be done easily, using the existing generic tools as a starting point.

6 An Introspective Persistence and Query Framework

A core requirement of information systems is persistent storage and efficient querying of business objects. Usually this is implemented using a relational database. There is a conceptual gap between relational data and object-oriented modeling of business objects. A *persistence framework* is used to bridge this gap with an object-relational mapping [18].

In this section we explain how to use an introspective whitebox framework for this purpose. The principal idea of our introspective whitebox framework is to explicitly represent the metamodel as introspective Java code. The framework provides abstract classes for all metamodel concepts, which have to be implemented and instantiated to model concrete business objects. For instance, there are abstract classes which can be used to specify persistent properties and persistent relationships between business objects. These abstract classes are introspective, because they have introspective methods. For example, these are value-methods which restrict the implementation to return a value literal or a reference to a final variable [1].

The schema of `Person` objects with a `firstName` property and a *one-to-many* relationship to `Group` objects is defined using inner classes that override (generic) framework classes.

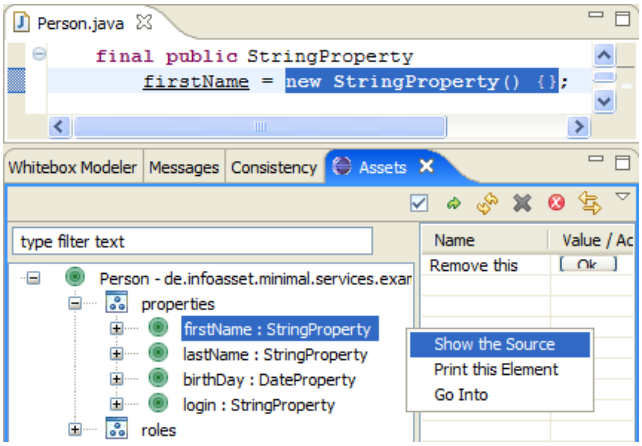


Figure 2: Visualization of the Data Model in a Tree View

```
public class Person extends Asset {
 final StringProperty firstName = new StringProperty() {
   @Override
   int getMaxLength() {
     return 100;
   }};

 final ManyRole<Group> groups = new ManyRole<Group>() {
   @Override
   Role otherRole() {
    return Group.SCHEMA.prototype().members;
   }};
 ...
```

Because this is introspective code, the model can be analyzed and visualized. Figure 2 visualizes the model using a tree view. It is possible to edit the model in this view. On the other hand it is always possible to navigate to the corresponding code, which represents the model attribute. A graphical visualization as a UML class diagram is given in Figure 3. More complex models used in industrial projects based on our IMF are presented in [2].

The following subsections use this example to explain the benefits of our introspective approach in terms of integration.

6.1 Model-Core Integration

The way models are represented in the proposed introspective persistence framework fulfills the requirement of model-core integration as introduced in section 3. This means,

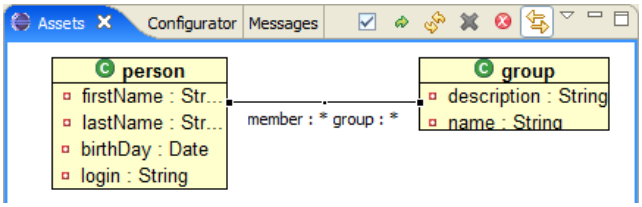


Figure 3: Visualization in UML Class Diagram Notation

that there is an explicit connection between the model attributes and the corresponding hooks of the customized framework. In the example presented above, the persistent model property `firstName` specifies its maximal length to be 100. This is done by overriding a method which is defined in the abstract super class `StringProperty`, which belongs to the framework core. The Java compiler knows about this relationship and additionally it is stated explicity through the use of the `Override` annotation. Therefore, the Java compiler already checks consistency constraints. On the other hand it is easily possible to navigate into the framework core and to find out in which context the attribute maximal length will be used. A post-IntelliJ-IDE like Eclipse can provide programmers with a list of available model attributes .

Another advantage of this model representation is the direct use of the refactoring capabilities of Eclipse to execute refactorings on both the framework and the models in one step.

6.2 Model-Code Integration

Now we can specify and analyze types of business objects at a higher level of abstraction. But how does the programming model look like? Accessing business objects is done using the explicitly represented model. Setting the first name of a person is done as follows:

```
person.firstName.set("Thomas");
```

Another important aspect concerning the programming model is navigation, also known as “traversal”. This means moving from one business object to another along existing association relationships. As already shown, in our persistence framework relationships are defined by instantiating metamodel classes. In particular, this is done specifying the association ends by instantiating objects of type `OneRole` or `ManyRole`. The instantiated model instances are used to navigate relationships in a type-safe way. Accessing all groups a person has a member association with is expressed like this:

```
Iterator<Group> groups = person.groups.getAssets();
```

Another aspect of a persistence framework is querying business objects for certain criteria. In our introspective persistence framework there is a query API to serve that purpose. The

following query retrieves all persons with the first name “Thomas”:

```
Query q = new QueryEquals
   (Person.SCHEMA.prototype().firstName, "Thomas");
Iterator<Person> persons = Person.SCHEMA.queryAssets(q);
```

These examples of the programming model show, that the handwritten customization code is integrated with the data model since it directly refers to the model definition. Also in this case type consistency is checked by the Java compiler and the IDE even for complex nested and join queries.

7 A Comparison with Model-driven Generation of POJOs

To help the reader to better understand the benefits of our approach, we now compare the introspective persistence and query framework with the prevailing approach to object-relational mapping using POJOs [8] (Plain Old Java Objects). Similar benefits arise in other modeling situations (e.g. interaction models and configuration models) as discussed in [2].

This is not exactly an adequate comparison, since representing persistent business objects using POJOs is not a model-driven approach. The model cannot be easily extracted and looked upon on a high level of abstraction. But most generative model-driven approaches using Java technologies generate POJOs [21] [22] and therefore inherit the lack of integration which we will show is inherent to this approach.

In a POJO-based approach persistent business objects are represented as JavaBeans. Properties are represented as private fields with a getter and setter method. Associations are represented using collection fields and getter and setter methods. In both cases, additional metainformation might be provided with annotations or through an external configuration file. The introduced business object `Person` will be represented as follows:

```
public class Person {
  private String firstName;
  @Column(length=100)
  String getFirstName() { return firstName; }
  void setFirstName(String s) { this.firstName = s; }

  private Set<Group> groups;
  Set<Group> getGroups() { return groups; }
  void setGroups(Set<Group> s) { this.groups = s; }
  ...
```

First lets have a look at model-core integration. Model attributes are represented here using annotations [17]. Java annotations are syntactically elegant but only provide very

limited automatic consistency checking capabilities. The scope of an annotation only can be restricted to very generic Java constructs as fields, types, methods, and constructors. The `Column` annotation used in the example could also be applied to the `setGroups` method, which would be an inconsistent modeling. This inconsistency cannot be checked by the Java compiler. Post-IntelliJ-IDEs cannot provide help answering questions about which modeling attributes are available in a specific situation.

Now lets focus on model-code integration. The programming model for accessing properties and navigating associations is straight forward and uses the getter and setter methods. Apart from aesthetic arguments of taste and style both programming models are equivalent and integrated, in that they provide a type-safe way of accessing properties and navigating associations.

But accessing fields and navigating associations is only one part of the overall usage scenarios. Another one is querying, as already introduced. In our introspective persistence framework, the metamodel instances can be referenced integrated to specify queries. This is not possible with a POJO-based persistence framework, because the metamodel instances are represented through Java fields and methods, which are not referenceable. This leads to an unsafe way of defining queries:

```
Criteria crit = session.createCriteria(Person.class);
crit.add(Expression.eq("firstName", "Thomas"));
List<Person> result = crit.list();
```

Unsafe means hereby, that the properties are identified using strings. This is an implicit binding which is not accessible for the compiler and the IDE. Renaming a property or association with a simple refactoring may lead to broken queries, which cannot be found automatically. This is not an issue with our introspective persistence framework.

Another advantage of our introspective modeling approach is, that it is very easy to access metadata at runtime. This enables generic services on business objects, e.g. generic visualization and manipulation. An asset can be asked directly for the model instances which define its schema, which in turn provide access to all available metadata. In a POJO-based persistence framework accessing metadata at runtime has to be done using Java reflection, which is cumbersome. Accessing more sophisticated metadata, like length restrictions as applied via annotations in our example, is even more complicated.

For the sake of completeness, we want to mention the high level modeling and visualization capabilities of our introspective persistence framework again. They come out of the box as an integral part of our framework. For POJO-based frameworks the same kind of tool support is theoretically possible, but the development of the framework core and the tooling do not go hand in hand.

The points mentioned so far concern the experience of the framework user. We believe, that also the framework developer benefits from introspection, because all metadata is directly accessible to the framework core.

8 Conclusions and Future Research

Integration of domain-specific models with the underlying system is a desirable goal in model-driven development because it improves consistency and enables productivity features developers are used to nowadays. Existing generative approaches lack integration because of their character as an external DSL. We have shown, that the proposed approach of introspective model-driven development using whitebox frameworks integrates domain-specific models with the framework core and with handwritten customization code. Referring to the integration requirements proposed in section 3, we have shown that introspective whitebox frameworks enable model-core and model-code integration and solve a key maintenance problem in model-driven software engineering [12].

Furthermore, an introspective software engineering approach increases the maintainability of long-lived business application by explicitely stating adaptations in the system. These adaptations are made on a high level of abstraction using a domain specific language and a programming model that provides abstractions tailored to the problem domain.

In the future, we plan to continue our research on introspective models and address the following issues:

- How can introspective models be enriched to capture more business logic, for example through cascading deletes, derived attributes, declarative constraints on classes and relationships, temporal constraints, security constraints and other business rules?
- How can domain-specific models be visualized and navigated including their links with implementation artifacts?
- How can introspective models be accessed by standard tools used in industry, like configuration management databases (CMDBs), enterprise architecture management tools?

References

[1] Thomas Büchner and Florian Matthes, *Introspective Model-Driven Development*. In Proc. of Third European Workshop on Software Architectures (EWSA 2006), pages 33-49, LNCS 4344, Springer, Nantes, France, 2006.

[2] Thomas Büchner, *Introspektive modellgetriebene Softwareentwicklung*. Dissertation, TU-München, Lehrstuhl für Informatik 19, August 2007

[3] Arie van Deursen, Paul Klint, and Joost Visser *Domain-specific languages: An Annotated Bibliography*. ACM SIGPLAN Notices, vol. 35, pp. 26–36, 2000.

[4] Arie van Deursen *Domain-specific languages versus object-oriented frameworks: A financial engineering case study.*. In Smalltalk and Java in Industry and Academia, STJA'97, pages 35-39. Ilmenau Technical University, 1997.

[5] Paul Hudak *Building Domain-Specific Embedded Languages*. ACM Computing Surveys, vol. 28, pp. 196, 1996.

[6] Markus Völter and Thomas Stahl, *Model-Driven Software Development*. John Wiley & Sons, 2006.

[7] Wolfgang Pree, *Essential Framework Design Patterns*. Object Magazine, vol. 7, pp. 34-37, 1997.

[8] Chris Richardson, *Untangling Enterprise Java*. ACM Queue, vol. 4, pp. 36 - 44, 2006.

[9] Shane Sendall and Jochen Küster, *Taming Model Round-Trip Engineering*. Proceedings of Workshop on Best Practices for Model-Driven Software Development (part of 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications), Vancouver, Canada, 2004.

[10] Martin Fowler, *To Be Explicit*. IEEE Software, vol. 16, pp. 10-15, 2001.

[11] Martin Fowler, *Language Workbenches: The Killer-App for Domain Specific Languages?* http://www.martinfowler.com/ articles/languageWorkbench.html

[12] Gregor Engels, Michael Goedicke, Ursula Goltz, Andreas Rausch, Ralf Reussner, *Design for Future - Legacy-Probleme von morgen vermeidbar?* Informatik-Spektrum, Springer Verlag 2009, DOI 10.1007/s00287-009-0356-3

[13] George C. Necula, *Proof-carrying code: design and implementation* PPDP'00: Proceedings of the 2nd ACM SIGPLAN international conference on Principles and practice of declarative programming, ACM 2000, New York, pp 175-177

[14] Michal Antkiewicz, *Round-Trip Engineering of Framework-Based Software using Framework-Specific Modeling Languages*. Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE'06), 2006.

[15] Stuart M. Charters, Nigel Thomas, and Malcolm Munro, *The end of the line for Software Visualization?*. VISSOFT 2003: 2nd Annual "DESIGNFEST" on Visualizing Software for Understanding and Analysis, Amsterdam, September 2003.

[16] OMG – Object Management Group, *Common Warehouse Metamodel (CWM), v1.1 – Glossary*. http://www.omg.org/docs /formal/03-03-44.pdf

[17] Joshua Bloch, *JSR 175: A Metadata Facility for the Java Programming Language*. http://www.jcp.org/en/jsr/detail?id=175

[18] Wolfgang Keller, *Mapping Objects to Tables - A Pattern Language*. Conference on Pattern Languages of Programming (EuroPLoP), Irsee, Germany, 1997

[19] Gail C. Murphy, Mik Kersten, and Leah Findlater, *How Are Java Software Developers Using the Eclipse IDE?*. IEEE Software, vol. 23, pp. 76-83, 2006.

[20] Eclipse Foundation, *Eclipse Java Development Tools (JDT) Subproject*. http://www.eclipse.org/jdt/

[21] Witchcraft, http://witchcraft.sourceforge.net

[22] AndroMDA, http://www.andromda.org

Federated Application Lifecycle Management Based on an Open Web Architecture

Florian Matthes, Christian Neubert, Alexander Steinhoff

Lehrstuhl für Informatik 19 (sebis)
Technische Universität München
Boltzmannstr. 3
85748 Garching
{matthes, neubert, steinhof}@in.tum.de

Abstract: In software maintenance one of the most important assets is a coherent, consistent and up-to-date documentation of the application. In practice, this information is often not accessible, outdated, or scattered over a multitude of special-purpose systems. We propose to abandon the idea of a perfect documentation in a central repository, and to apply the successful principles and technologies of the open web architecture to arrive at a federated architecture to capture, connect and query distributed information over the full lifecycle of an application.

The contributions of the paper are a concise problem analysis, an identification of relevant related work in software engineering and lessons learned from open web architectures. Based on these, we highlight how our approach can contribute to substantial quality and productivity improvements in the maintenance phase. The paper concludes with a discussion of research and development issues that need to be addressed to realize and validate such a federated application lifecycle management.

1 Motivation

For long-lived business applications, maintenance efforts and expenses exceed by far the costs of the initial development [Leh91]. As indicated in Figure 1, during the full lifecycle of a large business application numerous stakeholders with different roles carry out highly-specialized activities that generate and use digital artifacts of various types which are managed using problem-specific tools and repositories. If one looks at the system maintenance and evolution phase, change requests and bug reports are generated which are managed using tools like *ClearQuest* or *Bugzilla*. To assess the change requests and to resolve bugs, a deep understanding of the software application and its past evolution is required. Artifacts generated by other activities and managed in other tools and repositories (e.g. the Rational Software Architect or a wiki) have to be consulted and connected to gain this understanding. As an example, an explanation of the rationale for a design decision helps to choose evolution options consistent with the initial design.

However, even if the system was initially well documented, the quality of the documentation degrades over time since new or changing requirements lead to an evolution of the

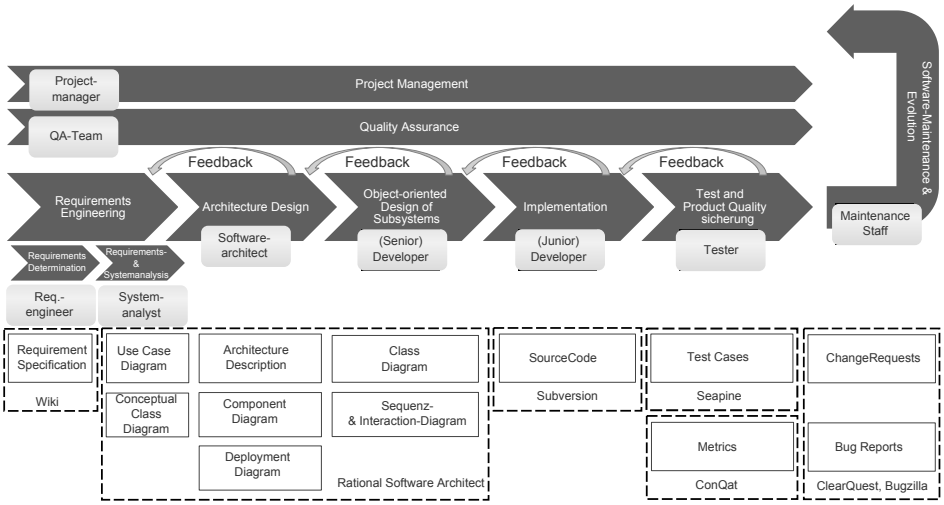


Figure 1: Activities, stakeholders, artifacts and tools in the application lifecycle

implementation and the associated documentation is often not updated accordingly, making it a useless and dead artifact in the long run [FRJ09]. To prevent this, it is necessary to detect anomalies and inconsistencies (e.g., invalid references) as early as possible and thereby keep the efforts for further documentation updates low.

Our paper is structured as follows: Section 2 analyzes the problems related to software maintenance and documentation in more detail. In Section 3 we briefly cover related work in software engineering addressing these problems. In Section 4 we outline web technologies inspiring our approach of a network of federated systems with web-based interfaces which is then presented in Section 5. In Section 6 we identify further research issues deduced from the suggested open distributed architecture and conclude with Section 7.

2 Problem Statement

Availability, completeness, quality and a holistic view of all artifacts containing documentation of the software, especially of its architecture, are critical for the success and efficiency of software maintenance activities. In the following, we identify potential factors which constrain these activities and may compromise their success.

In the literature various models can be found describing the activities and tasks during the software development process, e.g. the Waterfall Model [Boe76] and the V-Modell XT [RB08]. All of these approaches have in common that different stakeholders contribute to the project during different phases of the development. Stakeholders involved in earlier phases of the process (e.g., requirements engineering, design, implementation) often either are not available, when the software is maintained, or do not remember details of the documentation artifacts. Their knowledge is only present in those artifacts. Furthermore,

different views and different understandings of the participating stakeholders may lead to inconsistencies [FRJ09].

For the above mentioned reasons, software documentation artifacts are especially important for the software maintenance activities [Leh91]. A common problem is that these artifacts are either incomplete or do not exist at all. This is often caused by tight and short-term project plans and a limited budget [Par94]. Since these are very common and fundamental problems, the maintenance staff usually have to be content with the available artifacts created during the execution of the particular development phases and have to make best use of those resources. Most of them are strongly interrelated and have an architectural documentation character. For instance, the software architecture itself is documented in an object-oriented model, the description of software components and their dependencies are captured in textual documents (e.g., wiki pages), the architecture is realized by source code elements, which are coupled with the object-oriented models, the component descriptions and the source code documentation. To create and manage these various kinds of artifacts, a large set of highly specialized tools is utilized. Purposes of these tools include but are not limited to requirements management, bug tracking, change request management, version control for source code, workflow management, creation of graphical architecture and design models, build and release management or testing. Additionally, important information resides directly in the source code as inline comments.

A system evolves over time responding to changes in its environment, requirements and the implementation technologies [RLL98]. Thereby different version of the artifacts emerge. So it is challenging to keep all artifacts up-to-date and synchronized with regard to these versions. For instance, assume that the architecture of a software component is documented on a wiki-page. For all public interfaces of this component hyperlinks to the Java-Interface-Files are provided. The versions of those files are stored in a source code repository. If the signature of an interface needs to be modified, e.g., caused by a change request, and thus the version of the file within the repository changes, the reference in the documentation-wiki might need to be updated with regard to this version.

When considering artifacts during maintenance activities, it is helpful to have references available to other resources relevant for the current context. Unfortunately, artifacts are often separated and disconnected. Links between artifacts of different tools cannot be created because of incompatible APIs, heterogeneous protocols and different storage formats. Moreover the documentation artifacts cannot be found, because no unified search over all content is provided. To bridge these gaps between distributed tool-specific repositories, various vendors try to consolidate functionality from other tools in a single uniform platform and provide APIs to enable data exchange. This endeavor is challenging, because of the resulting system complexity, the enormous integration effort, and the lack of generally accepted interchange formats and protocols [Wie09].

In the following sections we present a web-based approach to provide solutions for some of these problems.

3 Related Work

In [WTA+08] Weber et. al. describe their approach of using Web 2.0 technologies to manage the information involved in the software development process. They introduce the concept of a central Wiki-based tool called a Software Organization Platform (SOP). Focusing on small and medium sized organizations (SME) their goal is to better leverage the collective knowledge associated with a single project as well as knowledge about processes that can be reused in later projects.

The Jazz initiative by IBM Rational Software aims at supporting team collaboration by the information integration and at providing guidance for the development process [Fro07]. The initiative offers an IDE-centered approach backed by a server-side central data repository. Further, it allows the integration of third-party tools but requires them to conform to certain web-based interfaces to access the Jazz services. Complementing the individual tool functionalities, there is a set of generic cross-tool capabilities like querying, content discovery or process administration provided by the Jazz platform.

Rooting in the experiences with the Jazz platform, the Open Services for Lifecycle Collaboration (OSLC) community aims at establishing common, less restrictive, web-based standards among software engineering tools [Wie09]. Inspired by the internet, the community is working on shared protocols and formats allowing independent tools to connect and easily access contents. This is very close to our approach, presented in Section 5, but differs in so far that we focus on opportunities not requiring universal standards beyond those currently established in the web, although we share the same long-term vision.

The SYSIPHUS project focuses on collaboration in geographically distributed teams. Its goal is to capture knowledge as a side effect of development and structuring it for long-term use [BDW06]. It provides a uniform framework for models and collaboration artifacts supporting the awareness of relevant stakeholders and the traceability of requirements. All artifacts reside in a single shared repository, which can be accessed through a variety of tools including a web-based client providing a hypertext view of the contents.

4 Web Technology – State of the Art and Trends

In this section, starting from the characteristics of hypertext systems, we outline which fundamental kinds of tools and applications have emerged in the past as well as current trends that we consider relevant in a network of federated software engineering tools.

4.1 General characteristics and trends of open hypertext systems

The most significant characteristic of an open hypertext system like the world wide web is the fact that from within one document it can be easily linked to any other document in the system. This is true for human readable documents as well as for more structured

ones interpretable by machines [BL06]. Furthermore, the uniformity of the presentation format allows the navigation in the complete system using a single tool – namely a web browser. However, the term *document* is misleading in so far that it suggests only static content is considered. Instead we are meanwhile used to seeing dynamically generated content everywhere on the web. Today there is even a clear trend towards complete desktop applications – meaning they were traditionally perceived as desktop applications – like email clients and word processors moving into the browser, facilitated by frameworks as for example *Eclipse RAP*[1]. We expect that even integrated development environments (IDEs) will be realized as web applications soon which would be particularly interesting for our approach (see Section 5).

4.2 Indexing and search

The most common approach to retrieve relevant documents on the web is full text search over all accessible documents via search engines like Google. One important aspect here is that not only the text contents of the documents can be considered but also the structure of the links between them. Additionally there exist so-called meta-search engines [MYLL02] which combine the results of other search engines and thereby intend to improve the relevance of the retrieved documents.

Manual building of structured catalogues of the available web content is on the one hand made difficult by the sheer amount of documents, on the other hand, it is infeasible to keep track of the changes and new additions. However, companies like yahoo offer a hierarchical catalogue of the most popular websites and of course for particular topics there exist special directories of relevant resources.

A recent phenomenon are services allowing users to upload their personal bookmarks of interesting sites and share them with others – known as *social bookmarking* [Neu07]. The idea is that users assigns labels – also called *tags* – to the bookmarks for personal organization. Additional value is generated by aggregating the tags of all users and thus creating a repository of URLs categorized by these tags and ranked by popularity. Furthermore, it is possible to subscribe to certain tags and that way effectively monitor new additions to the repository – filtered according to personal interests.

4.3 Content syndication and mashups

Making the contents of websites available for use in the context of external applications is known as *content syndication*. The data providers make part or all of their data available mainly to increase their reach. Subscribers have the opportunity to enrich their sites by providing the user with more relevant information and thus enhance their attractiveness. Common means for data exchange in this context are XML-based formats and protocols like RSS and ATOM, especially used by news portals and blogs mainly to inform about

[1] `http://www.eclipse.org/rap`. Visited on August 30th 2009

new additions to their contents. However, also non textual information like geographical maps (*Google Maps*[2]), videos (*youtube*[3]) and even mind maps (*mindmeister*[4]) or diagrams (*gliffy*[5]) are made available for use in other applications by the hosting site which usually provide libraries for the scripting language *JavaScript* or Flash plug-ins to embed the content in another site. While these mechanisms effectively determine the presentation of the data, lightweight and mostly XML-based RESTful interfaces, i.e., conforming to the REST architectural style [Fie00], allow to retrieve and modify the available data in an abstract format.

Content syndication enables a new class of applications, so-called *mashups*. These applications combine or enrich the contents provided by other applications and thereby add additional value for the user. A typical example is integrating small advertisements or photos with geographical maps (e.g., *panoramio*[6]).

4.4 Advances in browser technology

In addition to embedding functionality directly into the document by the hosting site, in recent years the trend can be identified that the browser itself can be enhanced with additional functionality. Most browsers offer a plug-in interface allowing developers to create arbitrary kinds of feature-rich add-ons realized for example as additional toolbars or menu items. Minor extensions can also come in the form of small portions of JavaScript code stored in and triggered as a browser bookmark, so-called *bookmarklets*.

Applications reach from generic tools for example for quick look-up of words in a dictionary or filtering advertisements out of the displayed page to functionality that is related to specific services as social bookmarking sites or social networks. A good example showing how far the integration of the contents of these specific services with arbitrary documents on other sites can go is *diigo*[7], a social bookmarking website. Diigo offers a browser extension in form of a toolbar which allows the user to place sticky notes and comments on any page, highlight text for future reference and even start discussions with other users directly in the context of any piece of text or resource on the page.

Modifying the contents of a page by filtering out parts of it or adding further information is also known as *augmented browsing* or *client-side web personalization* [AV08]. Moreover, there are browser extensions specializing on changing web pages via scripts every time before they are loaded and thus forming the basis for other plug-ins making use of this functionality. A popular such extension for the Firefox browser is *Greasemonkey*[8].

[2] `http://maps.google.com`. Visited on August 30th 2009
[3] `http://www.youtube.com`. Visited on August 30th 2009
[4] `http://www.mindmeister.com`. Visited on August 30th 2009
[5] `http://www.gliffy.com`. Visited on August 30th 2009
[6] `http://www.panoramio.com`. Visited on August 30th 2009
[7] `http://www.diigo.com`. Visited on August 30th 2009
[8] `https://addons.mozilla.org/en-US/firefox/addon/748`. Visited on August 30th 2009

5 Improving software maintenance using a web-based architecture

We consider the following kinds of interfaces elementary for software engineering tools for effectively being part of a federated network that facilitates the software maintenance process. Our requirements are very similar to those described by the OSLC community (see Section 3):

1. Every resource that is managed by or resides in one of the tools is accessible via a distinct and persistent URL. On the one hand, this URL serves as an identifier for the respective resource and on the other hand the document behind the URL provides some human readable information about it.
2. A tool provides information about recent updates of its content base via RSS feeds. It is possible to filter these feeds according to the internal structure of the content base. For example, if a tool organizes artifacts in a hierarchical folder structure, it is possible to subscribe to changes in a particular subtree of this hierarchy.
3. Extended access to the contained artifacts is enabled by lightweight RESTful APIs. This extended access includes listing all artifacts, search for them by particular properties and optionally modification.

Fulfilling the first requirement can be considered the absolute minimum for a tool for effectively being part of a federated network, while the last is not likely to be found in all tools in the network. In contrast to OSLC, we do not require shared resource formats among the tools. However, this issue will be covered later in Section 6.

Figure 2 illustrates how additional services make use of the interfaces provided by basic tools while providing the same interfaces themselves. Although it is not explicitly displayed in the figure for the sake of clarity, tools on the same level are allowed to use the interfaces of each other – for example to provide direct links to related resources.

Inspired by applications which have proven successful on the web (see Section 4), in the following, we describe some opportunities for additional services that build on the interfaces we suggested above.

5.1 Discovering relevant content

Perhaps the most important benefit of the outlined approach is that it allows indexing the resources contained by all tools effectively enabling a full-text search of all contents. However, an additional requirement for the searched systems is that they not only provide URLs for each resource, but these resources are at least indirectly reachable via HTML-links starting from one dedicated page of each system.

In contrast to automatic indexing, it is possible to build systems allowing users to bookmark certain resources and label them with tags, effectively categorizing the contents. This

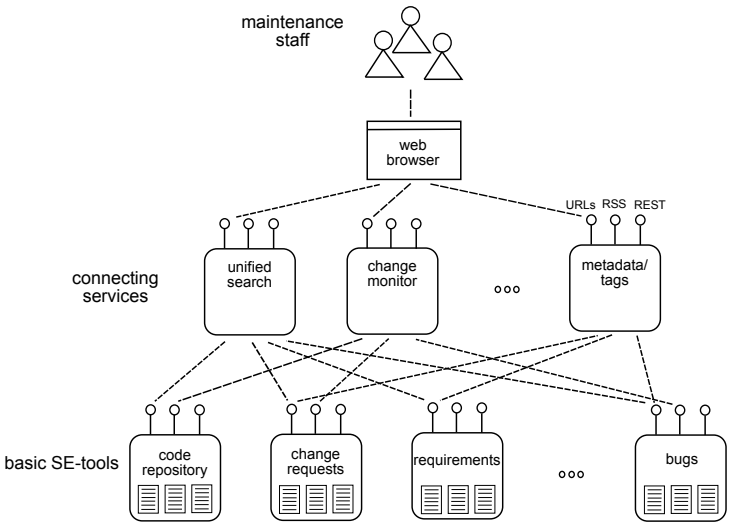


Figure 2: Additional connecting services built on simple web interfaces

can be seen as the equivalent of social bookmarking (see Section 4.2) in the context of software documentation artifacts. Since the user base is rather limited, not all advantages of this approach may unfold. However, we consider this simple and lightweight categorization of resources residing in separate systems a substantial advantage, since in addition to its usefulness for content discovery it can serve as the basis for additional services described later in this section.

5.2 Connecting contents of separate systems

The most fundamental link between two resources of web-based tools is apparently an HTML-link from within the representation of one resource to the URL of another one. For example in the context of a bug report it is possible to reference the documentation of the affected components or a processed change request that possibly caused the bug.

Another option is to directly embed the views on artifacts of one system in the context of documents residing in another system. This can be established through content elements like images – being static or generated upon each HTTP-request – or special Flash plug-ins provided by the system containing the artifacts to embed. For instance the online diagram software gliffy (see Section 4.3) allows to embed views on its diagrams as images into any HTML-document. This way – making manual exports of graphical representations or updating of links obsolete – it is easier to keep different separate documentation artifacts consistent with each other.

When linking or embedding contents directly within a document is not possible or appropriate, another possibility is to connect contents on the client side by enhancing the

presentation in the browser. The technologies presented in Section 4.4 allow for example parsing of a documents content each time before it is displayed and adding a link to Javadoc class documentation to any word matching the respective class name.

Another opportunity for bringing contents together on the client side is to make use of a central bookmarking and tagging application – as mentioned above – and display links to related resources in a browser sidebar. The relation of the documents would in this case be established via the tags of the current document and matching tags of other resources. Tags in this context can be considered not only classifiers but rather implicit links. By applying the same tag, as for instance the name of a component in the application architecture, to several resources, a relation between these resources is established without directly manipulating them.

5.3 Awareness

If changes are made to related pieces of information in separate places by different people, it is very hard to keep this information consistent if there is no mechanism for monitoring these changes in order to react if necessary. These inconsistencies always bear the risks of errors that are introduced because wrong assumptions are made by the developers resulting from outdated documentation. Therefore, persons with particular responsibilities concerning the maintained application are in need of tools keeping them informed of changes with regard to these responsibilities, whether those relate to certain parts of the application or global aspects as usability and security.

Content syndication mechanisms like RSS can help to meet these information needs by allowing systems to provide web feeds about changes to the resources they contain. Since we demand that the feeds provided can be filtered, it is possible to subscribe to task-specific subsets of the resources of each system. However, we only required that the native filtering provided by a tool resembles the respective internal organization of its resources which generally cannot accommodate any appropriate view on the contents. Here it can be useful to rely on tags attached to the URLs of the resources for filtering instead. For aggregating the various feeds, one option is to use special desktop applications like feed readers, email clients or a web browser capable of viewing RSS feeds. Additionally, the feeds can be viewed using special tools automatically aggregating a configurable selection of feeds on a website, effectively generating a dashboard that contains the current information about changes to content, relevant for a particular user.

5.4 Connecting to external knowledge

During software maintenance, especially when trying to understand existing code, familiarity with the application domain plays an important role [SV98]. However, software developers often have insufficient knowledge of the application domain, so being able to access information concerning the meaning of particular concepts is important for them

when editing a program. Additionally, it might be necessary to identify persons having the respective expertise. Provided that at least part of this information is stored in e.g. an enterprise wiki, a web-based architecture makes it easy to connect to this content in a way that is similar to those suggested above. The connection and exchange of knowledge can even cross the boundaries of different organizations, for example by leveraging the knowledge provided by projects like *The Open Model Initiative* [KSF06], a current effort to collaboratively create and refine conceptual models in a public process.

6 Research issues in federated application lifecycle management

Keeping track of different versions of artifacts becomes a serious issue when links exist between them, especially when they reside in separate systems having different versioning mechanisms. For example for each link it should be possible to make the distinction whether it points to a specific version of a resource or if it has to be adjusted when a new version of the resource is created. Considerable efforts have to be made to motivate and establish common standards among tools regarding the traceability of changes and the notion and representation of versions.

Having no central system but a variety of federated tools, the identification and authorization of users becomes another important challenge. On the one hand, it is desirable that the identity of a user does not have to be managed for each individual system, on the other hand, a user's roles and access rights have to be present when a resource is accessed locally. Decentralized authentication standards like OpenID[9] provide a solution for part of this problem. However, since a systems can contain links and metadata referring to resources of another system, more complicated problems arise that require further research: One example is that a user having no rights for a particular resource must not be granted access to meta information of this resource residing in another system.

While providing a distinct identification mechanism for resources and facilitating the discovery of related content items is a major benefit of the presented approach, for many tasks this is not sufficient. When for example searching for contents that were created or modified at a particular date or by a particular author, the access to structured metadata of the resources is required. To accomplish this, the systems in the network have to expose at least part of their internal schema. Further, the corresponding concepts of the various models have to be identified and mapped onto each other to make use of the provided schema information [Ste04]. Thereby we distinguish two types of schemas. On the one hand, data schemas provided by tools for specific purposes, e.g., the definition of requirements. An advantage of those schemas is that the concepts and their relationships are predefined and static. Therefore, mappings and data exchange protocols between multiple tools can be created in advance. On the other hand, some tools, e.g., wikis, are not intended to address a specific purpose. However, some of them allow the definition of metadata by adding semantic annotations to the contents. In this case the data schema and the formal definition of the concepts and their relationships is emerging bottom-up over time. Because of the

[9] `http://www.openid.net`. Visited on August 30th 2009

volatility of those schemas, a predefined resource mapping is infeasible. Yet, technologies like the RDF (Resource Description Framework[10]) provide means for dynamically making semantic information available to other applications. However, since we do not expect a single data interchange format and semantic representation for resources among all tools in the future, it remains challenging to combine data from various schemas to provide services like global search and querying, particularly if some schemas evolve dynamically.

As mentioned above, tags can serve as a generic tool for connecting resources of different systems. While research has been done on large tagging systems and their potential (e.g., [GH06] or [HKGM08]), tagging in a comparatively small environment like a software project is not well studied yet. An interesting research question in this context is for example in how far tagging can contribute to establishing a shared vocabulary of technical terms as well as concepts of the application domain among the developers. Additionally, solutions have to be found to combine tagging functionality of different systems which is not trivial because they can differ in various ways like for instance the possibility to assign weights or visibility restrictions to tags. Furthermore, the identification of synonymical tags among separate systems has to be accomplished. For this problem the mapping of tags to URLs – particularly wiki pages [HSB07] – might provide a solution.

7 Conclusion

Software maintenance is a complex and costly task relying heavily on the access to up-to-date documentation of the maintained system. To better accommodate this information need, we suggested that all tools being used during the application lifecycle offer certain web-based interfaces that allow them to form a federated network making all created documentation artifacts accessible in a hypertext format. Inspired by current trends and successful applications on the web, we demonstrated the opportunities the approach offers for the field of application lifecycle management. Finally, we identified the particular challenges of our approach and topics for future research, respectively.

References

[AV08] Anupriya Ankolekar und Denny Vrandečić. Kalpana - enabling client-side web personalization. In *Proceedings of the nineteenth ACM conference on Hypertext and hypermedia*, 2008.

[BDW06] B. Bruegge, A. H. Dutoit und T. Wolf. Sysiphus: Enabling informal collaboration in global software development. In *IEEE International Conference on Global Software Engineering (ICGSE'06)*, 2006.

[BL06] Tim Berners-Lee. Linked Data. Website, July 2006. Available online at `http://www.w3.org/DesignIssues/LinkedData.html`; visited on August 27th 2009.

[10] `http://www.w3.org/RDF`. Visited on August 30th 2009

[Boe76] Barry W. Boehm. Software Engineering. *IEEE Trans. on Computers*, C-25(12), 1976.

[Fie00] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. Dissertation, University of California, Irvine, 2000.

[FRJ09] Martin Feilkas, Daniel Ratiu und Elmar Jürgens. The Loss of Architectural Knowledge during System Evolution: An Industrial Case Study. In *Proceedings of the 17th International Conference on Program Comprehension*, 2009.

[Fro07] Randall Frost. Jazz and the Eclipse Way of Collaboration. *IEEE Software*, 24(6), 2007.

[GH06] Scott A. Golder und Bernardo A. Huberman. Usage patterns of collaborative tagging systems. *Journal of Information Science*, 32(2), 2006.

[HKGM08] Paul Heymann, Georgia Koutrika und Hector Garcia-Molina. Can social bookmarking improve web search? In *WSDM '08: Proceedings of the International Conference on Web Search and Web Data Mining*, New York, NY, USA, February 2008. ACM.

[HSB07] Martin Hepp, Katharina Siorpaes und Daniel Bachlechner. Harvesting Wiki Consensus: Using Wikipedia Entries as Vocabulary for Knowledge Management. *IEEE Internet Computing*, 11(5), 2007.

[KSF06] Stefan Koch, Stefan Strecker und Ulrich Frank. Conceptual Modelling as a New Entry in the Bazaar: The Open Model Approach. In *Proceedings of The Second International Conference on Open Source Systems*, Berlin, 2006. Springer.

[Leh91] Franz Lehner. *Softwarewartung – Management, Organisation und methodische Unterstützung*. Carl Hanser Verlag, München, Wien, 1991.

[MYLL02] Weiyi Meng, Clement Yu und King Liu-Lup. Building efficient and effective metasearch engines. *ACM Computing Surveys*, 34(1), 2002.

[Neu07] Dagmar Neuberger. *Social Bookmarking – Entwicklungen, Geschäftsmodelle und Einsatzmöglichkeiten*. VDM Verlag, January 2007.

[Par94] David Lorge Parnas. Software Aging. In *ICSE*, 1994.

[RB08] Andreas Rausch und Manfred Broy. *Das V-Modell XT : Grundlagen, Erfahrungen und Werkzeuge*. dpunkt.verlag, 2008.

[RLL98] David Rowe, John Leaney und David Lowe. Defining Systems Evolvability - A Taxonomy of Change. In *Conference on Engineering of Computer-Based Systems (ECBS '98)*, Los Alamitos, CA, USA, 1998. IEEE Computer Society.

[Ste04] Ulrike Steffens. *Metadaten-Management für kooperative Anwendungen*. Dissertation, TU Hamburg-Harburg, 2004.

[SV98] Teresa M. Shaft und Iris Vessey. The relevance of application domain knowledge: characterizing the computer program comprehension process. *Journal of Management Information Systems*, 15(1), 1998.

[Wie09] John Wiegand. The Case for Open Services. Website, May 2009. Available online at `http://open-services.net/html/case4oslc.pdf`; visited on August 27th 2009.

[WTA+08] Sebastian Weber, Ludger Thomas, Ove Armbrust, Eric Ras, Jörg Rech, Özgür Uenalan, Martin Wessner, Marcel Linnenfelser und Björn Decker. The Software Organization Platform (SOP): Current Status and Future Vision. In *Proceedings of the 10th International Workshop on Learning Software Organizations (LSO 2008)*, June 2008.

Maintainability and control of long-lived enterprise solutions

Oliver Daute Stefan Conrad

SAP Deutschland AG & Co. KG Heinrich-Heine Universität Düsseldorf
Business Solution Technology Institut für Informatik
Homberger Straße 25 Universitätsstraße 1
40882 Ratingen 40225 Düsseldorf
oliver.daute@sap.com conrad@cs.uni-duesseldorf.de

Abstract: Maintainability of long-lived enterprise solutions is today's top challenge. The constantly increasing functionalities of information technology provided these days, including hardware and software products, leads to giant networked application infrastructures. Business processes glue applications and data sources to enterprise solutions to fulfill business needs. Large firms are more and more dependent on the permanent availability of their enterprise solution. A failure in one part of the landscape can rapidly impair the whole enterprise landscape and harm the business. Maintainability is a critical issue and new approaches are required to keep complex landscapes under control at all times. We introduced the *Real-Time Business Case Database*, RT-BCDB as a basic concept. This paper presents the next reasonable step. *Process Activity Control* describes a steering mechanism for complex enterprise application landscapes, to gain more stability, transparency and visibility of processes activities in order to improve maintainability and to support the system administration in their daily operations.

1 Introduction

More transparency and control inside complex application landscapes is required [KMP08] since concepts like cloud computing [Vo08], IT service management [ITIL], architecture frameworks [TOG09] or service-oriented architecture [SOA06] make it possible to build up giant enterprise application solutions. But mechanisms how to manage the underlying technology have been neglected. Long-lived enterprise solutions are subject to evolution [RB09], obsolescence and replacement. We have to find answers to questions regarding system maintenance, troubleshooting, availability, change management and the integration of new components.

Maintainability and control of complex software solutions is today's top challenge.

This paper presents mechanisms for increasing the maintainability and control of complex application landscapes. *Process Activity Control* (PAC) is the next step after the introduction of the *Real-Time Business Case Database* [Da09].

PAC concentrates on the control of business processes which are currently active within an application landscape. The goal is to avoid indeterminate processing states which can cause further incidents within the application environments.

Most enterprise or service frameworks are focused on business requirements which have improved the design of enterprise solutions significantly but often with too little consideration for the underlying information technology [Ro03]. Operation interests are neglected and little information about how to run a designed enterprise solution can be found. A single business process is able to trigger process activities across the whole landscape, uses different applications, servers or exchanges data. If a server fails or a database system stops its processing, then several business processes can be impaired (Fig. 2). In such situations it is quite difficult to determine the cause or impact on other process activities of the enterprise solution [KMP08].

The challenge for the system administration is to manage these complex application environments and to react as swiftly as possible to incidents [SW08]. Several tools are available to monitor large enterprise landscapes. Some of them just monitor the operating or the application level. Only few tools collect details about business processes and try to control them. But each tools has its own proprietary view to applications. There is no overall concept for heterogeneous application environments available.

Numerous business cases within an enterprise solution make it impossible to be aware of what each single business process does. The system administration must be able to analyze business processing like a technical issue. Thus, business processes are seen as objects which have a run-state and are requesting to run. This eliminates the need to collect information about what a specific business process does. The focus on essential information also standardizes business processes and eliminates the need to identify its run-state in the case of an incident. System administration can react more purposefully and the designers of business processes get detailed information about the behavior of their business cases.

Complexity leads to longevity of software solutions for several reasons; here we need only to think of done investments, cross application dependencies and efforts to exchange software products. This paper discusses the challenges and circumstances in complex enterprise solutions and why it is so difficult to control business processes. We will show how to set up a steering & control mechanism within the application environments and will discuss the benefits for the system management administration.

For the communication with PAC, we will introduce a C*ode of Business Processes*. It is a valuable step forward in defining the frame to identify business processes.

Finally, the use of PAC in collaboration with the RT-BCDB will be presented, preceded by a short introduction to it.

2 Background & Terminology

RT-BCDB stands for *Real-Time Business Case Database* and it is an approach to collecting and providing information about business process activities in heterogeneous application landscapes. RT-BCDB aims to improve the transparency of activities.

The knowledge acquired supports the administration during maintenance activities and is an important source of information for the business designers as well. RT-BCDB supports tasks, like updates, recovery, planning or optimization of the time schedules.

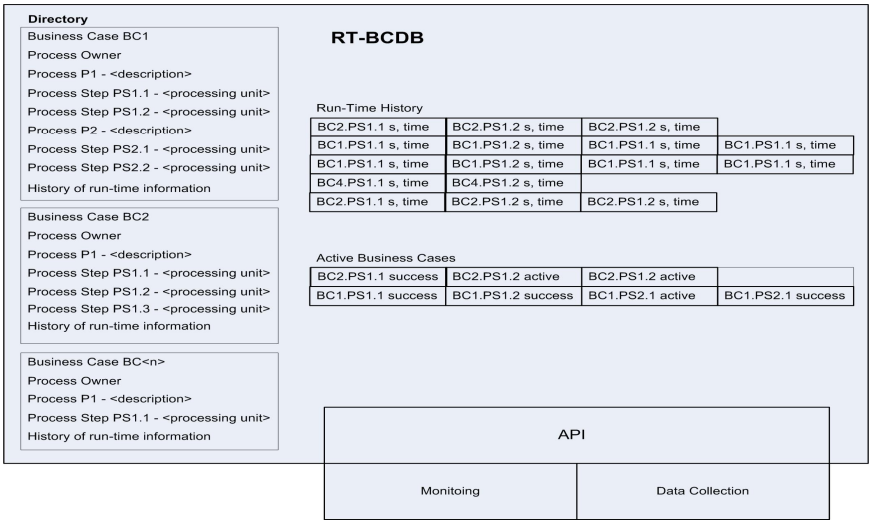


Figure 1: Real-Time Business Case Database

A system failure requires detailed investigations about the impact on business process activities before a system recovery can take place. This means processes which were impaired must be identified and must be included in the recovery process. The data of RT-BCDB is important during the analysis and restoration phase in order to bring back business processing to the latest consistent state. It is also indispensable for monitoring, reporting and changing the landscape.

RT-BCDB stores data about business cases, processes, owners, history of previous processing, execution frequencies, run-time, dependencies, as well as availabilities of processing units and applications. Knowledge about run-states of business processes is important information for maintaining and controlling business processes.

The term *enterprise solution* encapsulates a collaboration environment of hardware and software technologies with the common purpose of providing an infrastructure for business processes. A solution can consist of ERP software, various legacy systems, data warehouses, middleware for exchanging data and connecting software applications. Other expressions are *application landscape* or *application environment*.

Business cases are designed by the business requirements and needs. A business case consists of several processes which can be performed on different systems. Business cases determine the tasks of the customer's enterprise solution.

A *business process* consumes data or provides them and can be triggered by other processes or services. Business processes make use of different applications and data sources across an application landscape with regard to the enterprise needs.

System maintenance activity relates to ongoing tasks on the system administration level. Typical maintenance activities are updates, incident analysis, recovery, replacement and the like. (RT-BCDB supports active or reactive maintenance activities).

3 The Idea

Process Activity Control is required because of the continuously increasing complexity of long-lived enterprise solutions, driven by evolution, availability, growth, business requirements, modern tools and enterprise application framework methods [TOG09]. IT administration has to manage these solutions in any situation. New mechanisms are required to assist them.

The constantly increasing complexity of enterprise solution is the number one cause of cost-intensive system failures [EIU07].

Incidents interrupt business processes while they are performing a task. The malfunction of a processing unit or of an application can cause process activities to fail. Processes need to be restarted in order reach data consistency on business process level.

The idea of PAC is to minimize uncontrolled failure of business processes and reduce the amount of incidents. If problems within the application landscape are already known, e.g. a database stopped processing, then there is no reason for a process to start with the risk of halting in a failure situation. PAC acts proactively and thus avoids disruptions.

PAC also addresses another currently unsolved problem: the start and stop process of a complex application landscape or parts of it. It is still a complex matter to shutdown a single application without the knowledge of dependent business cases and without impairing the business. At the moment, there is no outer control for business cases available. Whenever a shutdown is required, PAC identifies active business processes to stop and will avoid the start of further business cases.

Business processes are triggered or started by different activators. A process can cause different activities across the whole application landscape and exchanges data. Various automated control instances start and stop processes, too. But there is no overall concept for heterogeneous application environments.

The figure (Fig. 2) depicts a well-known situation in application environments without control of processes. Uncontrolled failure of business processes may result in an unknown run-state or data inconsistencies on business level.

From the perspective of a business case, a consistent state requires more than data integrity on database level. Also dependent interfaces or single process steps must be taken into consideration. Those can halt in an inconsistent state anywhere in an application environment.

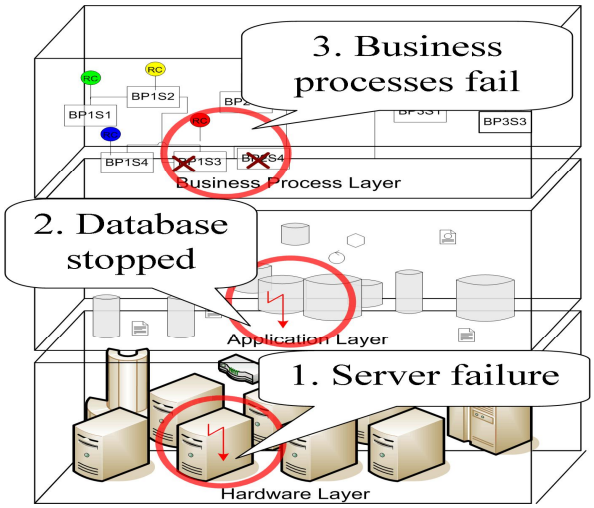


Figure 2: Failure within the application environment

PAC works as a steering mechanism for business processes and is especially valuable in the control of core business processes. To interact with business processes, PAC makes use of RunControl commands. We will discuss these later. PAC is able to collect run-state information of business processes and send them to RT-BCDB.

4 Code of Business Processing

Various situations arise in complex application landscapes because a form of identification for business processes is missing. These are not easy to handle or to overcome in case of incidents. Therefore, identification is required for business processes in the communication with PAC. The Code of Business Processing, CoBP covers some minimum requirements.

Traffic laws are simple and effective. They are necessary to control and steer the traffic within a defined infrastructure, the traffic network. Traffic laws and networks together describe a kind of code of conduct which participants have to accept in order to participate in the traffic. It is an appropriate steering mechanism for a complex environment. This example of a regulating mechanism gives us an idea of what is needed for application landscapes.

We will try to translate some elements of traffic laws and network into a code for business processes.

We propose a *Code of Business Processing* which contains general rules and requirements for using an application environment. The code should only be applied to processes which are of significance for the enterprise solution itself. That means only processes with a high impact in case of a failure have to meet the code. The CoBP consists of rules and requirements:

First CoBP: Each process must have a unique form of identification. This is required to identify and steer a process while it is active. Identification is needed to follow the tracks (footprints) during the processing lifetime and to refer to the process owner. Process IDs are unique across all applications to avoid false identification. A process ID can also be used to classify. The most important business processes have a distinctive ID.

Second CoBP: Each business process must be documented. It must belong to a business case and visualization must exist. A diagram contains the run-states a business process can run in. Procedures must be given for recovery purposes in case of a process failure.

Third CoBP: Each process must have a given priority. On the one hand it is important to decide which process should be given priority and also to determine a sequence of processing. On the other hand it is useful to mark processes regarding the impact on the application environment. The business processes with higher priority must process first, unless PAC decides differently.

Fourth CoBP: The higher a priority is the higher is the charge for a business process. Certainly, accounting and charging for processes is not very often done at customer site. But a process with a high priority does have a significant impact on all other processes that run within that environment.

For our purposes the first two rules are sufficient to control activities. Additional rules can be added. Ideally, communication between processes should always take place in traceable ways. As a result of this our last CoBP follows.

Fifth CoBP: Business Processes should use defined and traceable ways for interaction. This forces the use of known and open interfaces. The CoBP improves the traceability of business processing significantly and supports the maintainability. For existing application landscapes, the 5th CoBP requires reviewing the process interaction.

The rules of CoBP enable better steering of business processes, provide more transparency and thus improve maintainability.

5 Process Activity Control

Process Activity Control is an approach to controlling process activities in complex application environments. PAC will stop further business processing in case of problems occurring. This will prevent business processes running into undefined processing states and avoid further incidents.

PAC has to consider several issues in order to control process activities. A major task is, for instance, determining the function state of business processes, applications and processing units or the functioning of PAC itself. This can be done by RT-BCDB.

RT-BCDB uses agents to collect information about an application landscape. On the hardware and application level, agents search for a specific pattern to determine the function state and availability.

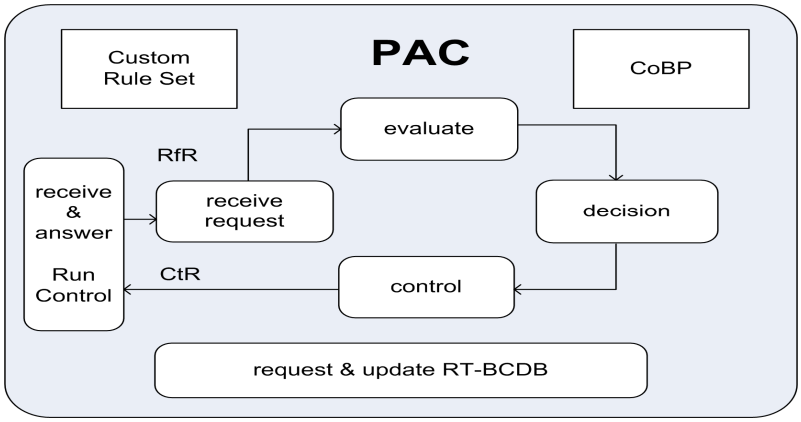


Figure 3: Architecture of PAC

Applications fork operating system processes, which can be monitored as well to identify throughput. A premature termination of an application process on operating systems can point to the failure of an application. The agents inspect the given sources of information and try to identify run-state and availability information. This information is used by PAC to react to current circumstances. PAC will try to avoid any starting or triggering of business processes which will make use of a malfunctioning processing unit or impaired applications.

For smaller software environments information about the availability of hardware and software application is sufficient enough to control and to avoid further incidents. But only few details about business process activities are visible.

For large application landscapes, such as long-lived enterprise solutions, PAC must also be informed of run-states of business processes. This information is provided by the knowledge base of RT-BCDB.

PAC consists of several basic elements: a decision-control mechanism, a Custom Rule Set, the CoBP, an interface to RunControl, and a communication interface to RT-BCDB. The decision-control mechanism is subdivided into four main activities: *receive request, evaluate, decision* and *control*. Each activity has one or more tasks.

Activity '*receive request'*, just receives the *Request for Run* (RfR) in sequence of income. Whenever a business process starts or stops or changes its run-state, then RunControl will send an RfR. The RfR contains the process ID and the state of running.

The activity '*evaluate'*, evaluates the RunControl request against the information stored in RT-BCDB. The run-state table of RT-BCDB always reflects the status of process activities within the application environment. Any known problems with the availability of applications or processing units are taken into consideration.

The '*decision'* process uses CoBP, Custom Rule Set, RT-BCDB and the evaluation of the previous activity. A final decision will be prepared to return a '*Confirmation to Run* (CtR)' or to stop a business process.

The '*control'* activity is the steering process of PAC. It has two functions. The first is to answer the RfR and to send a CtR. In the case of a business process having to be paused, the *control* process waits to send the CtR until problems are solved. The second function is to stop business processes in the case of the application landscape having to be shut down. Vice versa '*control'* enables the start-up of business cases in a predefined sequence, for instance after maintenance activities or after the elimination of incidents.

The *Custom Rule Set* contains rules given for a customer's application landscape. The rule set can contain an alteration of priorities or a list of business cases which have to run with a higher priority. Also preferred processing units can be part of the rule set. Another element is CoBP, which was described previously. Finally, an application interface, which is used to communicate with RT-BCDB, is also part of PAC.

In operation, PAC as a control instance must monitor its own availability. At least two instances of PAC must run within the environment. This is necessary to prevent PAC is becoming *single-point of failure* for the enterprise solution. In normal operation the second instance should be used to answer RfR.

6 Run Control

PAC introduces an extension to *RunControl* [Da09] commands. RunControl commands are used to receive information about processes and required for controlling the progress of their activities. Whenever a business process starts, stops or waits, the RunControl command will send a message with the process ID and the run-state. The information will be stored immediately in the run-state table of RT-BCDB. Due to this, active business cases and their status can be made visible (Fig.4).

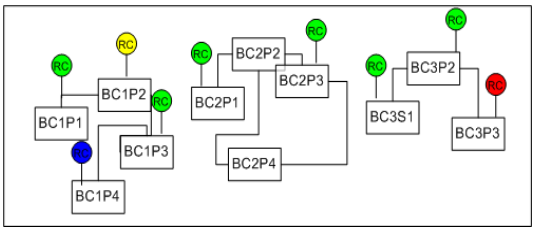


Figure 4: Run-States of active business cases

We adapted the 'RunControl' statement for use with PAC. Instead of sending the run-states information using the agents, this information is send to PAC immediately. PAC forwards the information to RT-BCDB. On the business process layer, PAC takes over the tasks of the agents. The second change is an extension of functionality. The RunControl function waits until it receives a '*Confirmation to Run*' from PAC. To distinguish between the two versions of RunControl statements, we will use **RunControlAC** when using PAC.

```
RunControlAC(BC1.PS1, "active")
call function PurchaseOrder;
if  return code <> 0 then
    call ExceptionHandling;
    RunControlAC(BC1.PS1, "exception")
end
RunControlAC(BC1.PS1, "successful")
```

Figure 5: Example for RunControl in source code or transparent layers

Several options are given to implement RunControl statements (Fig. 5). An option is to insert RunControl statements into the source code [UML] [Sv07]. For existing applications adaptations are possible during migration projects [St06]. Reverse engineering might be an appropriate discipline too. Consequently for the future design of business solutions, applications should be developed with regard to run-state information or the RunControl statements.

Instead of modifying the source code, processes can also be called up from a transparent layer in which functions are embedded in RunControl statements (Fig.6). This option is much easier to implement and to introduce for heterogeneous landscapes.

Certainly, some effort is needed for the implementation of the RunControl. But with the constantly increasing complexity of application landscapes, a mechanism as described is indispensable for keeping a long-lived enterprise solution under control.

7 Improving Maintainability and Control

The aim of our concepts is to gain more transparency of and control over process activities. To increase the overall availability, e.g. by the prevention of cost-intensive incidents, is one aim more to be listed. We will discuss next by means of a simple example, how PAC improves the maintainability of complex application landscapes.

An application stops its processing (Fig. 6). PAC recognizes this problem and stops further processing of business cases which will make use of the application. Here, a single business case is impaired and has to be stopped in order to prevent data inconsistencies. If a standby application is available, PAC can move the processing to it.

In the case of performance bottlenecks, PAC is able to stop a business process in order to prevent a problem from getting worse or shift an RfR to another application unit if possible.

Although users have to wait until the problem is solved, indeterminate processing states will be avoided. These can cause further incidents and additional efforts for solving.

PAC makes use of RT-BCDB knowledge to decide a '*Request for Run*'. If incidents to databases, applications, processing units or business cases are known, then PAC will determine if a '*Request for Run*' will make use of them. The run-state and availability information, stored in RT-BCDB, provides a virtual image of activities within the application infrastructure.

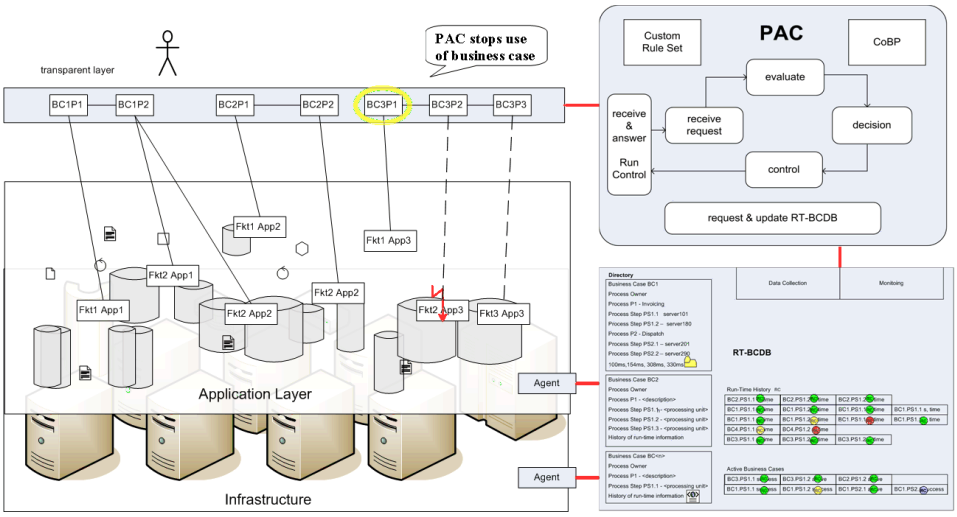


Figure 6: Avoiding indeterminate run-states and inconsistencies

Maintenance tasks like updates or upgrades of the applications also require detailed information about the business cases possibly involved. The constant availability of long-lived enterprise solution requests to maintain the landscape while numerous business processes are running. PAC can prevent process activities while parts of the application landscape are under construction. RT-BCDB provides the dependencies.

How to measure improvements in terms of *Return on Investments*? We will try to answer this question with regard to time, money and quality.

Time: Each incident which was prevented saves time. Identifying the cause of and solution to an incident cost time. Additional time is needed for reporting and documenting the solution, and several persons of different departments are involved. Users are hindered in their work and will lose time. PAC prevents cost-intensive incidents and improves the overall-availability of the enterprise solution.

Money: Costs are incurred by incident handling, software for incident tracking and support staff. Downtimes can cause less productivity and can result in fewer sales. An overall estimation is difficult to provide. In the worst case, especially in the area of institutional banks, an incident can cause bankruptcy within a few days [EIU07].

Quality is often not easy to measure. In enterprise application landscapes quality means availability, reliability, throughput, maintainability and competitiveness. Quality can save money but can increase costs. The introduction of PAC and RT-BCDB requires investments. But more visibility and transparency lead to more purposeful reactions to incidents within the landscape. This saves time.

And fewer incidents, however, increase the quality and the availability of an enterprise solution. We assume that for large environments the investment in regard to the increase in quality will save money in the end. In smaller environments our concept will at least improve quality. We expect that PAC & RT-BCDB reduce the TCO [Ga87], but further investigations are required to determine the quantity and quality of improvements.

8 Evolution and Obsolescence

Long-lived enterprise solutions are subject to evolution, obsolescence and replacement. Typical, for long-lived solutions is that the interaction between software components last longer than the components themselves. Therefore knowledge about activities is essential to maintain the landscape and to support the change management process.

Evolution is driven by new requirements, growth and replacements of hardware or software, including general maintenance. Evolving a long-lived application solution can be difficult. Lack of information about dependencies of process interactions, the way the solution works and the integration of new components are some examples. Moreover the enterprise solution requires close to 100% availability. RT-BCDB is oriented to the information about the processing-unit, business cases and process activities. PAC supports the maintenance process while the enterprise solution is in use.

Innovation can lead step by step to obsolescence of applications of the enterprise solution. Often, so called legacy systems are difficult to exchange for some reasons. Problems are missing documentations, designers left the organization and too little information about the purpose and frequency of usage. Our concepts can explain their usage and improve the replacement. They are able to detect monolithic applications. A constantly decreasing usage can indicate obsolescence.

As we pointed out, most frameworks are focused on business requirements and neglect the maintenance interest. Thus, insufficient transparency in operation and missing control mechanism are the results to name only some of the gaps. Maintainability and availability (incl. avoidance of cost-intensive incidents) are the main really challenging tasks. But also recovery is an interesting issue to ensure data consistency across large application landscapes. Our concepts are steps forward to improving maintainability and control of long-lived software-solutions.

9 Conclusion

The concepts presented support maintenance tasks and the evolution of long-lived enterprise solutions. System administration can react more purposefully. PAC improves the control over business process activities, avoids incidents and makes better utilization of resources. This leads to a higher availability and a better quality of the complex application landscape. RT-BCDB is the knowledge basis and provides transparency; an image of process activities and their dependencies.

Certainly, some work needs to be done to implement such concepts, especially if software applications need to be adapted. But, with the constantly increasing complexity of enterprise solutions, mechanisms as described are indispensable for keeping long-lived enterprise application landscapes maintainable in the future.

References

[Da09] Daute, O.: Introducing Real-Time Business CASE Database, Approach to improving system maintenance of complex application landscapes, ICEIS 11th Conference on Enterprise Information Systems, 2009

[Da04] Daute, O.: Representation of Business Information Flow with an Extension for UML, ICEIS 6th Conference on Enterprise Information Systems, 2004

[EIU07] Economist Intelligence Unit: Coming to grips with IT risk; A report from the Economist Intelligence Unit, White Paper 2007

[Ga87] Gartner Research Group: TCO, Total Cost of Ownership, 1987, Information Technology Research, www.gartner.com

[ITIL] ITIL, IT Infrastructure Library, ITSMF, Information Technology Service Management Forum, www.itsmf.net

[KMP08] Kobbacy, Khairy A. H.; Murthy, D. N. Prabhakar, 2008, Complex System Maintenance Handbook, Springer Series in Reliability Engineering

[PH07] Papazoglou, M.; Heuvel, J.: Service oriented architectures: approaches, technologies and research issues, Paper, International Journal on Very Large Data Bases (VLDB), 16:389–415, 2007

[Ro03] Rosemann, M: Process-oriented Administration of Enterprise Systems, ARC SPIRT project, Queensland University of Technology, 2003

[SW08] Schelp, J.: Winter, Robert: Business Application Design and Enterprise Service Design: A Comparison. In: Int. J. Service Sciences 3/4 (2008)

[SOA06] SOA: Reference Model for Service Oriented Architecture Committee Specification, www.oasis-open.org, 2006

[St06] Stamati, T: Investigating The Life Cycle Of Legacy Systems Migration, European and Mediterranean Conference on Information Systems (EMCIS), Alicante Spain 2006

[Sv07] Svatoš, O.: Conceptual Process Modeling Language: Regulative Approach, Department of Information Technologies, University of Economics, Czech Republic, 2007

[RB09] Riebisch, M; Bode, S.: Software-Evolvability, GI Informatik Spektrum, Bd 32 4, Technische Universität Ilmenau, 2009

[TOG09] TOGAF, 9.0: The Open Group Architecture Framework, Vendor- and technology-neutral consortium, The Open GROUP, www.togaf.org, 2009

[UML] UML: Unified Modeling Language, Not-for-profit computer industry consortium, Object Management Group, www.omg.org

Software Engineering Rationale: Wissen über Software erheben und erhalten

Kurt Schneider

Lehrstuhl Software Engineering
Leibniz Universität Hannover
Welfengarten 1, 30167 Hannover
Kurt.Schneider@inf.uni-hannover.de

Abstract: Damit Software lange genutzt werden kann, muss sie fortwährend korrigiert, erweitert und verändert werden. Dabei werden Entscheidungen und Annahmen getroffen, Domänenwissen wird in die Software eingearbeitet und in Modellen interpretiert. Programme zu warten, bei denen dieses Wissen fehlt, ist schwierig. Das ergänzende Wissen müsste eigentlich zusammen mit dem Programm aufgebaut, gepflegt und weitergegeben werden. Auf längere Sicht kann sich sogar das Programm selbst als weniger langlebig erweisen als das damit assoziierte Wissen. Dieser Beitrag betont den komplementären Charakter von Programm und Rationale, codiertem und ergänzendem Wissen und zeigt Forschungsherausforderungen auf dem Weg, diesem Charakter gerecht zu werden.

1 Einleitung

Von Anforderungsanalyse bis zu Test und Wartung werden unzählige große und kleine Entscheidungen gefällt und Informationen über Software in diese eingearbeitet. Verliert man das Wissen über die Gründe (das „Rationale“), so verliert man zunehmend auch die Fähigkeit, die Software zu pflegen. Durch innovative Techniken und Werkzeugen soll es gelingen, dieses Zusatzwissen zu sichern, ohne viel spürbaren Zusatzaufwand zu treiben. Entwurfsbegründungen, Erfahrungen und Feedback aus dem Betrieb helfen dann, die Software beweglich und nützlich zu erhalten.

Software altert relativ zu den veränderten Kontextbedingungen: Mitarbeiter gehen und nehmen ihr Wissen mit. Neue Mitarbeiter kommen und verändern die Software, ohne frühere Entscheidungen und ihre Grundlagen zu kennen. Nur wenn man schnell genug bemerkt, dass sich da eine Lücke auftut, kann man reagieren und die Veralterung aufhalten. Das gilt nicht nur in der Entwicklung, sondern auch später im Betrieb der Software. Zusammen mit dem *Programm* muss auch *Wissen* bewahrt werden.

Je nach Art und Granulat des bewahrten Wissens kann es in unterschiedlicher Weise später genutzt werden, wie die folgenden Beispiele zeigen.

Dabei geht es um ein weites Spektrum von Aspekten, nicht nur um Codefeinheiten:

- Es ist wichtig zu wissen, ob ein Programm eigentlich als „Prototyp“ gedacht war und dann doch weiter gewachsen ist. Das erklärt viele Besonderheiten und Schwächen; man glaubt ja anfangs, der Prototyp werde bald weggeworfen.
- Wenn eine Sicherheitsnorm wie ISO 61508 der Grund dafür war, sicherheitskritische von eher unkritischen Programmteilen zu trennen, so muss man das später wissen – sonst sind die darauf aufbauenden Architekturentscheidungen nicht nachvollziehbar und werden vielleicht verletzt.
- Begriffe, Regelungen und Terminologie aus einer Domäne können sich für spätere Entwickler „falsch“ anhören, weil diese die Domäne nicht kennen. Trotzdem sollte man sie beibehalten.
- Am codenahen Ende des Spektrums muss man wissen, wo und warum ein Design Pattern angewendet wurde, damit man es nicht später „kaputtoptimiert“.

IEEE Std 610.12 – 1990 definiert *Software* als „Computer programs, procedures and possibly associated documentation and data pertaining to the operation of a computer system”. Es kann geschehen, dass dieses ergänzende Wissen in Form von Modellen, Entscheidungen und Erfahrungen wichtiger wird als das Programm selbst. „Die Software“ lebt weiter, obwohl das ursprüngliche Programm ersetzt wird.

Mit einem Softwareprodukt muss ein Schatz technischer und umgebungsbezogener Informationen wachsen. Aufgebaut wird er mit möglichst wenig Zusatzaufwand, ja vielleicht sogar mit unmittelbarem Zusatznutzen schon während der Softwareentwicklung. Denn kaum ein Softwareentwickler ist bereit, Zeit und Mühe zu investieren, nur damit es die Nachfolger leichter haben. Eine der zentralen Herausforderungen für die Bewahrung des Softwarewissens steckt in dieser Tatsache: Oft wird beispielsweise ein scheidender Experte aufgefordert, sein Wissen „aufzuschreiben“. Das dauert viel zu lange und nützt nur dem Nachfolger! Wer aber einen Beitrag zur Wissensbewahrung leistet, muss selbst erkennbaren Nutzen irgendeiner Art haben. Gegen diese scheinbar so triviale Einsicht wird ständig verstoßen. Könnte der Experte sein Programmteil zum Beispiel einfach mündlich demonstrieren und erklären, wäre nicht nur sein Aufwand geringer (siehe FOCUS-Beispiel im Hauptteil). Seine Expertenrolle wird herausgehoben, was mit sozialer Anerkennung verbunden ist. Die ist oft viel wirksamer als etwas Geld.

In Abschnitt 2 werden Beispiele von assoziiertem Wissen gegeben, das im Zusammenhang mit Programmen – und als Teil von „Software“ – eine wichtige Rolle für Wartung und Langlebigkeit spielt. An diesen Beispielen wird gezeigt, was mit dem komplementären Charakter von Programm und Wissen gemeint ist. Auf dieser Basis kann Abschnitt 3 skizzieren, wie sich dieses Wissen auf die Langlebigkeit von Software im weiteren Sinne auswirkt. Abschnitt 4 beschreibt, worauf bei der Erhebung zu achten ist. Die Erhebung ist nicht als isolierte Wissensmanagement-Aufgabe konzipiert, sondern als Koevolution von Wissen und Programmartefakten angelegt. Damit wird die Wissenserhebung als integraler Bestandteil der Softwareentwicklung interpretiert. In so einer Konstellation werden sich „eigentliche Softwareentwicklung“ und „Wissensarbeit“ gegenseitig beeinflussen.

Da Wissen oft langlebiger ist als Programme, kann sich das anfängliche Verhältnis langfristig umkehren (Abschnitt 5). Dann wird das Wissen die treibende Kraft, an der das Programm hängt. Schließlich diskutiere ich in Abschnitt 6 einige wichtige Forschungsfragen, die sich aus diesem Ansatz für die Zukunft ergeben.

2 Rationale, Erfahrungen, Entscheidungen – Basis für Langlebigkeit

Ein wesentliches Postulat dieses Beitrags lautet:

> In Zusammenhang mit Softwareentwicklung wird viel Wissen zusammengetragen und erstellt, von dem sich nur ein kleiner Teil in den eigentlichen Programmen wiederfindet. Das ergänzende Wissen bleibt oft undokumentiert. In der Wartung fehlt es dann, und das verursacht Strukturbrüche und Fehler, macht die Altsoftware letztlich unbrauchbar.

Beim Übergang von Requirements Engineering zu Entwurf hat sich diese Einsicht seit langem durchgesetzt. Es gibt zahlreiche Arbeiten, die sich mit dem „Design Rationale“ auseinandersetzt, also mit den Begründungen für Entwurfsentscheidungen [DMMP06]. Beides, die Entscheidungen und die Gründe dafür, stellen Meta-Informationen (in Anlehnung an Metadata) dar, die nicht erklären, *wie* zum Beispiel ein Algorithmus funktioniert – sondern *wieso* er vor dem Hintergrund der konkreten Anforderungen und *mit welcher Begründung* ausgewählt worden ist. Diese Begründung steckt nicht in den Programmzeilen und kann von dort auch nicht extrahiert werden (von den Kommentaren einmal abgesehen). Dagegen enthalten die Programmzeilen das ganze Wissen über die Funktionsweise des Algorithmus; sie sind ja dessen operative Implementierung. Es mag auch Aspekte geben, die sowohl im Code als auch im ergänzenden Wissen ihre Spuren hinterlassen. Das Entwicklungswissen steckt also in der *Kombination* von Programmcode und zugehörigem Rationale (Abbildung 1). In sofern verwundert es nicht, dass man den nackten Code ohne zugehöriges Wissen irgendwann nicht mehr verstehen und pflegen kann.

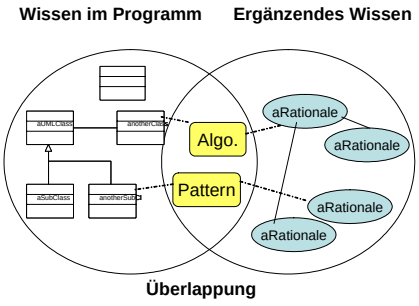


Abbildung 1: Wissen in der Softwareentwicklung landet im Programm oder wird zu ergänzendem Wissen. Manches trägt zu beidem bei (Prinzipdarstellung).

Die Rolle der Dokumentation im Software Engineering wird seit langem betont [Bo81]. Dabei ist jedoch nicht die Tatsache entscheidet, dass es möglichst viel Dokumentation gibt; dagegen wehren sich nicht nur die agilen Methoden mit gutem Recht [Be00, C02]. Es kommt vor allem darauf an, welche Inhalte dabei erfasst werden und wie man sie wieder nutzen kann. Die traditionellen Dokumente enthalten oft tautologische Beschreibungen von Programmstrukturen, die man leicht aus dem Code selbst extrahieren kann. Dabei handelt es sich also nicht um ergänzendes Wissen.

Verschiedene **Typen von ergänzendem Wissen** sind besonders wichtig, werden aber häufig nicht erfasst:

- **Annahmen** über die Randbedingungen. Ändern sich die Bedingungen, so können sich auch darauf aufbauende Entscheidungen ändern.
- **Domänenwissen**: Zusammenhänge im Anwendungsbereich schlagen sich einerseits direkt im Code nieder, andererseits bilden sie die Begründung für Entscheidungen.
- **Prüfergebnisse**: Review- und Testberichte können Informationen zur Bewertung von Code liefern. Diese sind nicht aus dem Code selbst abzulesen.
- **Feedback** aus der Programmnutzung: Rückmeldungen der Nutzer können Divergenzen zwischen Annahmen und realer Situation oder zwischen Anforderungen und Programmeigenschaften erkennen lassen. Sie sind ebenfalls naturgemäß nicht im Code repräsentiert.
- **Erklärungen über das Umfeld oder absehbare Änderungen**. Sie können sich teilweise in flexiblerem Code niederschlagen (z.B. durch den Einsatz von Design Patterns [GHJV95]).
- **Erfahrungen** verschiedener Art entstehen während der Entwicklung und Wartung eines Programms. Zum Beispiel kann man feststellen, dass ein Modul bisher besonders fehleranfällig war und bei neuen Fehlern zuerst geprüft werden sollte. Design Patterns sind typische Resultate von Erfahrungen.
- **Modelle**: Sie stellen gewisse relevante Merkmale der Software dar.

Die Liste ist sicher nicht vollständig. Sie nennt vor allem Informations- und Wissensarten, die Begründungen, Bewertungen und darauf aufbauende Schlussfolgerungen oder Entscheidungen enthalten. Daher ist es sinnvoll, kurz von „Software Engineering Rationale“ zu sprechen. Der Begriff schließt „Design Rationale“ ein, geht aber darüber hinaus.

Eine besonders wichtige Rolle für die Vermittlung zwischen einem Programm und dem Wissen über dieses Programm spielen zunehmend „Modelle“, oft in UML. Wie bereits von Stachoviak [St73] 1973 festgestellt, erweist ein Modell seine Nützlichkeit stets im Hinblick auf ein *Original*, einen *Modellzweck* und einen *Nutzer*. Das Original von UML-Entwurfsmodellen kann beispielsweise der später erzeugte Code sein. Das UML-Modell wird zum Zweck der Codeerstellung (manuell oder generiert) entwickelt, entweder für Entwickler oder für Generatoren.

In der MDA [PM06] oder der Modellgetriebenen Software-Entwicklung [SV05] wird Code aus den Modellen generiert. Damit enthält der Code keine zusätzlichen Informationen, die Modelle allein sind die Träger des Wissens.

An dieser Stelle sind zwei alternative Konsequenzen denkbar:

- Entweder werden die Modelle (anstelle des Codes) mit ergänzendem Wissen verbunden. Das Gesamtwissen der Software ist dann in zwei verschiedenen, jedoch miteinander verbundenen Modellen repräsentiert: im UML-Modell und im Wissensmodell.

- Oder der Modellzweck nach Stachoviak wird umdefiniert und erweitert: Das Modell dient dann nicht mehr nur der Codeerstellung, sondern außerdem der Sicherung ergänzenden Wissens. In diesem Fall wären Metamodelle zu erweitern. Neben Codegenerierung wären Wissensspeicherung und -pflege die Zwecke des Modells. Die Adressaten im Stachoviakschen Sinn sind dann nicht mehr allein die Erstentwickler, sondern alle, die im Lebenszyklus einer Software an dieser arbeiten.

Da sowohl UML-Modelle als auch Ontologien formal fassbar und sogar strukturell ähnlich sind, gibt es viele Möglichkeiten, sie miteinander zu verbinden. Abbildung 2 zeigt eine davon.

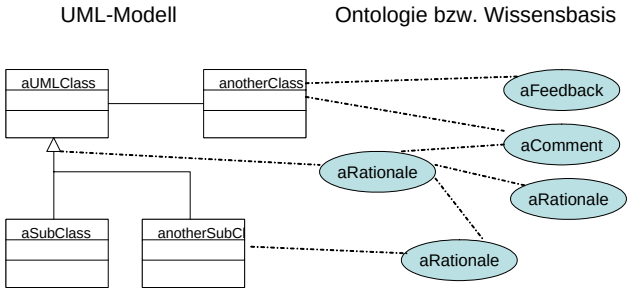


Abbildung 2: Verschiedene, aber verbundene Modelle für Produkt und Wissen bzw. Rationale

Diese Überlegung verdeutlicht, wie unterschiedlich technische Realisierungen für ergänzendes Wissen aussehen können. In den folgenden Abschnitten wird für die enge Integration und Verbindung argumentiert. Dafür bieten Metamodellerweiterungen mit UML-Profilen natürlich einen interessanten technischen Ansatz. Dieser Aspekt ist hinlänglich bekannt und soll daher hier nicht weiter ausgeführt werden.

Andererseits sollte man auch nicht aus den Augen verlieren, dass aus dem Wissens-management Konzepte wie Ontologien zur Verfügung gestellt werden, die den Aspekt der Wissensrepräsentation gezielter abdecken als ein kurzfristig erweitertes UML-Metamodell. Es wird eine der Forschungsfragen sein, hier eine Abwägung zu treffen.

3 Lebenszyklus von Software und “ergänzendem Wissen”

Software wird eingesetzt, so lange sie nützlich ist, also ihren Betreibern einen Nutzen verspricht. Weichen die Funktionen oder die nicht-funktionalen Eigenschaften zunehmend von den – möglicherweise veränderten – Erfordernissen ab, wird eine Möglichkeit gesucht, die Abweichung wieder zu verringern. In der Regel wird man versuchen, die Software an die neue Situation anzupassen. Wenn das zu aufwändig erscheint oder überhaupt nicht mehr geht, beginnt man, das eigene Verhalten und die eigenen Geschäftsprozesse an die Möglichkeiten der alternden Software anzugleichen. Das ist ein deutliches Zeichen für das nahende Ende des Software-Lebenszyklus. Symptomatisch ist auch, wenn keine Verbesserungen mehr gewagt werden. Die Software kann nur noch so verwendet werden, wie sie gerade ist. In dieser Phase des Lebenszyklus wird die Software als Last oder Legacy wahrgenommen, die nur deshalb weiter eingesetzt wird, weil es kurzfristig keine Alternative gibt.

Man kann sich die obigen Effekte aus den Wirkzusammenhängen zwischen dem Programm und dem ergänzenden Wissen erklären. In diesem Ablauf zeigt sich eine typische, aber nicht wünschenswerte Entwicklung, die man aufhalten müsste:

1. Während der Entwicklungsphase tragen die Kunden, Entwickler und anderen Projektbeteiligten die wichtigsten Informationen in sich. Es gibt persönliche Notizen, Protokolle und auch Modelle. Entscheidungen werden auf dieser Basis wohlüberlegt und gut informiert getroffen.

2. Durch die hohe Geschwindigkeit der Entwicklung und durch den meist hohen Zeitdruck werden die Aufzeichnungen nicht mehr systematisch erfasst und gepflegt. Manche Modelle sind nicht mehr aktuell, auch Anforderungen, die sich ändern, werden nicht bis in alle Konsequenz nachgeführt.

3. Nach und nach verlassen ursprüngliche Mitarbeiter und Stakeholder das Projekt. Sie nehmen ihr Wissen mit und lassen nur den Teil im Projekt, der sich inzwischen in Code oder expliziter Dokumentation niedergeschlagen hat. In der Praxis geht dabei viel ergänzendes Wissen verloren.

4. Unter den Effekten (2, 3) leidet die Software zunächst scheinbar nicht, da das Programm ja an und für sich „funktioniert“. Kommt es jedoch zu Änderungsbedarf (auf Grund von Fehlern, Weiterentwicklung oder Anpassung an die geänderte Umwelt), so fehlen immer mehr Informationen.

5. Daher wird oft versucht, nur kleine, lokale Änderungen vorzunehmen. Dies führt allerdings erst recht zu einer schwer durchschaubaren Struktur mit vielen Ausnahmen und möglicherweise Inkonsistenzen: die „Patches“ passen nicht mehr zueinander und zu der grundlegenden Systemstruktur. Die Struktur verwildert zusehends.

6. Nur im Idealfall werden die Änderungen ebenfalls dokumentiert und das so gesicherte Wissen mit dem zuvor gesammelten, ergänzenden Wissen integriert. In der späten Lebensphase eines (Legacy-) Systems, in der man kein grundlegendes Refactoring und keine Umstrukturierung mehr wagt, wird kaum mehr Energie in Wissenssammlung gesteckt werden. Ohnehin ist das System nun nur noch schwer zu verstehen, was die Wissensintegration weiter erschwert und zunehmend sinnlos erscheinen lässt.

7. Ein Teufelskreis ist in Gang gekommen, der letztlich zu Lücken und Inkonsistenzen im gesamten Wissen (bestehend aus Code, Modellen und ergänzendem Wissen) führt. Der Code hat sich durch die lokalen Patches von den ursprünglichen Plänen wegentwickelt. Die Pläne und Entwürfe selbst können mangels Überblick nicht mehr konsistent gepflegt werden.

Für die Entwicklung langlebiger Softwaresysteme müsste man an mehreren Stellen in den skizzierten Lebenszyklus eingreifen:

In Schritt (1) werden Anforderungen, Rationale und weiteres Wissen zusammen getragen. Wenn man sie in Schritt (2) wirkungsvoll erfassen könnte, dann könnte die nachteilige Wirkung der folgenden Schritte gemildert werden. In unseren Arbeiten nimmt Schritt (2) hier eine Schlüsselstellung ein: Die Phase, in der zwar die Entwicklungstätigkeit alle Energie an sich zieht, in der aber andererseits auch die Entscheidungen und das Rationale noch präsent sind, betrachten wir als geeigneten Einstiegspunkt für die Rettung von ergänzendem Wissen: Im „By-Product Approach“ [Sc06] ist beschrieben, wie man durch gezielte Computerunterstützung in dieser Situation Wissen einsammelt, das sonst flüchtig und „flüssig“ (in der Terminologie der Informationsflüsse [SSK08]) bliebe und rasch versickern würde. Es stünde dann in der Wartung nicht mehr zur Verfügung.

Wenn es zusätzlich gelingt, das gesammelte Wissen bei Änderungen nutzbar zu machen (Schritt 5), so steigt die Chance, die kombinierte Struktur länger zu achten. Wiederum sollte man aufwandsschonende Computerunterstützung nutzen, um auch bei Änderungen die Begründungen und Entwurfsentscheidungen einzusammeln. Dabei muss man wieder ganz konkret bedenken: auch bei Änderungen muss jeder Aufwand durch einen persönlichen Nutzen kompensiert werden.

Es ist kaum zu erwarten, dass man den Alterungsprozess ganz zum Stillstand bringen kann. Aber eine merkliche Verlangsamung wäre bereits ein wichtiger Erfolg.

4 In Zukunft: Programme, Modelle und Wissen verbinden

Es gibt mehrere Beispiele, wie das „By-Product“-Konzept umgesetzt werden kann. Als Beispiel wird hier das Werkzeug FOCUS verwendet, das die Konzepte besonders deutlich zeigt: Mit FOCUS kann man während Software-Vorführungen (Demos) Wissen extrahieren, das zuvor in dem Programm und seinem Entwickler steckte. Schon 1996 wurde eine Smalltalk-Version von FOCUS auf der ICSE vorgestellt [Sc96].

Die Ideen hinter FOCUS sind einfach, aber nicht ganz einfach umzusetzen:

- Während ein kenntnisreicher Entwickler seinen Prototypen vorführt, wird diese Demonstration aus mehreren Perspektiven aufgezeichnet: die Bildschirmanzeige und ein Trace (aufgezeichnete Sequenz) ausgeführter Methoden ergeben zwei, durch FOCUS miteinander verbundene Resultate.

- FOCUS kann nun mit Hilfe der Traces die technische Erklärung des Prototypen anleiten: Im Browser werden die Methoden der Reihe nach angezeigt, die während der Demo ausgeführt wurden. So lassen sich – von der Oberfläche bis zu Algorithmen, Architekturentscheidungen und sogar fragwürdigen Kompromissen – viele relevante Punkte erläutern, indem man den Traces folgt.

- Es steht dem Entwickler frei, kommentarlos weiterzuschalten – oder die Wirkungsweise einer Methode zu erklären. Wieder ergibt sich ein Pfad aus den Methoden, die erklärt wurden. Er folgt teilweise den Traces, weicht dann aber wieder ab.

- Das Verfahren kann mit mehreren Demonstrationen und mehreren Erklärungssitzungen angewendet werden und führt so – als Nebenprodukt – dazu, dass FOCUS jedes Mal einen „Ausführungspfad“ und einen „Erklärungspfad“ aufzeichnet und zu den bestehenden hinzufügt. Diese Pfade können sich überscheiden und parallel laufen, oder sie können divergieren.

- In späteren Sitzungen können Entwickler auch Erklärungspfaden folgen, die zuvor von anderen angelegt wurden. Da die Aufnahme der Pfade vollständig automatisiert durch FOCUS erfolgt, bieten die Pfade und alle darauf aufbauenden Mechanismen die Gelegenheit, weiteres Wissen einzusammeln. Hat der Entwickler dann das Projekt verlassen (wie in Schritt 6), so ist doch ein Netz aus Pfaden und Bildschirmvideos entstanden und im Projekt verblieben, das eng mit den Methoden (also dem Programmcode) verbunden ist.

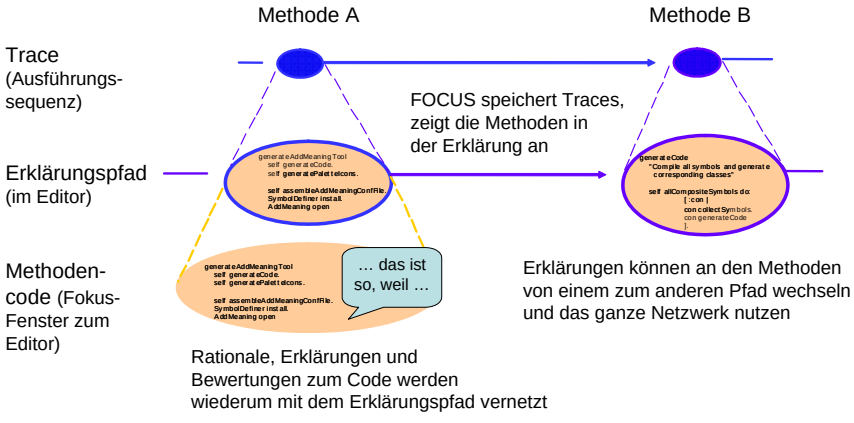


Abbildung 3: Traces und Erklärungspfade werden in FOCUS mit Code vernetzt

Selbstverständlich sind viele andere Beispiele vorstellbar, die durch Aufzeichnung, Beobachtung und Integration durch Indizierung nach dem gleichen Prinzip Wissen erfassen und strukturieren [Sc06]. In der Regel setzt die Anwendung des Prinzips jedoch eine spezifische Werkzeugunterstützung voraus, die tief in die Details der wissensrelevanten Arbeitsschritte integriert sein muss. Bei FOCUS müssen Traces erfasst, Browser damit gesteuert und gleichzeitig Erklärungen aufgezeichnet werden. Ein Trace ist die Folge ausgeführter Methoden von ausgewählten Klassen. Ferner müssen die Bildschirmaufzeichnung indiziert werden, damit man gezielt einzelne Sequenzen anspringen kann; beispielsweise die Erklärung einer komplizierten Methode.

Bei ganz anderen wissensrelevanten Aufgaben prägt sich das Prinzip auch anders aus: In der Projektplanung oder im Risikomanagement würde man anders vorgehen. Auch FOCUS wurde mehrere Jahre später noch einmal für Java als Eclipse plug-in realisiert [Sc06]. Dabei konnte FOCUS in Smalltalk auf sich selbst angewendet werden, um zumindest einige Grundkonzepte über Jahre hinweg nach Java zu retten.

5 Programme, Modelle und Wissen separat nutzbar machen

Soeben wurde das Netzwerk aus Pfaden geschildert, die in FOCUS mit dem Code über die Methoden feinmaschig verbunden sind. Ab Schritt (4) im Lebenszyklus beginnt jedoch eine neue Entwicklung, der man mit anderen Mitteln begegnen muss:

Teile des Codes müssen ersetzt oder portiert werden, oder sie funktionieren wegen Fehlern und Änderungen in der Umwelt nicht mehr wie zuvor. Dann kann es sinnvoll sein, das Verhältnis zwischen Code und ergänzendem Wissen (in FOCUS: Pfaden und Aufzeichnungen) umzudrehen. Das (ursprünglich: ergänzende) Wissen wird zur primären Wissensrepräsentation, die Bildschirmaufzeichnungen und Quellcodeauszüge werden zu Beispielen und Illustrationen für das Rationale.

In der ersten Version von FOCUS in Smalltalk [Sc96] erhielten die Pfadelemente durch Vererbung die Fähigkeit, das gesamte Netzwerk in der oben geschilderten Weise in html auszugeben und mit dem Group Memory der Abteilung zu verbinden. Bildschirmaufzeichnungen und der Quellcode einzelner Methoden wurden darin zur Veranschaulichung eingebettet. Die Pfade wurden durch Links abgebildet. Diese Darstellung ist gut lesbar, aber nicht ausführbar und stellt eindeutig einen Bruch im Sinne von Lebenszyklus Schritt (6) dar. Daher sollte so lange wie möglich mit der integrierten Variante gearbeitet werden. Andererseits ist die so entstehende Wissensdarstellung nicht mehr auf Smalltalk-Umgebungen angewiesen; ein Browser reicht.

Die Integration von Rationale und Wissen in den Code bedeutet übrigens nicht, dass in einem Softwareprodukt auch alle eingebetteten Wissenselemente an den Kunden ausgeliefert werden müssten oder sollten. Lediglich solche Mechanismen, die für die Erfassung von Feedback von den Bedienern gedacht sind, müssen dort verbleiben – was zu vielen weiteren interessanten Fragestellungen führt [LS06, LK07]. Es muss jedoch eine Referenzinstallation bei der Entwicklung geben, die das ergänzende Wissen enthält. Bei Bedarf wird eine neue Produktversion extrahiert und ausgeliefert.

6 Herausforderungen und Forschungsrichtungen

Der oben vorgestellte Ansatz, ergänzendes Wissen durch Computerunterstützung mit Programmartefakten zu integrieren, steht noch am Anfang seiner Entwicklung. FOCUS ist ein relativ codenahes und leicht erklärbares Beispiel für seine Umsetzung; viele andere Arten von Wissen können nach ähnlichen Prinzipien bewahrt werden.

Dabei ist die Idee, Wissenmanagement-Methoden auf Software Engineering anzuwenden, nicht neu: Es gibt sogar Tagungsreihen zu dem Thema (wie SEKE: Software Engineering und Knowledge Engineering, KBSE: Knowledge-Based Software Engineering) mit internationalem Journal dazu. Auch der Workshop LSO (Learning Software Organizations) bearbeitet seit Jahren verwandte Themen. Der Workshop ist manchmal einer Software-Engineering-Tagung, dann wieder einer Wissensmanagement-Veranstaltung angegliedert. Die Relevanz des Themas ist grundsätzlich schon lange bekannt. Allerdings ist der hier besprochene Aspekt der „nebenbei wirksamen", auf die Software-Engineering-Aktivitäten fein abgestimmten Strukturen und Werkzeuge noch nicht im Zentrum der Aufmerksamkeit. Ohne Werkzeuge – insbesondere ohne *akzeptierte* Werkzeuge – können Wissenserhebung und -nutzung aber nicht funktionieren.

Eine ganze Reihe von Forschungsfragen lässt sich aus diesem Ansatz ableiten. Sie bilden einen Kern dieses Beitrags, weil sie Möglichkeiten aufzeigen, den Problemen langlebiger Softwaresysteme einen Schritt näher zu kommen:

- *Welche Informationen und Wissensarten braucht man?* Zunächst müssen solche Wissensarten identifiziert werden, die in der Wartung (Lebenszyklus Schritt 4-6) gebraucht würden, aber oft fehlen. Diese können dann gezielt gesammelt werden. Die Beispiele in Abschnitt 2 werden dazu gehören.

- *Wie kann man ergänzendes Wissen nebenher erheben?* Der hier vorgestellte Ansatz sieht vor, durch spezifische Computerunterstützung aufzunehmen und zu indizieren. Dabei ist genau auf das Verhältnis von Aufwand und Nutzen zu achten.

- *Wie integriert man die Wissenserhebung und -verwendung in den Arbeits- und Lernprozess einer lernenden Software-Organisation [Bi06]?* Nach Kelloway und Baring [KB03] kommt Lernen am Arbeitsplatz nur in Gang, wenn dafür die (1) Fähigkeit des Einzelnen, die (2) Gelegenheit durch Aufgabe und Umfeld und (3) Motivation zusammenkommen. Wie das in der Softwareentwicklung zu gewährleisten ist, geht über das reine Software Engineering hinaus. Aber auch für das Software Engineering ergeben sich konkrete Konsequenzen, wenn wirksame Unterstützung angeboten werden soll.

- *Wie bringt man die Experten dazu, ihr Wissen für andere preiszugeben?* Diese Frage ist ein Spezialfall der vorigen und hat ihrerseits viele Facetten, von der Gestaltung von Incentives [DP00] bis hin zur Bedienbarkeit der eingesetzten Werkzeuge, einem Aspekt der Usability im Software Engineering [La05].

- *Was ist der primäre Wissensträger?* Bisher wird außer dem Code kaum etwas aufgehoben; Code ist dann der primäre Wissensträger, zu dem ergänzendes Wissen hinzukommen kann (in Ontologien, Büchern, FOCUS-Pfaden). Wie sieht das aber aus, wenn Code aus Modellen generiert wird?

- *Wie verhalten sich die Modelle des Softwaresystems und Modelle des ergänzenden Wissens zueinander?* Sind die Metamodelle integriert oder stehen sie nebeneinander (z.B. UML und Wissensschema)? Wie kann man sie trennen, wenn dies erforderlich wird?

- *Welche Mechanismen können bei der Wiederverwendung des Wissens eingesetzt werden?* Auch hier ist an eine aufwandsoptimierte Lösung aus Sicht der Beteiligten zu denken. Die Computerunterstützung beim Erfassen kann auch das Wiederverwenden erleichtern, wie das Beispiel der html-Ausgabe von FOCUS-Pfaden zeigt. Keinesfalls reicht es aus, Informationen „auf dem Internet verfügbar" zu machen. Sie müssen aktiv viel näher an die Arbeitsaufgaben der Entwickler und der Wartungsingenieure gerückt werden.

Viele weitere Fragen aus dem Bereich des Wissensmanagement und der Psychologie können interdisziplinär angegangen werden. Im vorliegenden Beitrag sieht man, dass schon aus der reinen Software-Engineering-Perspektive viele interessante Fragen auftauchen [Sc09]. Für manche Aspekte haben wir bereits Vorarbeiten geleistet. Aber die meisten Fragen sind noch offen.

Manche der Vorarbeiten adressieren einzelne Fragen, während andere versuchen, eine durchgängige Lösung für ein spezielles Anwendungsproblem anzubieten [Vo06, Vr06]. Es entstehen auch praktisch einsetzbare Werkzeuge, die sich an die Modelle oder Code anlehnen, um Wissen zu erheben und zu erhalten. Dahinter sind semantisch weitergehende Aufbereitungen und Auswertungen möglich. Zunächst ist es aber am wichtigsten, an viel versprechenden Stellen in Entwicklung und Betrieb Feedback einzusammeln und Erfahrungen zu erheben – damit sie für langlebigere Konzepte und Software genutzt werden können.

7 Fazit

Dieser Beitrag hebt die Bedeutung von „ergänzendem Wissen", insbesondere von Software Engineering Rationale, für langlebige Software hervor. Für das Software Engineering ist es wichtig, dieses Wissen wirksam zu sammeln und mit den Programmen oder Modellen so zu verbinden, dass es bei der Wartung an Ort und Stelle ist und nicht mühsam gesucht werden muss. Das verlangt nach neuen Konzepten, spezialisierten Techniken und Werkzeugen. Ihre Entwicklung ist schwieriger, als es zunächst erscheinen mag: Eine Reihe dabei relevanter Forschungsfragen wurde genannt. Es reicht nicht aus, sich einmal mit einer davon zu beschäftigen. Um Software langlebiger zu machen, muss man das ergänzende Wissen retten – und seine Beziehung zu den Programmen und Modellen ohne ständigen Zusatzaufwand aktuell halten.

Literaturverzeichnis

[Be00] Beck, K.: Extreme Programming Explained: Addison-Wesley (2000).

[Bi06] Birk, A., et al. (eds): Learning Software Organisation and Requirements Engineering: The First International Workshop. J.UKM Electronic Journal of Universal Knowledge Management (2006).

[Bo81] Boehm, B.: Software Engineering Economics, N.J.: Prentice Hall, Engelwood Cliffs (1981).

[Co02] Cockburn, A.: Agile Software Development: Addison Wesley (2002).

[DMMP06] Dutoit, A.H.; McCall, R.; Mistrik, I.; Paech, B. (eds.): Rationale Management in Software Engineering. Springer: Berlin (2006).

[DP00] Davenport, T., Probst, G..: Knowledge Management Case Book - Best Practises, München, Germany: Publicis MCD, John Wiley & Sons (2000).

[GHJV95] Gamma, E., Helm, R., Johnson, R., Vlissides, J..: Design Patterns - Elements of Reusable Object-Oriented Software: Addison-Wesley Publishing Company (1995).

[KB03] Kelloway E K, Barling, J.: Knowledge work as organizational behavior, International Journal of Management Reviews, 2000. 2(3). (2000) 287-304.

[La05] Lauesen, S.: User Interface Design - A Software Engineering Perspective, Harlow, London: Addison Wesley (2005).

[LK07] Lübke, D. and Knauss, E.: Dealing with User Requirements and Feedback in SOA Projects, in Workshop on Software Engineering Methods in Service Oriented Architecture. Hannover, Germany (2007).

[LS06] Lübke, D. and Schneider, K.: Leveraging Feedback on Processes in SOA Projects, in EuroSPI. Joensuu: Springer-Verlag Berlin-Heidelberg (2006).

[PM06] Petrasch, R. and Meimberg, O.: Model-Driven Architecture. Eine praxisorientierte Einführung in die MDA, Heidelberg: Dpunkt Verlag (2006).

[Sc06] Schneider, K.: Rationale as a By-Product, in [DMMp06] 91-109.

[Sc09] Schneider, K.: Experience and Knowledge Management in Software Engineering. Springer, Berlin (2009).

[Sc96] Schneider, K.: Prototypes as Assets, not Toys. Why and How to Extract Knowledge from Prototypes, in 18th International Conference on Software Engineering (ICSE-18). Berlin, Germany (1996).

[SSK08] Schneider, K., Stapel, K., and Knauss, E.: Beyond Documents: Visualizing Informal Communication, in Third International Workshop on Requirements Engineering Visualization (REV 08). Barcelona, Spain: IEEE Xplore (2008).

[St73] Stachoviak, H.: Allgemeine Modelltheorie, Wien, New York: Springer Verlag (1973).

[SV05] Stahl, T. and Völter, M.: Modellgetriebene Softwareentwicklung. Techniken, Engineering, Management, Heidelberg: Dpunkt Verlag (2005).

[Vo06] Volhard, C.: Unterstützung von Use Cases und Oberflächenprototypen in Interviews zur Prozessmodellierung, Fachgebiet Software Engineering, Gottfried Wilhelm Leibniz Universität Hannover (2006).

[Vr06] Vries, L.d.: Konzept und Realisierung eines Werkzeuges zur Unterstützung von Interviews in der Prozessmodellierung, Fachgebiet Software Engineering, Gottfried Wilhelm Leibniz Universität Hannover (2006).

Was braucht es für die Langlebigkeit von Systemen? Eine Kritik

Lutz Prechelt
Institut für Informatik, Freie Universität Berlin
prechelt@inf.fu-berlin.de

Abstract: Der Aufruf zum 1. Workshop "Langlebige Softwaresysteme" lässt in seiner Sicht darauf, was forschungsseitig für erfolgreiche Langlebigkeit von Softwaresystemen zu tun ist, vernachlässigte Bereiche erkennen. Dieser Artikel zeigt auf, wo diese liegen, warum die Vernachlässigung problematisch ist und mit welcher Maßnahme eine Beschädigung des resultierenden Forschungsprogramms vermutlich vermieden werden kann.

1 Ziel und Aufbau dieses Artikels

Der Einreichungsaufruf zu diesem Workshop[1] charakterisierte dessen Zweck wie folgt: Im Workshop "sollen die oben geschilderte Entwicklung [dass es zunehmend bedeutsamer wird, der Software-Erosion erfolgreich entgegenzutreten], Erfahrungen hierzu sowie Lösungsansätze [...] beleuchtet werden." Die Sicht der Workshop-Organisatoren darauf, welche Aspekte dafür wichtig sind, beleuchtet im Aufruf die dann folgende Themenliste.

Dieser Artikel geht anhand dieser Themenliste der Frage nach, ob es in dieser Sicht, was für erfolgreiche Langlebigkeit wissenschaftlich zu bearbeiten wichtig ist, "blinde Flecken" gibt, ob also bedeutsame Teilaspekte nicht oder kaum beachtet werden – was die Chancen einer erfolgreichen Bearbeitung natürlich verringern würde.

Ich gehe dabei davon aus, dass (a) der Aufruf die Haltung der Organisatoren gut wiedergibt und (b) die Organisatoren die deutsche Forschungsszene in diesem Sektor gut repräsentieren, so dass gefundene blinde Flecken also Risiken dafür aufzeigen, dass eine gemeinschaftlich definierte größere Forschungsanstrengung in eine nicht optimale Richtung gehen könnte, wenn diese unterrepräsentierten Aspekte nicht mehr Beachtung erhalten als bisher.

Dazu stelle ich als erstes eine einfache Taxonomie relevanter Teilaspekte auf, die so allgemein ist, dass sie im Grundsatz hoffentlich konsensfähig ist (Abschnitt 2). Die Frage, welche Teile davon sich in welcher Gewichtung am stärksten für eine Erforschung aufdrängen, bleibt dabei zunächst offen. Dann vergleiche ich diese Taxonomie mit den Einträgen der Themenliste aus dem Workshop-Aufruf, um deren Orientierung und Gewichtung aufzuzeigen und eventuelle blinde Flecken zu entdecken (Abschnitt 3). Im dritten Schritt führe

[1] http://www.fzi.de/index.php/de/forschung/forschungsbereiche/se/6783

ich eine Kritik dieser gefundenen Gewichtung vor und schlage eine Ergänzung des Forschungsprogramms vor (Abschnitt 4).

2 Problembereiche aus der Vogelschau

Hier eine mögliche Sicht, was insgesamt an Facetten zu bedenken und zu erforschen ist, wenn man die Langlebigkeit von Systemen stärken möchte.

Nicht jedem wird die Einteilung und die Formulierung gefallen – etwa, weil sie vielleicht eine Gewichtung nahelegt, die er oder sie nicht teilt, oder weil manche Themen als nicht softwaretechnischer Art empfunden werden. Ich hoffe aber, dass niemand hierin klaffende Lücken entdeckt oder Einträge, die er oder sie als komplett falsch einstufen würde.

- **1.1 Produktstruktur**: Wissenschaftliche Erkenntnis darüber, was langlebigkeitsgeeignete Software-Architekturen ausmacht.
- **1.2 Entwicklungsmethoden**: Wissenschaftliche Erkenntnis darüber, wie man solche Architekturen im gegebenen Einzelfall entwickelt und dauerhaft erhält. Das betrifft alle Teilaufgaben der Softwaretechnik von der Anforderungserhebung über den Entwurf und die Realisierung bis hin zu Verifikation/Validierung sowie dem Projekt-, Prozess-, Qualitäts-, Anforderungs- und Risikomanagement.
- **1.3 Produktinfrastruktur**: Wiederverwendbare Technologien, um Strukturen nach 1.1 einfach, verlässlich und schnell umsetzen zu können. Dies betrifft geeignete Programmiersprachen, generische Komponenten (wie Betriebssysteme, DBMS, Frameworks, Middleware und andere, deren Natur zum Teil vielleicht noch gar nicht entdeckt ist), sowie bereichsspezifische Komponenten.
- **1.4 Methodeninfrastruktur**: Softwarewerkzeuge zur technischen Unterstützung der Methoden von 1.2.
- **2.1 Investitionsentscheidung für die nötigen Einmalkosten**, z.B. Beschaffung von Werkzeugen und deren Einführung. Dies geschieht bei der jeweils betroffenen Softwareorganisation. Der wissenschaftliche Aspekt daran besteht darin, zu verstehen, wie solche Entscheidungen heute zustande kommen und zu beschreiben, wie sie idealerweise zustande kommen sollten.
- **2.2 Investitionsentscheidung für die nötigen Kosten im Einzelfall**. Das betrifft insbesondere die Bereitschaft, entsprechend qualifiziertes Personal zu bezahlen und diesem die nötige Zeit zum sauberen Verfolgen der Methoden von 1.2 zur Verfügung zu stellen. Wissenschaftlicher Aspekt analog zu 2.1.
- **3.1 Individuelle Möglichkeiten**: Die nötigen Fähigkeiten (Talent) und Fertigkeiten (Ausbildung und praktische Erfahrung) bei den einzelnen handelnden Softwareingenieur/inn/en, um die Grundlagen nach 1.1 bis 1.4 geeignet einsetzen zu können. Wissenschaftliche Fragestellungen in diesem Bereich betreffen die Beschreibung,

welches Maß an Fähigkeiten und Fertigkeiten in welchen Bereichen von 1.1 bis 1.4 nötig wäre, welches Maß an Fähigkeiten heute typisch ist und mit welchen Methoden man die Fertigkeiten optimal trainieren kann.

- **3.2 Individuelle Bereitschaft**: Die nötige Disziplin bei den einzelnen handelnden Softwareingenieur/inn/en, um die Grundlagen nach 1.1 bis 1.4 im konkreten Einzelfall (insbesondere unter Zeitdruck) auch tatsächlich hinreichend vollständig und korrekt einzusetzen. Wissenschaftliche Fragen in diesem Bereich: Wo mangelt es wie sehr an solcher Disziplin? Warum? Wie kann man das reduzieren?

3 Vergleich mit dem Workshop-Aufruf

Hier gebe ich die Themenliste aus dem Workshop-Aufruf wieder und markiere jeweils, welchen Bereichen der obigen Taxonomie ich die Einträge zuordne. Über diese Zuordnung mag man hier und da streiten, das ist nicht schlimm: Für die Zwecke dieses Artikels reicht eine ungefähre Zustimmung aus.

- Anpassbarkeit und Evolution
  - evolutionsfähige Software-Architekturen (1.1)
  - Anpassbarkeit von Anwendungen an wechselnde Plattformen, sich verändernde Anforderungen und Randbedingungen (1.1, 1.2)
  - Koevolution zw. Anforderungen, Architektur und Implementierung (1.1, 1.2)
  - Variabilität auf Anforderungs- und Architekturebene wie z.B. in Produktlinienansätzen (1.1, 1.2)
- Reengineering
  - Verfahren zur Erkennung von Legacy-Problemen und ihrer Ursachen (1.2)
  - Metamodelle für die Integration von Wissen über Software (1.2)
  - modellbas. Reengineering und Refactoring bestehender Systeme (1.2, 1.3, 1.4)
  - Modellextraktion durch Programmverstehen (1.2, 1.4)
- Entwicklungs-, Betriebs- und Wartungsmethoden
  - modellbasierte und architekturzentrierte Methoden und Techniken für (die Integration von) Entwicklung und Betrieb langlebiger Softwaresysteme (1.2, 1.4)
  - Business-IT-Alignment, geschäftsorientierte Evolution der IT (1.2, evtl. 2.2)
  - Lebenszyklus-übergreifendes Anforderungsmanagement (1.2)
  - Roundtrip-Engineering, integrierte Entwicklung auf Modell- und Codeebene (1.2, 1.4)
  - virtuelle Maschinen zur Erzielung von Plattformunabhängigkeit (1.1, 1.3)
  - kooperative Entwicklungsmethoden (1.2, evtl. 3.1/3.2), Kommunikations- und Kooperationswerkzeuge und Entwicklungsumgebungen, die die enge Zusammenarbeit von Domänenexperten, Anwendern und Softwareentwicklern unterstützen (1.4)

- Qualitätsmanagement
  - Qualitätsanforderungen an langlebige Softwaresysteme (1.2)
  - Qualitätsbewertung langlebiger Softwaresysteme (1.2)
  - Qualitätssicherung beim Betrieb langlebiger Softwaresysteme (1.2)

Wie man sieht, deckt der Workshop nur die Bereiche 1.1 bis 1.4 der Taxonomie ab (reine Software-Technik), aber berührt allenfalls zart die Sektoren 2.1/2.2 (Verhältnisse in SW-Organisationen) und 3.1/3.2 (individuelle menschliche Faktoren).

4 Kritik

Dies ist kein Übersichtspapier und deshalb werde ich gar nicht erst versuchen, die folgenden Aussagen mit wissenschaftlicher Literatur zu untermauern, sondern sie ganz als persönliche Ansichten formulieren:

1. Meiner Ansicht nach sind die weitaus wirksamsten Faktoren für die heute zu beobachtenden (und in der Tat heftigen) Langlebigkeitsprobleme in den Bereichen 2.2, 3.1 und 3.2 zu suchen.[2]

2. Selbst gewaltige Durchbrüche in den Bereichen 1.1 bis 1.4 würden daran nur graduell etwas ändern, weil selbst mit der bestmöglichen Technologie ein erheblicher Restwiderspruch zwischen "kurzfristig günstig" und "langfristig günstig" bestehen bleiben wird.[3] Diesen gibt es in so gut wie allen Lebensbereichen, auch den industriellen.

3. Folglich kann eine Wirksamkeit von Fortschritten in den Bereichen 1.1. bis 1.4 am besten sichergestellt werden, indem man zuvor oder zugleich die "Hygienefaktoren" 2.1 bis 3.2 studiert und sich bei der Bearbeitung von 1.1 bis 1.4 bestmöglich an die dort gefundenen (oder erzielten) Bedingungen anpasst.

4. Damit das gelingen kann, muss der Kreis der am Forschungsprogramm beteiligten um Personen ergänzt werden, die an der nötigen empirischen Forschung interessiert und in entsprechenden sozialpsychologischen und mikrosoziologischen Methoden versiert sind. Diese gibt es zwar nicht zahlreich, sie sind aber vereinzelt in diversen Fächern zu finden, z.B. in Informatik, Soziologie, Psychologie, Wirtschaftswissenschaften.

[2] Ich vermute, dass auch die große Mehrheit ganzheitlich denkender Praktiker dem zustimmt.

[3] Als eindrucksvolles Beispiel bietet sich aktuell die modellbasierte Softwareentwicklung an, die trotz ihrer großen potentiellen Vorteile nur sehr schleppend in der Praxis Fuß fasst, was zum Teil daran liegen dürfte, dass (a) der hilfreiche Einsatz nicht einfach ist (3.1) und (b) viele der Vorteile sich erst langfristig materialisieren (2.2, 3.2).