Technische Herausforderungen modellgetriebener Beherrschung von Prozesslebenszyklen aus der Fachperspektive: Von der Anforderungsanalyse zur Realisierung

Dissertation

zur Erlangung des Grades eines

Doktors der Naturwissenschaften

der Technischen Universität Dortmund

an der Fakultät für Informatik

von

Ralf Nagel

Dortmund

2009

Tag der mündlichen Prüfung:	21. Juli 2009
Dekan:	Prof. Dr. Peter Buchholz
Gutachter:	Prof. Dr. Bernhard Steffen
	Prof. Dr. Jakob Rehof

Es gibt nichts Praktischeres als eine gute Theorie

Immanuel Kant

Ich möchte mich ganz herzlich bei Prof. Dr. Bernhard Steffen bedanken. Seine fachlich kompetente und fürsorgliche Art der Unterstützung war mir oft eine große Hilfe bei dieser Arbeit. Das mir entgegengebrachte Vertrauen und die weitreichenden Freiheiten haben mich immer motiviert.

Bedanken möchte ich mich auch bei Prof. Dr. Jakob Rehof, der sich bereit erklärt hat, Zweitgutachter für meine Arbeit zu sein.

Weiterhin bedanke ich mich bei auch bei Prof. Dr. Heinrich Müller für die Übernahme des Vorsitzes der Prüfungskommission und Dr. Oliver Rüthing als weiteres Mitglied der Prüfungskommission.

Außerdem danke ich allen Mitarbeitern des Lehrstuhls für Programmiersysteme, die mit ihren Ideen und ihrer Arbeit das jABC-Projekt unterstützt haben. Weiterhin möchte ich mich herzlich bei Barbara, Stephan, Sven und Heinz für das geduldige Korrekturlesen dieser Arbeit bedanken.

Zusammenfassung

Der Alltag eines Menschen wird durch unterschiedlichste Arbeitsabläufe oder Prozesse bestimmt. Ob Kuchenrezept oder Bau eines Containerschiffs - mit zunehmender Komplexität eines Vorgangs werden Anweisungen benötigt, wie die Aufgabe zu bewältigen ist. Das jABC-Projekt bietet eine intuitive Möglichkeit zur grafischen Beschreibung von Prozessen. Insbesondere die konsequente Ausrichtung an der Zielgruppe der Nichtprogrammierer ist kennzeichnend für das jABC. Das auf Java basierende Komponentenmodell macht die Entwicklung neuer Bausteine für das System extrem einfach. Die Komponenten sind plattformunabhängig und universell. Das jABC kann deshalb mit jeder Form von Services zusammenarbeiten: vom Cobolprogramm bis hin zum Webservice. Durch den modularen Aufbau eignet sich das jABC sowohl für Alltagsprozesse als auch für *Modellgetriebene Softwareentwicklung* (MDSD). In Kombination mit dem ContentEditor und dem Tracer-, Localchecker-, Modelchecker und Codegenerator-Plugin entsteht ein ganzheitliches System zur Softwareentwicklung nach dem *One Thing Approach*. Das breite Einsatzspektrum dieses Ansatzes wurde in diversen Forschungs- und Industrieprojekten aufgezeigt.

Inhaltsverzeichnis

1 Einführung

In einem Industrieunternehmen existiert eine Vielzahl von sehr unterschiedlichen Arbeitsabläufen. Um Stahl zu produzieren, müssen Eisenerz, Kohle und andere Zusatzstoffe in bestimmter Qualität und Zusammensetzung am Weltmarkt eingekauft werden. Die Logistik muss dafür sorgen, dass diese Rohstoffe rechtzeitig am Produktionsstandort vorrätig sind. Das Stahlkochen im Hochofen folgt genauen Arbeitsabläufen. Nicht nur die Vorgänge der Logistikkette (Supply Chain Management) müssen beachtet werden. Von entscheidender Bedeutung sind dabei ebenso Prozesse wie Personalwesen (Human Resources) oder Rechnungswesen (Financial). Es ist einleuchtend, dass nicht jeder Mitarbeiter die ihm übertragenen Aufgaben irgendwie erledigen kann. Präzise Vorgaben sind nötig, damit alle Abläufe harmonisch ineinandergreifen und alles funktioniert.

Die Definition solcher als *Prozesse* bezeichneten Arbeitsabläufe ist die Aufgabe eines *Business Developers*[1]. Die Position des Business Developers ist in Unternehmen üblicherweise nicht mit Informatikern besetzt. In verschiedenen Projekten mit Partnern aus der Industrie stellten wir fest, dass Prozesse häufig mit ungeeigneten Werkzeugen dokumentiert werden (wie z.B. Microsoft Word, Powerpoint oder Visio). Damit die im Unternehmen eingesetzte Software diese Prozesse unterstützen kann, muss aus den Arbeitsabläufen eine formale Softwarespezifikation erstellt werden. Eine solche Spezifikation kann dann in die Entwicklung oder Anpassung der Unternehmenssoftware einfließen. Das Erstellen einer Software-Spezifikation gehört in den Aufgabenbereich eines Informatikers. Dabei ergeben sich mehrere Probleme:

- Ein Informatiker ist kein Spezialist für Unternehmensprozesse. Daher kann es zu Fehlern bei der Interpretation der Prozessdefinition kommen, die von einem Business Developer erstellt wurde.
- Analog ist ein Business Developer nicht mit einer formalen Software-Spezifikation wie z.B. in UML[2] vertraut. Kommunikationsfehler sind bei

[1] Die Bezeichnungen dieser Position sind in Unternehmen nicht einheitlich. Diese Position wird auch als Business Analyst, (Process-) Stakeholder, Business Process Management oder Business Process Design bezeichnet. In dieser Arbeit werde ich durchgehend die Bezeichnung *Business Developer* verwenden.

[2] Unified Modeling Language [Fow04]

diesem Vorgehen vorprogrammiert und schwer zu vermeiden.

- Änderungen der Prozessdefinition durch den Business Developer führen dazu, dass die Softwarespezifikation überarbeitet werden muss. Eventuelle Auswirkungen auf existierende Softwaresysteme oder andere Prozesse sind vom Business Developer kaum abzuschätzen.

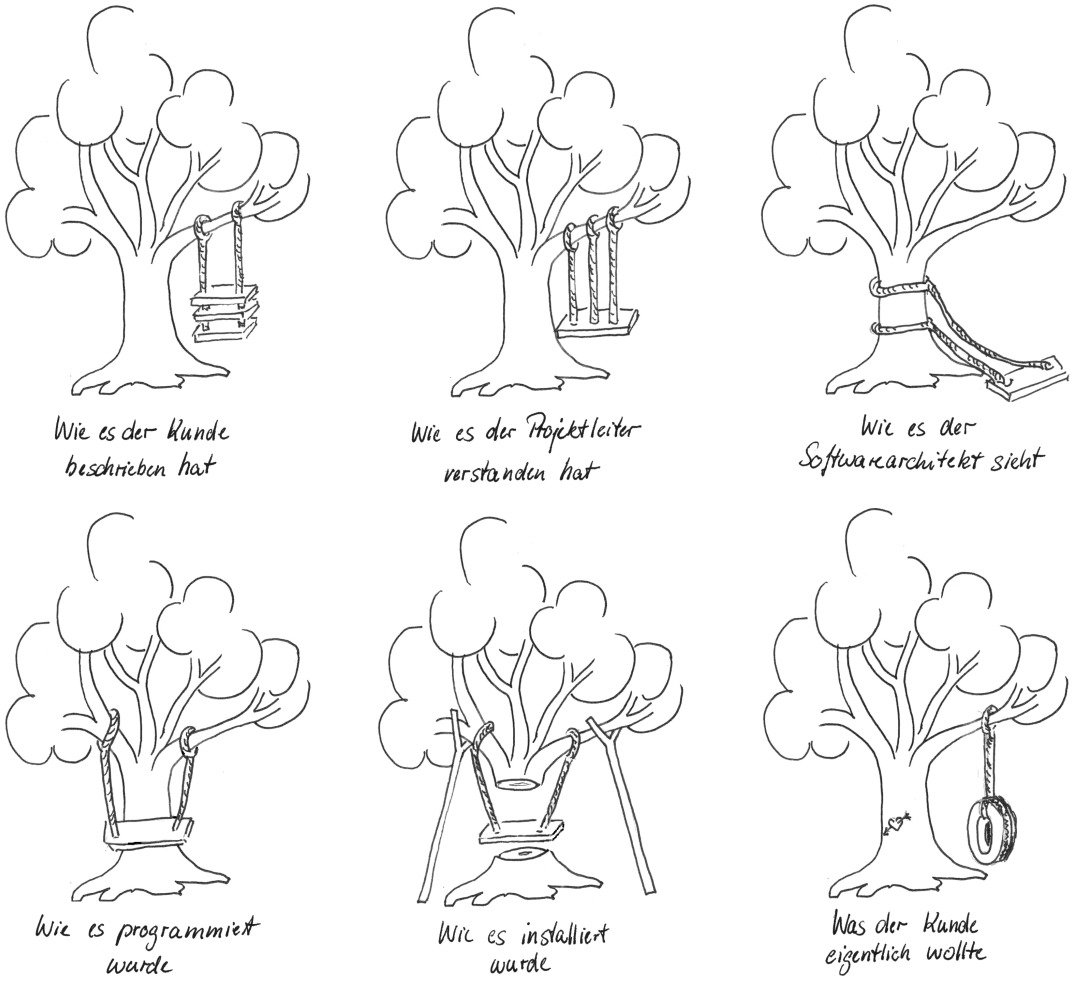

Abbildung 1.1: Cartoon: Softwareentwicklung

Die geschilderte Situation stellt ein in der Softwareentwicklung häufig auftretendes Problem dar und wurde bereits in Cartoons thematisiert (Abbildung 1.1). Zur Lösung benötigt man ein Softwarewerkzeug zur Beschreibung von Prozessen, das von Business Developern bedient werden kann und integraler Bestandteil der Softwareentwicklung ist.

Mit der Verwendung eines formalen Modells wie beim Ansatz der *modellgetriebenen Softwareentwicklung*[3] kann diese Aufgabe gelöst werden. Damit ein solches Modell vom Business Developer selbständig formuliert werden kann, muss die Grundlage der Modelle auch ohne Informatikkenntnisse leicht verständlich sein. Wir werden sehen, dass Kontrollflussgraphen sich dazu sehr gut eignen.

1.1 Grundlegende Anforderungen

Industrieunternehmen der verschiedenen Branchen setzen sehr unterschiedliche Softwaresysteme ein. Die Palette reicht von Standardsoftware, die zum Teil auf das jeweilige Unternehmen angepasst wurde, bis hin zu vollständigen Eigenentwicklungen. Auch das Alter der Software und der eingesetzten Technologien variiert stark. In einigen Unternehmen werden immer noch 30 Jahre alte Cobolprogramme eingesetzt. Mit wachsender Größe des Unternehmens sind auch immer größere IT und Entwicklungsabteilungen anzutreffen. Dort werden Softwarelösungen speziell für das eigene Unternehmen entwickelt. Diese Abteilungen setzen von nahezu vorbildlichen bis hin zu hochgradig ungeeigneten Arten der Spezifikationen ein. Häufig klagen sowohl IT als auch die Fachabteilungen dabei über Kommunikationsprobleme. Einige Werkzeuge, die zur Spezifikation oder Entwicklung eingesetzt werden, benötigen aufwändige Schulungen. Beim Wechsel der eingesetzten Werkzeuge ist bei den beteiligten Personen ein Umdenken erforderlich, um mit einer ungewohnten Notation oder Formensprache zurechtzukommen. Bei Standardsoftware kommt es vor, dass nicht die Software an die Anforderungen des Unternehmens angepasst werden kann, sondern die Arbeitsabläufe nach den Möglichkeiten der Software ausgerichtet werden. So steht die Unzulänglichkeit der Software optimalen Arbeitsabläufen im Weg. Bei Updates von Softwarepaketen, die an ein Unternehmen angepasst wurden, müssen diese Anpassungen bei einer neuen Version nochmals durchgeführt werden. Wurden solche Anpassungen nur unzureichend dokumentiert, führt dies in Folge zu massiven Problemen. Neben den eigentlichen IT Abteilungen werden relativ häufig auch in den Fachabteilungen in Eigenregie Softwareartefakte entwickelt, die weder versioniert noch im Unternehmen abgestimmt sind. Wächst das Unternehmen oder ändert sich der unternehmerische Schwerpunkt kommt es immer wieder zu Problemen, wenn die IT-Abteilung sich nicht schnell genug an neue Situationen anpassen kann.

Ausgehend von dieser Situation sollen die folgenden Anforderungen einige grundlegende Eigenschaften eines Werkzeugs zur Prozessdefinition sicherstellen:

[3] Model-Driven Software Development (MDSD) [MSUW04]

Anforderung G1 - Intuitivität

Der Verzicht auf eine komplexe Benutzeroberfläche oder Semantik reduziert den Schulungsaufwand beim Anwender. Daraus folgt, dass eine solche Software insbesondere von der angestrebten Zielgruppe, dem Business Developer, ohne besonderen Einarbeitungsaufwand eingesetzt werden kann. Eine intuitive Semantik, die sich an vertrauten Darstellungen anlehnt, sorgt dafür, dass die erstellten Modelle für Jedermann verständlich und nachvollziehbar bleiben.

Anforderung G2 - Agilität

Die Entwicklung von Prozessen muss eine agile Weiterentwicklung unterstützen. Als *agile Entwicklung* wird die Eigenschaft bezeichnet, bei der (Weiter-) Entwicklung von Unternehmenssoftware die permanenten Änderungen der Anforderungen durch gesetzliche Vorgaben, Firmenpolitik oder die Marktsituation zu berücksichtigen. Ziel ist es, diese Änderungen schnell und mit geringem Aufwand in die laufende Entwicklung einfließen lassen zu können.

Anforderung G3 - Konsistenz

Für die Modellierung wird vom Anfang bis zum Ende ein und dasselbe Modell verwendet, das alle notwendigen Informationen in sich vereint. Das bedeutet, dass zum Teil sehr unterschiedliche Aspekte in einem solchen Modell vereinigt werden müssen. Zu diesen Aspekten gehören die Spezifikation und Dokumentation bis hin zum Ausführungscode. Die Palette der Modellaspekte soll zusätzlich flexibel und variabel erweiterbar sein, um das System in einem breiten Anwendungsspektrum mit heterogenen Aspekten einsetzen zu können.

Anforderung G4 - Plattformunabhängigkeit

Bei der Entwicklung muss berücksichtigt werden, dass in Industrieunternehmen eine Vielzahl von zum Teil veralteten, unterschiedlichen Betriebssystemen und Hardwarearchitekturen eingesetzt werden. Softwarewerkzeuge, die nur für eine beschränkte Auswahl von Betriebssystemen verfügbar sind, engen ihren Einsatzbereich enorm ein. Software sollte, wo immer es möglich ist, für nahezu jede Computerplattform ohne zusätzlichen Portierungsaufwand verfügbar sein. Die Unterstützung selbst zukünftiger Computer-Plattformen garantiert dann auch eine hohe Zukunftssicherheit und Investmentschutz.

Anforderung G5 - Universalität

Die Vielzahl und Unterschiedlichkeit der in Industrieunternehmen eingesetzten Technologien muss berücksichtigt werden. Eine Spezialisierung auf ausgewählte Technologien würde den Einsatzbereich des Werkzeugs zu sehr einschränken. Das Komponenten-Konzept muss daher ein Adaptie-

ren neuer Technologien vorsehen und unterstützen.

Anforderung G6 - Skalierbarkeit
Je nach Unternehmensgröße variiert auch die Anzahl der Prozessdefinitionen, -versionen, -instanzen und -benutzer. Mit wachsender Anzahl der Prozesse muss auch der Einsatz adäquat skalierbarer Technologien unterstützt werden. Mit der Anforderung G5 ergibt sich daraus, dass Technologien nicht von einem Werkzeug vorgegeben werden dürfen. Das Werkzeug muss mit beliebigen, der jeweiligen Aufgabe angemessenen Technologien umgehen können.

Anforderung G7 - Korrektheit
Neben der reinen Lösung der Aufgabe müssen in vielen Fällen auch eine Reihe von gesetzlichen oder firmeninternen Auflagen erfüllt werden. Die Einhaltung dieser Vorgaben sollte von einem Werkzeug während der gesamten Entwicklungszeit des Prozesses automatisch überprüft werden. Dazu ist es notwendig, diese Auflagen in geeigneter Weise im System zu hinterlegen. Bei der Missachtung einer Auflage muss der Anwender darauf hingewiesen werden, sodass eine entsprechende Anpassung des Prozesses vorgenommen werden kann.

1.2 Erweiterte Anforderungen

Formale Modelle sind eine gut geeignete Ebene zur Spezifikation von Softwaresystemen, die zwischen einer unpräzisen natürlichsprachlichen Beschreibung und der Realisation als Sourcecode liegt. Dieser Ansatz wird als *Modellgetriebene Softwareentwicklung* bezeichnet. Formale Modelle können auch mit grafischen Benutzeroberflächen entworfen werden. Aktuelle Produkte von Apple[4] wie das Mac-Betriebssystem oder das iPhone sind berühmt für die intuitive Verständlichkeit der Benutzeroberfläche auch für technisch nicht versierte Menschen. Auf der anderen Seite zeigen VHS-Kurse zum Thema: „Umgang mit den elektronischen Fahrkartenautomaten der DB“[5], dass intuitive, an die Zielgruppe angepasste Benutzeroberflächen keine Selbstverständlichkeit sind.

Eine Anforderung an unser Werkzeug ist es, dass die Handhabung für die Zielgruppe des Business Developers intuitiv ist (*Anforderung G1 - Intuitivität*). Die Arbeitsweise soll daher an die bekannten Lego™-Bausteine erinnern. Der Mechanismus, mit dem Lego-Bausteine verbunden werden, ist sehr einfach und

[4] http://www.apple.com/
[5] Kurs der VHS Dortmund: „Umgang mit elektronischen Fahrkartenautomaten“, Januar 2009

bedarf kaum einer Erklärung. Mit ausreichend vielen Lego-Bausteinen in verschiedenen Formen und Farben können die unterschiedlichsten Modelle kreiert werden, vom Dinosaurier bis hin zur Raumstation.

Übertragen auf ein Werkzeug zur Modellierung von Prozessen bedeutet dies: Grundlage der Modellierung sind verschiedene Bausteine, die als *Komponenten* bezeichnet werden. Aus diesen Komponenten setzt der Anwender seine Modelle zusammen. Langjährige Erfahrung hat gezeigt, dass gerichtete Kontrollflussgraphen von den meisten Menschen intuitiv mit der richtigen Semantik gelesen werden. Als Beispiel wird hier das Plakat zur *Ersten Hilfe* des Deutsches Rotes Kreuz gezeigt (siehe Abbildung 1.2). Der dort abgebildete Prozess beim *Auffinden einer Person* muss für Jedermann intuitiv einleuchtend und ohne lange Erklärungen verständlich sein.

Abbildung 1.2: Auffinden einer Person (aus [Deu06])

Das Platzieren der grafischen Bausteine auf einer elektronischen Zeichenfläche per Drag and Drop bereitet selbst Computeranfängern heutzutage keine Probleme. Solche Gesten sind mit der Verbreitung von grafischen Benutzeroberflächen

den meisten Benutzern vertraut. Die Verbindung zweier Bausteine mit einer Kante stellt daher auch keine zusätzliche Herausforderung dar. Bei diesem Ansatz ist es klar, dass die Art und Weise der Bausteindefinition, die Bereitstellung der Bausteine in Paletten und die unterstützten Technologien zentrale Details des Entwurfs sind. Die nun folgenden erweiterten Anforderungen ergänzen die bisher definierten grundlegenden Anforderungen:

Anforderung E1 - Modellbasierter Ansatz

Im Gegensatz zu fließtextbasierten Beschreibungen kann ein maschinenlesbares formales Modell mit einer exakt definierten Semantik ausgestattet werden. Die Ungenauigkeiten von natürlichsprachlichen Spezifikationen durch Mehrdeutigkeit, unterschiedliche Interpretation und implizit vorausgesetzte Fakten werden bei der Verwendung eines Modells vermieden. Ein solches Modell ist mit seiner festgelegten Semantik die Grundlage für die spätere automatisierte Weiterverarbeitung und für Analysen.

Anforderung E2 - Grafische Modellierung

Die Verwendung einer grafischen Notation zur Beschreibung eines Modells erfüllt die Kombination der *Anforderung G1 - Intuitivität* und *Anforderung E1 - Modellbasierter Ansatz*. Der Anwender soll keine irgendwie geartete Syntax erlernen, sondern seine Spezifikation mit Hilfe einer intuitiv verständlichen Formensprache ausdrücken können. Durch frühzeitige Fehlermeldungen und Hinweise kann ein System den Anwender bei dieser Arbeit zusätzlich unterstützen. Gerade für neue Anwender sind solche Hilfen sehr nützlich und verkürzen die Einarbeitungsphase.

Anforderung E3 - Hierarchische Modellierung

Mit wachsender Größe der Modelle wird eine hierarchische Unterteilung in Teilmodelle notwendig. Dies unterstützt verschiedene Detaillierungsgrade und fördert den Überblick über das Gesamtmodell. Für den Anwender sollte es keinen Unterschied zwischen einem elementaren Baustein oder einem Teilmodell geben. Jedes (Teil-) Modell sollte als Baustein in einem anderen Modell eingesetzt werden können, sodass die Modelle dann auch leicht wiederverwendet werden können.

Anforderung E4 - Einfaches Komponentenmodell

Ein Anwender soll seine Modelle aus vorgefertigten Komponenten konstruieren. Dazu muss eine passende Palette von Komponenten zur Verfügung stehen. Aus der *Anforderung G5 - Universalität* folgt, dass jeweils für jeden Anwendungsfall eine spezialisierte Komponentenpalette bereitgestellt werden muss. Ein technologisch einfaches Komponentenmodell gewährleistet, dass die Bereitsstellung von Komponenten keinen Overhead für die Gesamtentwicklungszeit bedeutet. Der Einsatz von Standardtechnologien

kann die Einarbeitungszeiten in das Komponentenmodell im Vergleich zu proprietären Ansätzen zusätzlich verringern.

Anforderung E5 - Service-Orientierung
Neben der Einfachheit des Komponentenmodells (Anforderung E4) ist der Service-orientierte Ansatz von zentraler Bedeutung. Die Komponenten im Modell repräsentieren unterschiedlichste Services. Services sind austauschbare, wiederverwendbare, in sich abgeschlossene und zustandslose Funktionalitäten. Im Gegensatz zu (Java-) Komponenten sind Services nicht objektorientiert und unabhängig von der eingesetzten Programmiersprache. Die Verwendung von Services als Baukasten ist die grundlegende Idee des *Service Centered Continuous Engineering* (SCCE) [MS06].

Anforderung E6 - Semantische Komponentenverwaltung
Für die Modellierung umfangreicher und komplexer Systeme sind entsprechend große Komponentenpaletten notwendig. Wird ein solches System als Zusammenarbeit mehrerer Modellierer entworfen, ergeben sich unterschiedliche Spezialisierungen und Fokussierungen. Um die Arbeit der verschiedenen Anwender optimal zu unterstützen, muss die Komponentenpalette in Umfang und Strukturierung an das jeweilige Spezialgebiet des Anwenders angepasst werden können. Dies führt unter Umständen dazu, dass ein und dieselbe Komponente von mehreren Anwendern mit unterschiedlichen Namen, Piktogrammen oder Bedeutungen belegt werden.

Anforderung E7 - Statusspezifische Implementierungen
Durch die Arbeit mit einem konsistenten Modell (*Anforderung G3 - Konsistenz*) während des gesamten Entwicklungszyklus werden unterschiedliche Anforderungen an die Implementierungen gestellt. Steht in frühen Designphasen eine symbolische Ausführung im Vordergrund, werden mit zunehmendem Detaillierungsgrad des Modells auch die Implementierungen genauer. Letztendlich muss beim fertigen Produkt weiterhin zwischen der Test-, Abnahme- und der Produktionsumgebung unterschieden werden (z.B. wegen unterschiedlichen Servern und Datenbanksystemen). Diese verschiedenen Realisierungen sollen aber durch ein und dieselbe Komponente gebündelt werden. Die Art des Zugriffs auf die Komponente muss festlegen welche Implementierung verwendet werden soll.

1.3 Technische Anforderungen

Viele der heute gängigen Softwaresysteme bieten eine Programmier- oder Plug-in-Schnittstelle, über die diese Systeme erweitert werden können. Exemplarisch seien hier drei Beispiele mit sehr unterschiedlichen Schnittstellen genannt:

- Microsoft-Officeanwendungen können mit eigenen .NET-Funktionen erweitert werden.
- Adobe Photoshop bietet eine Pluginschnittstelle, über die zusätzliche Filterfunktionen geladen werden können.
- Alle Funktionen der Eclipse-IDE sind als Plugins realisiert.

Die Erweiterbarkeit von Softwaresystemen ist allerdings keine neue Idee. Bereits im Jahr 1984 bot der Texteditor Emacs[6] die Möglichkeit, durch zusätzliche Lisp-Makros den Funktionsumfang zu erweitern. Komplexe Plugin-Frameworks wie OSGi [WHKL08][7], das in der Eclipse-IDE eingesetzt wird, erfordern allerdings auch eine längere Einarbeitungsphase. Durch die technische Weiterentwicklung ist der Notebookarbeitsplatz in Flugzeug und Bahn zu einem alltäglichen Bild geworden. Trotz der hohen Abdeckung mit Mobilfunknetzen ist nicht immer der Zugriff auf das Internet, Datenbanken oder andere Infrastrukturen gegeben. In vielen Programmen wurden daher zusätzliche Offlinefunktionen integriert, um auch ohne Netzwerkverbindung weiterarbeiten zu können. Die folgenden technischen Anforderungen sollen unter anderem den mobilen Einsatz und die Erweiterbarkeit der Software garantieren:

Anforderung T1 - Framework-Architektur

Um ein Softwaresystem an variable Anforderungen anpassen zu können, liegt die Verwendung einer flexiblen Architektur auf der Hand. Flexibilität bedeutet hier, dass nicht jeder Anwender alle verfügbaren Funktionen einer Software benötigt und es sogar Bedarf für Funktionen gibt, die erst in der Zukunft entwickelt werden. Mit der Verwendung eines Framework-Pluginkonzepts wird der Funktionsumfang der Software durch Hinzufügen oder Weglassen von Plugins anpassbar. Das Entwickeln neuer Plugins und deren nachträgliche Installation wird dadurch ermöglicht. Das Framework ist als zentrale Basis des Systems immer vorhanden.

Anforderung T2 - Einfache Plugins

Mit der Verbreitung von Plugin-Konzepten sind eine große Vielfalt von unterschiedlichen Technologien und Implementierungen entstanden. Nur wenige Standards konnten sich in den letzten Jahren etablieren. Dies liegt unter anderem daran, dass Plugins wegen unterschiedlichen internen APIs nicht zwischen verschiedenen Systemen austauschbar sind. Um eine große Zahl von Entwicklern zu motivieren, eigene Plugins zu erstellen, muss das Werkzeug eine einfache Schnittstelle aufweisen. Eine kurze Einarbeitszeit ermöglicht dann schnelle Erfolgserlebnisse.

[6] http://www.gnu.org/software/emacs/

[7] OSGi Allianz, früher *Open Services Gateway initiative*[OSG09]

Anforderung T3 - Service-Entkopplung

Durch eine lose Kopplung zwischen einer Komponente und dem assoziierten Service können Komponenten auch dann noch in Modellen verwendet werden, falls die Verbindung zum Service unterbrochen wird. Dies kann durch fehlende Softwarebibliotheken auf verschiedenen Computern oder eingeschränkte Konnektivität verursacht werden. Die Entkopplung ermöglicht die Weiterarbeit an einem Modell z.B. auf einem Notebook, auch wenn die Ausführung wegen fehlender Services temporär nicht möglich ist.

Anforderung T4 - Fehlertoleranz

Die Modelle unseres Werkzeuges sollen auch dann noch geladen werden können, falls eine oder mehrere Komponenten nicht mehr verfügbar sind. Dies kann beim Wechsel zwischen verschiedenen Computern oder beim Laden älterer Modelle vorkommen. Durch die Weiterentwicklung von Komponenten kann es zusätzlich zu inkompatiblen Kombinationen zwischen Modell und verwendeten Komponenten kommen. Falls dies auftritt, muss das Modell geladen und der Anwender auf dieses Problem hingewiesen werden.

Anforderung T5 - Ausführbarkeit

Die Art der Ausführung von Modellen verändert sich mit fortschreitenden Fertigstellung der Entwurfsarbeiten. In frühen Phasen hilft dem Anwender eine symbolische Ausführung direkt aus dem Werkzeug heraus. Ein Modell sollte auch dann ausführbar sein, wenn es aus einer Mischung von unfertigen und bereits fertiggestellten Komponenten besteht. Stößt der Interpreter bei seiner Arbeit auf eine unfertige Komponente, sollte dies als *leere Implementierung* gewertet werden. Je weniger unfertige Komponenten im Modell vorhanden sind, umso größere Bereiche des Modells können dann ausgeführt werden.

Anforderung T6 - Codegenerierung

Ist ein Modell vollständig beschrieben und enthält es keine unfertigen Komponenten mehr, muss es unabhängig vom Werkzeug ausführbar sein. Dazu muss das Modell durch Codegenerierung in äquivalenten, vom Werkzeug unabhängigen Sourcecode übersetzt werden. Dieser compilierte Code ist dann autark ausführbar.

Anforderung T7 - Versionierbarkeit

Die erzeugten Modelle sollen mit üblichen Versionierungssystemen[8] verarbeitet werden können. Durch eine Versionierung der Modelle werden die Änderungen an den Modellen nachvollziehbar und können zwischen den Mitgliedern eines Entwicklerteams ausgetauscht werden.

[8] auch als Revision Control Systeme (RCS) bezeichnet

1.4 Meine Arbeitsschwerpunkte

In dieser Arbeit wird ein Java-Framework zur modellgetriebenen Entwicklung von Prozessen (jABC) vorgestellt. Die Schwerpunkte meiner Arbeit lagen dabei in den folgenden Teilbereichen:

jABC-Gesamtarchitektur
: Der Entwurf und die Realisation der Softwarearchitektur des jABCs ist ein wesentlicher Schwerpunkt meiner Arbeit. War die erste Version des jABCs noch als Client-Server-Architektur konzipiert, besteht die Applikationsebene des jABC heute aus drei Anwendungen: dem Prozesseditor, einer IDE und einem RCS-System (vgl. Kapitel 3.2 auf Seite 20). In Zukunft wird diese Ebene um eine vierte Anwendung erweitert: den Update-Server (vgl. Kapitel 7.2 auf Seite 99). Neben der Gesamtarchitektur verantworte ich auch die Architektur der zentralen Kernbibliotheken: dem Framework und der Editoranwendung (vgl Kapitel 3.5 auf Seite 26 und Kapitel 3.6 auf Seite 28). Der modulare Aufbau des Prozesseditors macht die Spezialisierung des jABCs durch Plugins erst möglich (vgl Kapitel 3.7 auf Seite 35).

Java Komponentenmodell
: Zweiter wesentlicher Teil meiner Arbeit ist der Entwurf und die Weiterentwicklung des jABC-komponentenmodells (vgl. Kapitel 3.8 auf Seite 42). Durch die Erfahrungen in diversen Forschungs- und Drittmittelprojekten und den dadurch ständig wechselnden Anforderungen an die Bausteine des jABC, fließen immer neue Aspekte in die Definition des Komponentenmodells ein (vgl. Kapitel 7.1 auf Seite 97).

ContentEditor
: Mit der Idee, dem Entwurf und der Realisation des ContentEditors, habe ich weiterhin einen eigenständigen Editor geschaffen, mit dem strukturierte Informationen bearbeitet werden können. Der Editor kann durch unterschiedliche jABC-Modelle leicht auf verschiedene Anwendungsszenarien angepasst werden. Diese sogenannten ContentEditor-Grammatikmodelle sind ein Beispiel für eine alternative Graphsemantik im jABC (vgl. Kapitel 4 auf Seite 55).

2 Wissenschaftlicher Hintergrund

Wie die Finanzkrise im Herbst 2008 gezeigt hat, können sich die wirtschaftlichen Anforderungen durch radikale Veränderung der Marktsituation in kürzester Zeit ändern. In der aktuellen Krise wurden innerhalb weniger Tage die Vergaberichtlinien für Kredite neu ausgerichtet. Halbautomatische oder automatische Scoringsysteme in den betroffenenen Kreditinstituten mussten schnellstmöglich angepasst werden. Es zeigt sich, dass dabei klassische Vorgehensmodelle wie das *Wasserfallmodell* agilen Ansätzen wie *Scrum* [TN86] unterlegen sind. Im Folgenden wird der agile Ansatz vorgestellt, für den das jABC entwickelt wurde.

Die *Lightweight Process Coordination* (LPC) [MS04] ist ein Entwicklungsprozess, in dem aus vorproduzierten Bausteinen neue Systeme zusammengesetzt werden. Die verwendeten Bausteine sind grob-granulare, technologieunabhängige Funktionsbausteine für eine festgelegte Anwendungsdomäne. Das Vorgehensmodell ist so einfach, dass dafür kein Informatik- oder technologiespezifisches Wissen benötigt wird. Zielgruppe von LPC ist der *Anwendungsexperte*, der über detailliertes Wissen der Anwendungsdomäne verfügt. Da ein formales Modell im Zentrum der Entwicklung steht, gehört LPC zu den *modellgetriebenen Entwicklungsprozessen* (MDSD). LPC-Modelle verfügen über eine intuitive, grafische Kontrollflusssemantik, sodass eine Schulung des Anwendungsexperten nahezu überflüssig ist. Sowohl grafische Semantik als auch die zur Verfügung stehenden Bausteine entstammen der Anwendungsdomäne des Anwenders. Im Fokus der LPC-Modelle stehen die Prozesse der Fachabteilungen aus der Anwendersicht. In [HMM+08] wird gezeigt, wie ein Dokumentmanagementprozess eines international agierenden Möbeldiscounters mit LPC modelliert wurde. Mit LPC gelingt auch eine Trennung zwischen dem Prozess, der als geistiges Eigentum eines Unternehmens unter Umständen einer Geheimhaltung unterliegt und daher nur innerhalb des Unternehmens gepflegt wird und den Services, die als Dienstleistung oder Produkt kostengünstig von entsprechenden Anbietern extern eingekauft werden können (siehe Abbildung 2.1 auf der nächsten Seite [JKN+06]). Für die Umsetzung eines LPC-Prozesses sind besonders *Serviceorientierte Architekturen* geeignet. Durch die Prozessebene werden die Services orchestriert und die Benutzeroberfläche gesteuert. Im Fall der *Electronic Tool Integration Platform* (jETI) [SMN05], [MNS05a] wird der jABC-Prozesseditor dann sogar

zur Benutzeroberfläche des Anwenders und steuert komplexe Abläufe und Benutzerinteraktionenen durch jABC-Modelle.

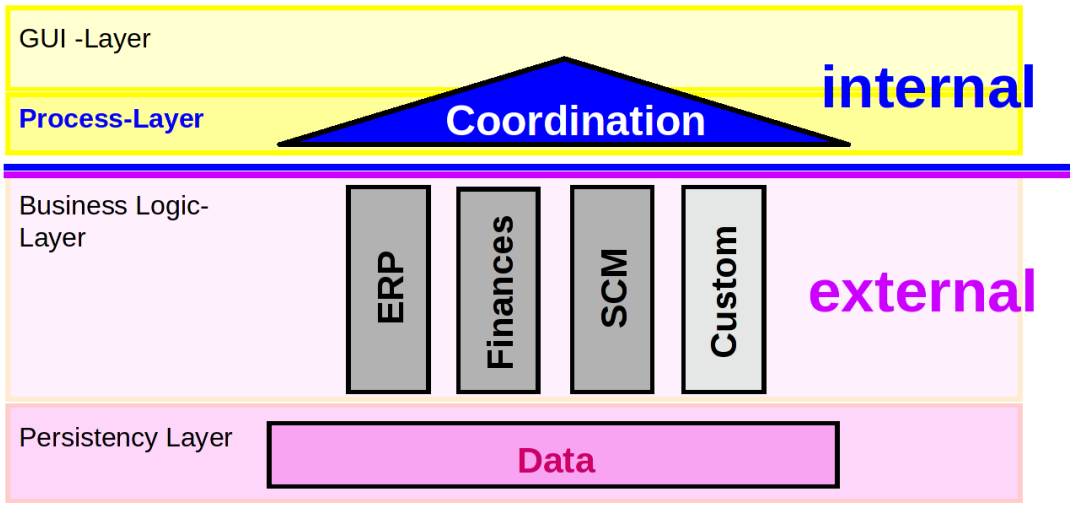

Abbildung 2.1: Lightweight Process Coordination

Für einen *agilen, modellgetriebenenen Softwareentwicklungsprozess* ist es wesentlich, die Konsistenz zwischen Modell und Softwarelösung zu gewährleisten (Anforderung G3: Konsistenz). Das *extreme Model Driven Design* (XMDD) [MS08] setzt daher einen vollständig automatischen Codegenerierungsprozess voraus. Alle Änderungen werden ausschließlich am XMDD-Modell vorgenommen, der generierte Sourcecode ist für manuelle Änderungen tabu. Weiterhin setzt XMDD ein technologieunabhängiges Prozessmodell voraus. Das Technologie-Mapping erfolgt erst durch die verwendeten Bausteine und den Generierungsprozess. Der Modellierer wird davon befreit, sich den Anforderungen einer spezifischen Technologie zu unterwerfen. Das entstehende Prozessmodell ist eine universelle Lösung in der spezifischen Anwendungsdomäne und kann mit unterschiedlichen Technologien realisiert werden. Als Bausteine eignen sich insbesondere Services, die nur lose mit dem Modell gekoppelt werden [MS06]. Die Implementierung der verwendeten Bausteine ist unabhängig vom Modell und der eingesetzten Methodik. Bausteine können mit klassischen Methoden programmiert oder aus Schnittstellenbeschreibungen wie WSDL [W3C01] generiert werden. Wie beim SIB Adapter-Entwurfsmuster (vgl. Kapitel 3.8.4 auf Seite 47) ergibt sich eine Trennung zwischen dem eigentlichen Service und dem Modelladapter, der den Service zum Bestandteil des Modells werden lässt. Solche serviceorientierten Bausteine sind austauschbar. Ein Technologiewechsel erfolgt ohne Anpassung des eigentlichen Prozessmodells.

Durch lokale und globale Verifikationsmethoden werden alle Prozesse einer ständigen Überprüfung unterzogen. Grundlage der Verifikation sind firmeninterne oder gesetzliche Vorgaben. Die automatische Prüfung der Modelle im Hintergrund informiert den Modellierer, sobald eine der festgelegten Regeln verletzt wurde. Somit kann die Verifikation zu einer wertvollen Hilfe bei einer späteren Abnahme der generierten Lösung werden. Auch die Verifikationsregeln müssen wegen gesetzlicher Änderungen oder Anpassungen an die Marktsituation permanent weiterentwickelt und verfeinert werden. Je mehr Eigenschaften durch automatische Verifikationsmethoden bereits am Modell nachgewiesen wurden, umso weniger Aufwand muss in Testverfahren bei der Abnahme einer neuen Prozessversion investiert werden.

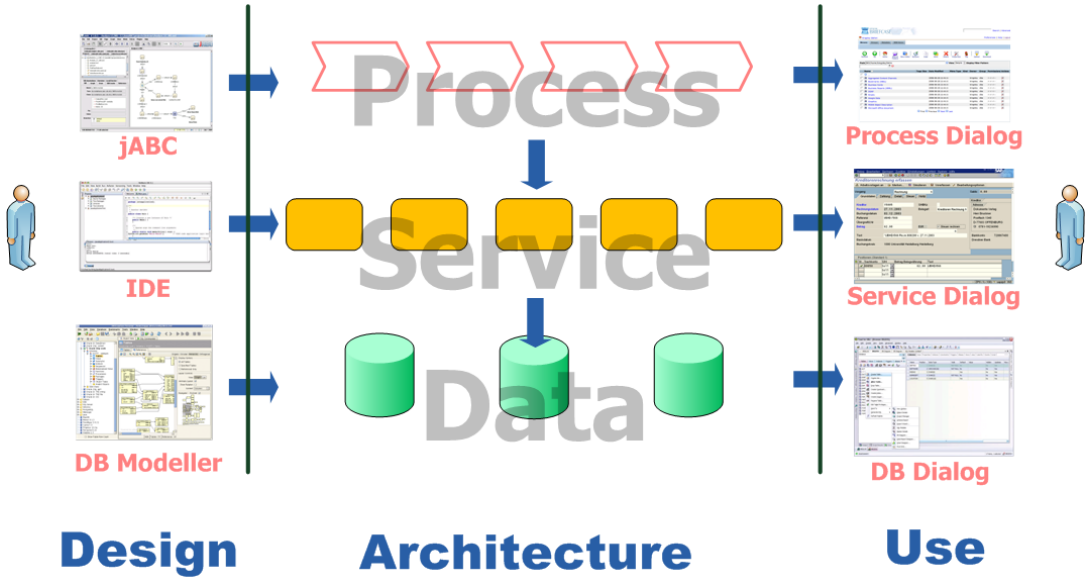

Abbildung 2.2: XMDD serviceorienterte Architektur

Der *One-Thing-Approach* [SN07] vereinigt alle unterschiedlichen Aspekte des *agilen, modellgetriebenen Designs* zu einem einzigen formalen Modell. Dieser ganzheitliche Ansatz mit einem allumfassenden Modell beinhaltet alle Facetten des zu beschreibenden Systems. Das bedeutet allerdings nicht, dass es sich zwingend um eine einzelne Datei handeln muss. Das One-Thing-Modell beinhaltet verschiedene Teilmodelle und Aspekte. Ein OTA-Modell kann folgende Aspekte enthalten:

- die Anwenderprozesse
- die Organisationsstruktur
- die Datenstruktur

- die Softwarearchitektur
- die Verifikationsanforderungen
- den Codegenerierungsprozess
- die Dokumentation

Die OMG unterscheidet bei ihrer *Model Driven Architecture* (MDA) die Trennung zwischen dem Beschreibungsmodell (CIM[1]), dem Anwendungsmodell (PIM[2]), dem plattformspezifischen Modell (PSM[3]) und dem Code-Modell (ISM[4]). Beim *One-Thing-Approach* sind diese Aspekte Bestandteil des OTA-Modells. Die Hierarchie eines OTA-Modells für das genannte Beispiel der Stahlindustrie aus der Einführung könnte im OTA wie folgt strukturiert sein:

Domänenmodelle oder CIM
: Die Domänenmodelle enthalten die Top-Level Unternehmensprozesse wie *Supply Chain Management*, *Human Resources* und *Financial*. Diese Modelle sind eine Art Inhaltsverzeichnis aller Unternehmensprozesse.

Fachmodelle oder PIM
: Ein Business Developer entwickelt Verfeinerungen wie beispielsweise den *Neuer Auftrag*-Prozess im *Supply Chain Management*. Diese Prozesse steuern die Arbeitsabläufe der einzelnen Fachabteilungen.

Komponentenpalette
: Durch die Komponentenpalette werden die Bausteine der Fachmodelle auf reale Implementierungen abgebildet. In unserem Beispiel könnte der *Neuer Auftrag*-Prozess auf die SAP-*Business Application Programming Interface* (BAPI) abgebildet werden (ersetzt PSM und ISM).

Ein OTA-Modell kombiniert drei Dimensionen bei der modellgetriebenen Entwicklung von Prozessen: Die *horizontale Dimension* sind die Domänenmodelle, die alle in einem Unternehmen existieren globalen Prozesse enthalten. Die *vertikale Dimension* enthält die Verfeinerungen der einzelnen Prozessschritte in den Fachmodellen. Dabei findet ein Übergang von technologieunabhängigen Fachmodellen zu technologiespezifischen Komponentenbausteinen statt. Die *temporale Dimension* beinhaltet die Evolution der Prozesse, von der Anforderungsanalyse bis hin zur Realisation. Mit dem jABC wurde ein Framework realisiert, mit dem alle Dimensionen eines OTA-Modells von einem Fachexperten erfasst werden können.

[1] Computation Independent Model
[2] Platform Independent Model
[3] Platform Specific Model
[4] Implementation Specific Model

3 Java Application Building Center

Das jABC-Projekt kann inzwischen auf eine 15-jährige Geschichte zurückblicken. Bereits 1993 wurden mit der *DaCapo*-Umgebung [SFC+94] Funktionen realisiert, die noch heute ein Bestandteil von *jETI* [MNS05b] sind. Die nächste Generation wurde dann unter dem Namen *META-Frame* [SMC95] veröffentlicht. Mit dem *Agent Building Center* [MS04] entstand 1998 ein System, das zur Basis vieler weiterführender Projekte wurde. Dazu gehören:

ETI
: Electronic Tool Integration [MS07]

OCS
: Online Conference Service [KM06]

ITE
: Integrated Test Environment [MNS02]

MaTRICS
: Management Tool for Remote Intelligent Configuration of Systems [BM06]

Seit April 2003 bin ich für die Weiterentwicklung des Projektes verantwortlich. Eine der markantesten Eigenschaften des jABC-Projektes ist die strikte Fokussierung im Software-Entwicklungsprozess auf die Zielgruppe des Business-Developers. Ein Business-Developer (auch als Anwendungsexperte bezeichnet) verfügt in der Regel über keine Informatik-Kenntnisse. Trotzdem soll er mit dem jABC prozess- und serviceorientierte Modelle beschreiben, die zu einem integralen Bestandteil der Softwarelösung werden. Anders als bei anderen Werkzeugen muss der Anwender dafür aber keine Syntax oder Semantik[1] erlernen, um das jABC einsetzen zu können. Allein durch die Einfachheit und Intuitivität der grafischen Benutzeroberfläche und der eingesetzten Formensprache des jABCs wird dies in der Praxis gewährleistet. In Forschungsprojekten konnten ungeschulte Anwender bereits nach 30 Minuten selbständig mit dem jABC arbeiten.

[1] wie z.B. bei UML, BPEL oder eEPK

3.1 Designentscheidungen

In der Einführung (Kapitel 1 auf Seite 1) wurden eine Reihe von Anforderungen aufgestellt, die in das Design und die Architektur des jABCs eingeflossen sind. Diese Anforderungen führten zu folgenden Designentscheidungen:

Java/JVM
Als technologische Grundlage des jABC-Entwurfs wurde *Java* und die *Java Virtuelle Maschine* (JVM) ausgewählt. Durch die hohen Kompatibilitätsanforderungen an die JVM-Implementierungen auf verschiedenen Hardwarearchitekturen und Betriebssystemen wird eine uneingeschränkte Portabilität des Java-Bytecodes garantiert. Kompatibilitätsprobleme durch spezifische Erweiterungen verschiedener Hersteller wie bei C oder SQL sind bei den JVM-Implementierungen unbekannt. Wegen der freien Verfügbarkeit sowohl der Entwicklungstools als auch der JVM (inklusive der Sourcen) entstehen keine Lizenzkosten und es wird eine hohe Investitions- und Zukunftssicherheit erzielt, da eine Migration auf kommende Hardwarearchitekturen oder Betriebssysteme entfällt. Zusätzlich wird durch neue Programmiersprachen wie JRuby [NE09], die auch die JVM verwenden und Java-kompatiblen Bytecode erzeugen, die Möglichkeit geschaffen, Teile des jABC-Systems mit verschiedenen Programmiersprachen zu erweitern. Weiterhin gibt es für Java als *General Purpose Programmiersprache* eine permanent wachsende, reiche Palette von freien oder kommerziellen Bibliotheken für die unterschiedlichsten Aufgaben. Mit Bibliotheken kann sogar auf Funktionen zugegriffen werden, die nicht mit Java realisiert oder lokal verfügbar sind. Zur Illustration seien hier nur einige Beispiele genannt:

Java Database Connectivity (JDBC)
Über JDBC können Daten in nahezu allen relationalen Datenbanken abgelegt werden.

Java SOAP
Für den Umgang mit Webservices gibt es eine ganze Reihe von Lösungen wie Axis oder JAX-RPC.

Java Native Interface (JNI)
Mit JNI kann nativer C/C++-Code aufgerufen werden.

Java Runtime
Über die Runtime-Biblitohek können native ausführbare Programme des Host-Computers aufgerufen werden.

Die Java-Plattform erfüllt somit die Anforderungen Plattformunabhängigkeit (G4) und Universalität (G5).

Grafische Modellnotation

Wie beim *Model Driven Design* (MDD) basiert das jABC auf Modellen, die rein grafisch bearbeitet werden (Anforderung E1: Modellbasierter Ansatz und E2: Grafische Modellierung). Die verwendete Graphsyntax ist minimalistisch: Ein Graph besteht aus Knoten und gerichteten, benannten Kanten. Für die Knoten können beliebige Piktogramme verwendet werden. Anders als z.B. bei eEPKs[2] gibt es keine Vorgaben für die verwendeten Piktogrammformen oder deren Bedeutungen. Für Kanten werden keine unterschiedlichen Formen unterschieden. Jeder Knoten definiert eine Liste von Branchnamen. Diese Branches dienen als Vorgaben für die Benennung der ausgehenden Kanten dieses Knotens. Viele Anwender empfinden diese Syntax als intuitiv oder gewöhnen sich sehr schnell an sie (Anforderung G1: Intuitivität). Modelle können wiederum als Bausteine in anderen Modellen verwendet werden (Anforderung E3: hierarchische Modellierung). Die Speicherung der Modelle in XML-Dokumenten ermöglicht eine einfache Versionierung mit üblichen RCS-Systemen (Anforderung T7).

SIB-Komponentenmodell

Das Komponentenmodell des jABCs basiert auf einer sehr einfachen Abbildung zwischen Java-Sprachkonstrukten und den Eigenschaften einer SIB-Komponente (Anforderung E4: Einfaches Komponentenmodell). Durch die Programmiersprache Java ist die service-orientierte Verwendung von skalierbaren Technolgien sehr leicht möglich (Anforderungen G6: Skalierbarkeit, E5: Service-Orientierung). Das SIB-Komponentenmodell unterstützt eine agile Weiterentwicklung (Anforderung G2: Agilität), da SIB-Komponenten sowohl *Top Down* als auch *Bottom Up* entwickelt werden können (vgl Kapitel 3.8 auf Seite 42). Verschiedene Java-Interfaces ermöglichen, unterschiedliche Aspekte der Implementierung in einer Komponente zu vereinigen (Anforderung T5 Ausführbarkeit, T6: Codegenerierung, E7: Statusspezifische Implementierungen). Eine Entkopplung zwischen SIB und Service wird durch die Verwendung des *SIB-Adapter-Entwurfsmusters* erreicht (Anforderung T3: Service-Entkopplung). SIB-Komponenten können in anpassbaren Taxonomien verwaltet werden, sodass der Benutzer sich darin optimal zurechtfindet (Anforderung E6: Semantische Komponentenverwaltung).

Fehlertoleranz

Das jABC bietet verschiedene Mechanismen, um mit fehlenden, fehlerhaften oder inkompatiblen Bestandteilen von Modellen umgehen zu können (Anforderung T4: Fehlertoleranz). Folgende Situationen werden berücksichtigt:

[2] erweiterte Ereignisgesteuerte Prozesskette (eEPK)

Fehlende Plugins

Der Plugin-Container lädt beim Start alle Plugins, soweit diese verfügbar sind. Auf fehlende Plugins wird durch eine Fehlermeldung hingewiesen. Sobald ein fehlendes Plugin wieder verfügbar ist, wird es beim nächsten Start wieder geladen. Ein spezieller Mechanismus im SIB-Container sorgt dafür, dass SIBs immer geladen werden können unabhängig davon, ob die dazu nötigen Plugins geladen wurden. Ein SIB, das Localcheck-Code enthält, kann daher auch dann geladen werden, falls das LocalCheck-Plugin nicht mitgeladen wurde. Diese Entkopplung zwischen SIBs und Plugins macht beliebige Kombinationen von SIB-Paletten und Plugins möglich (vgl. Kapitel 3.9 auf Seite 52).

Fehlende SIBs

Beim Laden eines Modells müssen alle verwendeten SIBs zur Verfügung stehen. Fehlt ein SIB oder ist es fehlerhaft, wird dieses SIB durch ein spezielles Platzhalter-SIB ersetzt. Das sogenannte Proxy-SIB hat dazu die Fähigkeit beliebige Parameter- und Branchwerte zu speichern. Beim Speichern werden alle Informationen wieder so rekonstruiert, als ob das SIB niemals gefehlt hätte. Sobald das SIB wieder erreichbar ist, kann das Modell wieder vollständig geladen werden (vgl. Kapitel 3.9 auf Seite 52).

Fehlende Services

Die Verwendung des SIB-Adapter-Entwurfsmusters entkoppelt die SIB-Implementierung von den eingesetzen Services (vgl. Kapitel 3.8.4 auf Seite 47).

Erweiterbares Modell

Das jABC-Modell kann um beliebige Informationen erweitert werden. Diese Zusatzinformationen können entweder direkt durch Plugins oder vom Benutzer mit dem AnnotationEditor zum Modell hinzugefügt werden (vgl. Kapitel 4 auf Seite 55). SIBs, aus denen die Modelle bestehen, können wiederum mehrere Implementierungen durch verschiedene Plugin-Interfaces enthalten. So kann ein jABC-Modell durchgehend vom Prototyp bis hin zur Fertigstellung genutzt werden (Anforderung G3 - Konsistenz). Auch die agile Weiterentwicklung profitiert davon, dass alle Informationen zentral in nur einem Modell zusammengeführt sind. Dieser Aspekt ist ein wesentlicher Bestandteil des *One-Thing-Approachs* [MS08].

Framework-Architektur

Durch seine Framework-Architektur kann das jABC einfach um neue Funktionen erweitert werden (vgl. Kapitel 3.3 auf Seite 23). Plugins im jABC sind Java-Klassen, die ein vorgegebenes Interface implementieren müssen. Die Programmierung eines Plugins ist, wie bei SIB-Komponenten,

sehr einfach gehalten (vgl. Anforderungen T1: Framework-Architektur: und Anforderungen T2: Einfache Plugins). Verschiedene Plugins tragen dazu bei, einige Anforderungen abzudecken (vgl. Kapitel 3.7 auf Seite 35).

Tracer
Das Tracer-Plugin kann Modelle, die aus entsprechenden SIBs bestehen, interpretieren und ausführen. Ein Anwender kann die Ausführung manuell oder durch Haltepunkte unterbrechen. Der Tracer arbeitet entweder integriert in die Oberfläche des Prozesseditors oder als eigenständige Anwendung ohne grafische Benutzeroberfläche. Mit der Monitoring-Funktion ist es jederzeit möglich, die Kontrolle einer laufenden Tracer-Instanz aus dem Prozesseditor zu übernehmen (vgl. Anforderungen T5: Ausführbarkeit).

Codegenerator
Der Codegenerator übersetzt ein jABC-Modell in Sourcecode. Durch Anpassungen des Codegenerators kann Source für unterschiedliche Programmiersprachen erzeugt werden (z.B.: Java oder C++). Nachdem der Source übersetzt wurde, ist es danach möglich, den Prozess als eigenständiges Programm auszuführen, ohne das Modell wie beim Tracer zu interpretieren (vgl. Anforderungen T6: Codegenerierung).

Localchecker
Das Prüfen von lokalen Eigenschaften eines SIBs wird durch das Localchecker-Plugin vorgenommen. Die Prüfroutine ist dabei als Teil der Implementierung des SIBs ausgeführt. Der Localchecker ruft sequenziell die Prüfroutinen aller SIBs auf. Falls Fehler oder Warnungen auftreten, werden diese direkt im Modell und als Übersicht im Prozesseditor angezeigt (vgl. Anforderungen G7: Korrektheit).

Modelchecker
Die Aufgabe des Modelchecker-Plugins ist die Verifikation von globalen Eigenschaften eines jABC-Modells. Dazu kann der Anwender verschiedene temporallogische Formel hinterlegen, die der Modelchecker dann anhand des vorliegenden Modells auswertet. Wie beim Localchecker erfolgt eine visuelle Rückmeldung, falls das Modell die Formel nicht erfüllt (vgl. Anforderungen G7: Korrektheit).

3.2 Applikationsebene

Das jABC-Gesamtsystem setzt sich aus drei Anwendungen zusammen. Dies sind der *jABC-Prozesseditor*, eine *Entwicklungsumgebung* für Java-Klassen (IDE) und ein *Versionierungssystem* (RCS).

Diese Top-Level-Schicht wird als *Applikationsebene* bezeichnet. Ein *Anwendungsexperte* entwirft mit dem jABC-Prozesseditor aus vorgefertigten Komponenten seine Prozessmodelle. Dazu verwendet er eine intuitive, grafische Semantik für Kontrollflussgraphen (vgl. Anforderung G1: Intuitivität). Die Komponenten, die auch als SIBs[3] bezeichnet werden, stellt ein *SIB-Experte* mit Hilfe eines kompatiblen Java-Compilers bereit. Dazu eignen sich besonders Entwicklungsumgebungen wie Netbeans [SUN07b] oder Eclipse [Ecl07]. Diese IDEs verfügen wie das jABC über Pluginschnittstellen zur Erweiterung des Funktionsumfanges. Mit Plugins für diese Entwicklungsumgebungen ist eine erweiterte Integration zwischen IDE und dem Prozesseditor möglich. Für die Eclipse-IDE wurden solche Plugins im Rahmen einer Diplomarbeit erstellt und getestet [Ded05].

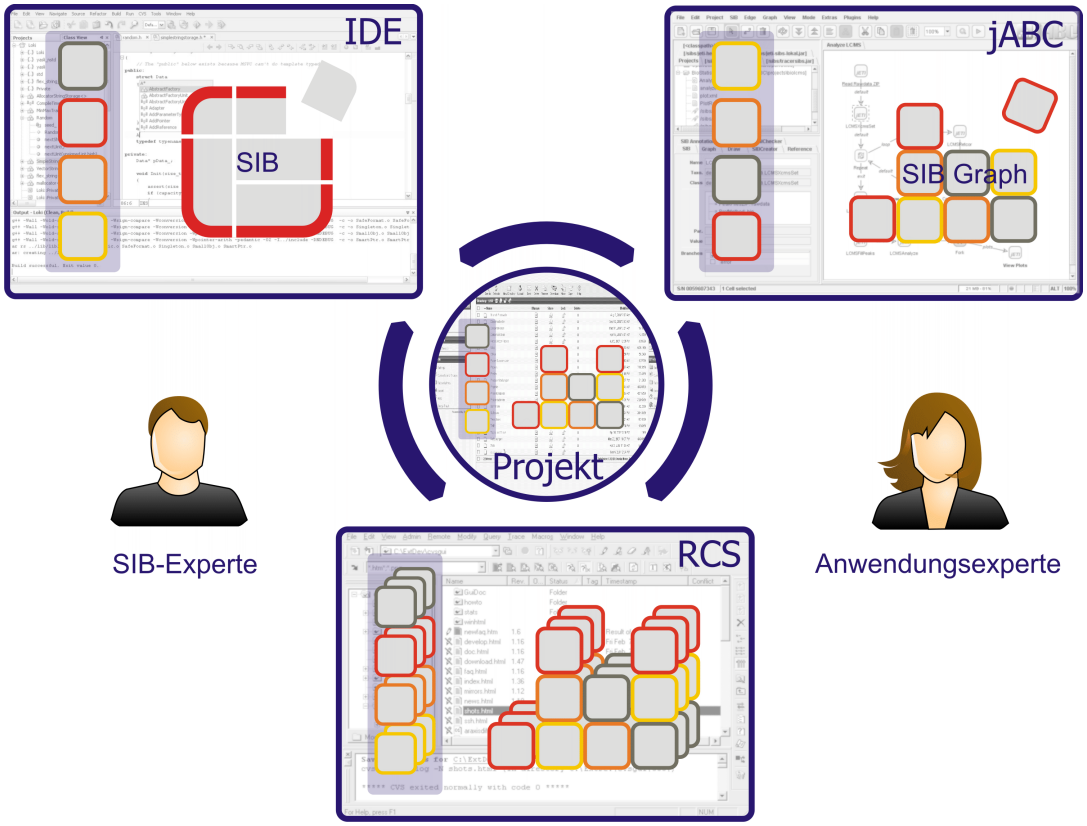

Abbildung 3.1: Applikationsebene

Der Anwendungsexperte entspricht der in der Einführung genannten Zielgruppe, dem Business Developer in einem Industrieunternehmen. Mit dem SIB-Experten

[3] siehe Kapitel 3.8 Service Independent Building Blocks, Seite 42

steht dem Anwendungsexperten ein auf die Umsetzung von SIB-Komponenten spezialisierter Programmierer zur Seite.

Technische Grundlage der Zusammenarbeit der beiden Anwenderrollen und deren eingesetzten Anwendungen ist ein *Projektordner* im Dateisystem des jeweils eingesetzten Computers. Auf der Applikationsebene verwenden die drei Anwendungen den selben Projektordner. Durch ein Versionierungssystem (RCS) können die Projektordner auf unterschiedlichen Computern synchronisiert werden (vgl. Anforderung T7: Versionierbarkeit). Weit verbreitete kostenlose RCS-Systeme sind CVS[4] oder Subversion[5]. Der Projektordner wird bei diesen Systemen von einem Server überwacht und die Änderungen an den überwachten Dateien in einem *Repository* protokolliert. Mit Hilfe des Versionierungssystems ist es möglich, konsistente Snapshots eines Projektes zu beliebigen Zeitpunkten wiederherzustellen und die Verteilung der Änderungen zwischen verschiedenen Computern in einem Netzwerk zu organisieren. Einfache Rollen- und Rechte-Konzepte von RCS Systemen liefern die Basis für die Verwaltung von mehreren Anwendern mit unterschiedlichen Aufgabenbereichen und Privilegien. Zusätzlich bieten RCS-Systeme die Möglichkeit, Dateien für den exklusiven Zugriff zu sperren. Damit lassen sich Probleme bei der Synchronisierung von Dateien nach parallelen Änderungen an verschiedenen Computern vermeiden. Zu den genannten RCS-Systemen existieren eine Vielzahl von Clientprogrammen für unterschiedliche Betriebssysteme. Für Eclipse und Netbeans sind auch integrierte Clients in Form von Plugins erhältlich.

Durch diese simple Art der Zusammenarbeit können beliebige IDE- oder RCS-Kombinationen mit dem jABC-Prozesseditor verwendet werden. Häufig ist eine Infrastruktur zur Softwareentwicklung bestehend aus IDE und RCS in entsprechenden Abteilungen von Unternehmen vorhanden. Somit erweitert der jABC-Prozesseditor nur eine Quasi-Standardlösung zur Softwareentwicklung.

Alle hier genannten IDE- und RCS-Softwaresysteme sind für die meist genutzten Betriebssysteme[6] verfügbar. Insbesondere kann der jABC-Prozesseditor als Java-Anwendung mit nahezu jedem Betriebssystem und -architektur verwendet werden, für das eine JVM existiert (Anforderung G4: Plattformunabhängigkeit).

[4] Concurrent Versions System [GNU06]
[5] Subversion Version Control System [Col07]
[6] Windows, Linux, Mac OS X, Solaris, AIX, HP-UX usw.

3.3 Framework und Bibliotheken

Monolithische Softwarearchitekturen sind für die Umsetzung eines anpassbaren, auf Benutzer und Anwendungsfall zugeschnittenen Softwaresystems ungeeignet. Diese Anforderung wird dagegen von einer *Framework/Plugin*-Architektur erfüllt (vgl. Anforderung T1: Framework-Architektur). Es handelt sich dabei um eine flexible Softwarearchitektur, bei der Erweiterungen und Anpassungen zur Laufzeit vorgesehen sind. Zusätzliche Softwarebibliotheken können neue Funktionen zum Framework hinzufügen. Solche Erweiterungsbibliotheken nennt man *Plugins*. In der Regel ist ein Plugin nicht selbstständig ausführbar, sondern kann nur zusammen mit dem zugehörigen Framework verwendet werden. Da das Vorhandensein der Plugins nicht vorausgesetzt werden kann, sind Framework und Plugins niemals statisch gelinkt. Pluginkonzepte gibt es für unterschiedliche Sprachen und Compiler. Mit der Verfügbarkeit der Reflection-API sind Framework-Pluginarchitekturen in Java besonders einfach zu realisieren. Bei einem Frameworkansatz können mit geringem Aufwand Anpassungen und Erweiterungen durch die Auswahl, den Austausch oder durch zusätzliche Plugins vorgenommen werden.

Mit der breiten Verfügbarkeit von (Java-) Bibliotheken hat sich auch die Art und Weise der Softwareentwicklung weiterentwickelt. Funktionen, die vor Jahren noch selbst entwickelt werden mussten, stehen heute als fertige Bibliotheksprojekte bereit. Für die Vorgängerversion des jABCs wurden beispielweise eine C++-Graphbibliothek inklusive Persistenzfunktion (PLGraph) sowie ein Scriptinterpreter (HLL) entwickelt. Das jABC verwendet heute für die gleichen Funktionen die Graphbiliothek jGraph, XML als Persistenzmechanismus und Java als Ersatz für die Skriptsprache. Bei der Auswahl von Bibliotheken muss der Softwarearchitekt sorgfältig zwischen verschiedenen Möglichkeiten abwägen. Eigenschaften, die die Auswahl von Bibliotheken beeinflussen sind:

- Lizenzbedingungen
- Qualität und Effizienz der Implementierung
- Wartung der Bibliothek
- Verfügbarkeit des Sourcecodes

Zusammenfassend profitiert ein Softwarearchitekt durch den konsequenten Einsatz von Bibliotheken. Der eigene Implementierungsaufwand sinkt und die Entwicklung kann sich auf die eigentlichen Kernaufgaben fokussieren. Dabei hängt der Erfolg dieses Ansatzes entscheidend von der Auswahl der geeigneten Bibliothek ab. Durch falsche Entscheidungen (z.B. für eine stark fehlerhafte Bibliothek) kann sich der Aufwand auch erheblich erhöhen. Letztendlich kommt es

bei der Auswahl der eingesetzten Bibliotheken auf die Sorgfalt und Erfahrung des Softwarearchitekten an.

Als Konsequenz wurde das jABC als Framework/Plugin-Architektur (vgl. Anforderung T1: Framework-Architektur) unter konsequenter Nutzung von geeigneten Bibliotheken realisiert.

3.4 Architektur

Der jABC-Prozesseditor wurde, wie in Abschnitt 3.3 motiviert, als Framework/Plugin-Architektur in der Programmiersprache Java realisiert. Durch die Verwendung von Java-Technologien kann das jABC auf nahezu allen aktuellen Betriebssystemen und Hardware-Plattformen verwendet werden (vgl. Anforderung G4: Plattformunabhängigkeit). Einzige Voraussetzung ist eine kompatible Java-VM[7]. Zur Installation oder Laufzeit des jABCs werden vom Anwender keine besonderen Privilegien[8] benötigt. Mit diesen minimalen Anforderungen kann der jABC-Prozesseditor von jedem Computeranwender unabhängig vom jeweiligen Betriebssystem verwendet werden.

Der jABC-Prozesseditor setzt sich aus mehreren *Kernbibliotheken* zusammen, die auch als *jABC-Kernprojekte* bezeichnet werden. Diese Kernprojekte werden mit automatisierten Maven-Scripten [Apa07b] separat erzeugt. Diese einzelnen Kernprojekte verwenden eine Reihe von zusätzlichen Hilfsbibliotheken, deren Abhängigkeiten über Maven verwaltet wird. Zu den Kernprojekten zählen:

jABC-Framework
: Diese Bibliothek ist das Rückgrad des Systems. Sie enthält die grundlegenden Klassen und Methoden der jABC-Modelle zusammen mit anderen häufig benötigten Bestandteilen. Die Framework-Methoden werden immer dann benötigt, wenn jABC-Modelle in prozessgesteuerten Projekten eingesetzt werden. Ein Beispiel dafür ist der ContentEditor. Natürlich setzt der jABC-Prozesseditor selbst auch auf den Funktionen des Frameworks auf (siehe Abschnitt 3.5, Seite 26).

ContentEditor
: Der ContentEditor ist ein grafischer Editor für strukturierte Informationen. Die Struktur dieser Informationen wird durch eine Grammatik definiert, die als jABC-Modell hinterlegt ist. Solche Grammatiken bestehen

[7] aktuell benötigt das jABC mindestens eine JRE 1.5
[8] Administrator oder Rootrechte

aus spezialisierten Grammatik-SIBs. Mit auf den jeweiligen Anwendungsfall zugeschnittenen Grammatiken können spezialisierte Editoren erzeugt werden. Im jABC-Prozesseditor wird der ContentEditor als Dokumentationswerkzeug eingesetzt. Mit entsprechenden Grammatiken wird er dann als *AnnotationEditor* bezeichnet (siehe Abschnitt 4, Seite 55).

jABC-Editor
Die Editorbibliothek enthält die grafische Benutzeroberfläche zusammen mit den dazugehörigen Methoden des Prozesseditors. Auch die grafische Oberfläche des Prozesseditors ist wie ein Baukasten aufgebaut. In der Editorbibliothek sind eine Vielzahl unterschiedlicher GUI-Komponenten enthalten. Beim Start wird durch die aktuelle Konfiguration entschiedenen, welche Teile des Baukastens zum Einsatz kommen und wie diese kombiniert werden sollen. Insbesondere kann das Menüsystem angepasst werden. Durch sogenannte Sprachbundles ist es möglich, die Benutzeroberfläche in andere Sprachen zu übersetzen (siehe Abschnitt 3.6, Seite 28).

verschiedene Plugins
Als Framework/Plugin-Architektur sind Plugins ein wesentlicher Bestandteil des Systems. Plugins fügen neue Funktionen, Menüeinträge, Inspektoren und andere GUI-Komponenten zum System hinzu. Während der Prozesseditor ohne Plugins nur die grundlegenden Funktionen zum Modellieren, Laden und Speichern bereitstellt, wird den Modellen durch die Plugins erst eine Bedeutung gegeben. Erst mit einer genau definierten Semantik, die von einem Plugin implementiert wird, kann ein Modell dann ausgeführt werden (siehe Abschnitt 3.7, Seite 35).

verschiedene SIB-Paletten
SIB-Paletten strukturieren die SIB-Komponenten zu logischen Einheiten. Nur durch eine reiche Palette von Komponenten kann das jABC-System vielseitig genutzt werden. Das simple Komponentenmodell erleichtert die Erweiterung von SIB-Paletten. Ein Java-Programmierer kann ohne lange Einarbeitungszeit schnell neue SIBs bauen. Dazu werden keine Spezialwerkzeuge benötigt, ein standardkonformer Java-Compiler reicht aus. Durch die Plugin-Interfaces können SIBs mit mehreren Plugins zusammenarbeiten. Die Common-SIB-Palette unterstützt z.B. gleichzeitig Tracer, Localchecker und den Codegenerator Genesys (siehe Abschnitt 3.8, Seite 42).

Für die Verteilung des jABCs an Projektpartner kann ein Java-Installer erzeugt werden. Der Installer basiert auf dem IzPack-Projekt [Jul09]. Auch der Installer wird über Mavenskripte automatisiert zusammengesetzt. Je nach Anforderung können verschiedene Kombinationen, bestehend aus Prozesseditor, Plugins, SIB-Paletten und Demoprojekten, zu einem Installer zusammengepackt werden.

Beim Mavenaufruf kann per Parameter ausgewählt werden, welche Zusammenstellung verwendet werden soll.

Abbildung 3.2: Auswahlbildschirm im jABC-Installer

3.5 jABC-Framework

Zentrales Element des jABC-Systems ist die Frameworkbibliothek. Diese Bibliothek stellt grundlegende Funktionen als API zur Verfügung. Das Framework wird in allen jABC-basierten Projekten eingesetzt, die direkt auf einem Prozessmodell arbeiten. Dazu gehört neben dem ContentEditor natürlich auch der eigentliche Prozesseditor. Alternativ kann ein jABC-Modell vom Codegenerator auch in frameworkunabhängigen Sourcecode übersetzt werden, wenn die SIBs das entsprechende Plugin-Interface implementieren. Durch diese Unabhängigkeit fehlen im resultierenden Sourcecode allerdings viele Annehmlichkeiten des Frameworks. Für welche Art der Umsetzung sich ein Software-Architekt entscheidet, hängt von den Anforderungen des Zielsystems ab.

Das Framework basiert selbst auf einer Reihe von weiteren Bibliotheken. Von zentraler Bedeutung für das jABC ist dabei die Open-Source-Graphbibliothek *jGraph* [JGr07]. Diese liefert Funktionen zur Repräsentation und Visualisierung von beliebigen Graphenmodellen. Typische Java-Klassen von jGraph sind Kanten (`DefaultEdge`), Knoten (`DefaultCell`), Modelle (`DefaultGraphModel`) und die dazu passende Swing-Visualisierungskomponente (`JGraph`). Die Frameworkbibliothek enthält auf diesen Klassen basierende Spezialisierungen für das jABC.

So sind beispielsweise Kanten eines jABC-Modells im Unterschied zum jGraph-Modell gerichtet und haben eine oder mehrere Beschriftungen. Daher gibt es zu den jGraph-Klassen entsprechende jABC-Pendants. Diese sind: eine Kante (`SIBGraphEdge`), ein Knoten (`SIBGraphCell`), das Modell (`SIBGraphModel` und die Swing-Komponente (`SIBGraph`). Neben den erwähnten Klassen enthält das Framework eine Vielzahl weiterer Funktionen, die den folgenden Kategorien zugeordnet werden können:

SIB-Container
: Dieser Container beinhaltet Funktionen, die für den Umgang mit SIB-Komponenten benötigt werden. Als Container werden Java-Klassen bezeichnet, die als Laufzeitumgebung für andere Objekte dienen. Ein bekanntes Beispiel ist der Web-Container Tomcat, der eine Laufzeitumgebung für Java Servlets und JSP-Seiten bereitstellt. Zu den Aufgaben des SIB-Container gehört das Scannen, Laden, Instanzieren und die Verwaltung der gefundenen SIB-Komponenten. Der `SIBClassLoader` übernimmt analog zu einem Java-`ClassLoader` die Aufgabe des Ladens von SIB-Java-Klassen. Eine besondere Eigenschaft des `SIBClassLoader` ist es, dass SIB-Komponenten ohne Neustart der JVM erneut geladen werden können.

Model-Container
: Dieser Container enthält die bereits genannten Klassen zur Repräsentation von jABC-Prozessmodellen im Speicher. Mit der Klasse `SIBGraph` kann ein Modell in einer Swing-Oberfläche angezeigt werden. Diese Klasse enthält eine Menge von Änderungen und Optimierungen gegenüber der ursprünglichen jGraph-Implementierung. Für das Laden und Speichern der Modelle verwendet der Container die Funktionen der Persistenz-API. Der Model-Container übernimmt auch die Verwaltung der hierachischen Beziehung zwischen verschiedenen Modellen. Dazu wird ein besonderes SIB, das *GraphSIB*, verwendet. Ein GraphSIB repräsentiert eine Referenz auf ein anderes Prozessmodell. GraphSIBs werden wie alle anderen SIBs einfach per *Drag and Drop* auf der Zeichenfläche platziert. Danach muss das GraphSIB konfiguriert werden, indem in einem zweiten Schritt das referenzierte Modell ausgewählt wird. Zusätzlich gibt es im jABC die Möglichkeit, eigene GraphSIBs zu erzeugen, die fest auf ein anderes Prozessmodell vorkonfiguriert sind.

Plugin-Container
: Die Erweiterung des jABCs durch Plugins wird durch den Plugin-Container ermöglicht. Beim Start versucht der Container, die verschiedenen Plugins zu laden. Plugins im jABC sind Java-Klassen, die ein einfaches Java-Interface implementieren. Das Interface enthält Methoden, die in der Startphase (`start()`) und vor dem Beenden des Systems (`stop()`) aufge-

rufen werden. Beim Start meldet ein Plugin seine Funktionen am Framework an, indem z.B. eigene Menüeinträge hinzugefügt werden (siehe Abschnitt 3.7, Seite 35).

Persistenz-API
Diese API stellt einen erweiterbaren Mechanismus zur persistenten Speicherung von Modellen als XML-Dateien bereit. Besonderen Wert wurde bei der Implementierung auf fehlertolerante Mechanismen beim Laden von jABC-Modellen gelegt, sodass auch fehlende oder inkompatible XML-Fragmente abgefangen werden (siehe Kapitel 3.9 auf Seite 52).

Utility-API
Wie in jedem größeren Softwareprojekt stellt das jABC-Framework eine ganze Reihe von Hilfsklassen zur Verfügung, die den Pluginentwicklern die Arbeit erleichtern oder wiederkehrende Aufgabe erledigen.

Tracer-Plugin-API
Zur Vereinfachung werden die grundlegenen Methoden des Tracer-Plugins als Bestandteil des Frameworks mitgeliefert. Trotzdem handelt es sich dabei nicht um Frameworkfunktionen im eigentlichen Sinne, sondern um ein eigenständiges Plugin.

Framework-SIBs
Das Framework beinhaltet zusätzlich noch einige besondere SIBs wie das *ProxySIB* oder *GraphSIB*. Das ProxySIB wird von der Persistenz-API als Platzhalter für fehlende oder inkompatible SIBs verwendet. Der Model-Container interpretiert das GraphSIB als eine Referenz auf ein anderes Prozessmodell.

3.6 jABC-Prozesseditor

Der Prozesseditor ist eine Java-Anwendung, die mit einer Swing-Benutzeroberfläche ausgestattet ist. Swing ist eine leichtgewichtige Bibliothek zur Erzeugung von Benutzeroberflächen in Java, deren Erscheinungsbild (*Look and Feel*) austauschbar ist. Die Darstellung der Benutzeroberfläche erscheint auf unterschiedlichen Betriebssystemen nahezu identisch, da zum Rendern keine Funktionen des nativen Betriebssystems eingesetzt werden. Der Prozesseditor wird beim Start entsprechend einer Konfigurationsdatei dynamisch aus den verfügbaren Swing-Komponenten zusammengesetzt. Die *Konfigurations-API* hat auf fast alle Bestandteile des Editors Einfluss.

Die Benutzeroberfläche des Prozesseditors verwendet die folgenden Basiselemente:

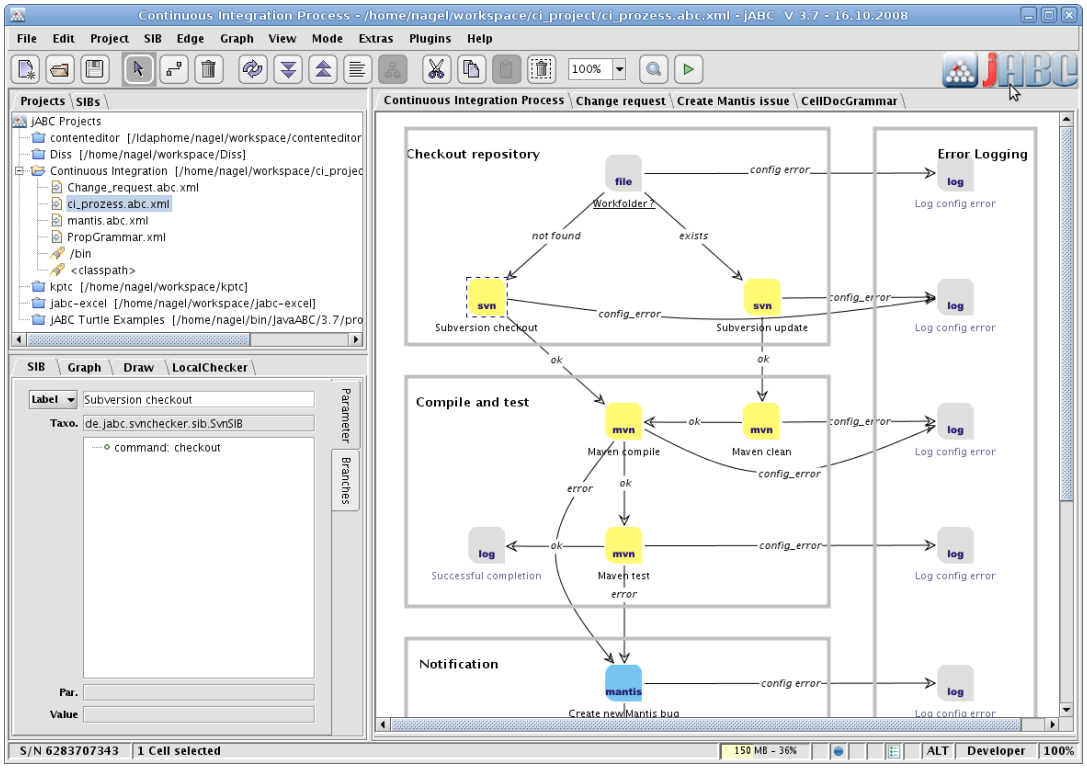

Abbildung 3.3: jABC Prozesseditor

Menüsystem/Toolbar

Das Menüsystem enthält verschiedene Aktionen, die ein Benutzer auslösen kann. Für den schnelleren Zugriff sind die wichtigsten Aktionen auch über die Toolbar erreichbar. Andere Teile der Benutzeroberfläche enthalten spezialisierte Popupmenüs. Viele Aktionen sind in den verschiedenen Menüs mehrfach vorhanden. Die Zusammensetzung des Menüsystems und der Toolbar wird in der Editorkonfiguration festgelegt. Zusätzliche Aktionen können durch Plugins hinzugefügt werden. Üblicherweise befinden sich die Menüeinträge aller Plugins im Hauptmenü `Plugins`. Technisch können die Plugins aber an jeder Stelle des Menüsystems eigene Erweiterungen vornehmen.

Projekt-/Taxonomiebrowser

Der Projektbrowser zeigt alle angelegten Projekte in einer Baumstruktur an. Nur das jeweils *aktive Projekt* ist als Detailansicht geöffnet. Diese Detailansicht enthält alle Prozessmodell-Dateien, die einfach per Doppelklick geöffnet werden können. Bestandteil einer Projektdefinition ist auch die Liste der verwendeten SIB-Bibliotheken. Ein Eintrag einer SIB-Bibliothek

wird als *SIB-Pfad* bezeichnet. Der Benutzer kann die Liste der SIB-Pfade im Projektbrowser bearbeiten. Der Taxonomiebrowser zeigt in einer Baumstruktur alle SIBs an, die in den SIB-Pfaden des aktiven Projekts gefunden wurden. Die Baumstruktur des SIBs kann vom Anwender auf seine Bedürfnisse angepasst werden. Das TaxonomieEditor-Plugin stellt dafür einen entsprechenden Editor bereit (vgl. Anforderung E6: Semantische Komponentenverwaltung).

Inspektorpanel

Dieses Panel zeigt alle Inspektoren des Prozesseditors an. Inspektoren sind spezialisierte Formulare der Benutzeroberfläche, mit denen Eigenschaften von selektierten SIBs oder Graphen bearbeitet werden können. Zusätzlich werden Inspektoren auch unabhängig vom aktuellen Modell als Ersatz für Dialoge eingesetzt. Für eine bessere Übersichtlichkeit der Benutzeroberfläche kann der Benutzer häufig verwendete Inspektoren in eigenständige Fenster auslagern. Inspektoren, die nicht benötigt werden, können ausgeblendet werden.

Der Prozesseditor enthält in der Basiskonfiguration ohne Plugins drei Inspektoren:

SIB-Inspektor

Der SIB-Inspektor dient zur Manipulation der grundlegenden Eigenschaften eines SIBs. Dazu zählen der *Name*, das *Label*, die *Parameter* und die *Branches* eines SIBs. Einzelne Parameter (und Branches) eines SIBs können zu Modellparametern erklärt werden. Der Wert eines Modellparameters wird nicht über das SIB sondern am Modell gesetzt. Wird das Modell über ein GraphSIB referenziert, werden die Modellparameter zu den SIB-Parametern des GraphSIBs.

Graph-Inspektor

Analog zum SIB-Inspektor können mit dem Graph-Inspektor die Eigenschaften eines Modells wie *Name*, *Parameter* und *Branches* bearbeitet werden. Modellparameter können mit einem Namen belegt werden, der unabhängig vom Namen des SIB-Parameters ist (*Parameter-Mapping*). Verwenden mehrere (kompatible) Modellparameter den gleichen Namen, können über einen Modellparameter die Parameterwerte von mehreren SIBs gleichzeitig gesetzt werden. So wird es möglich, einheitliche Werte von Parametern für verschiedenen SIBs zu setzen.

Draw-Inspektor

Über diesen Inspektor können rudimentäre Zeichenobjekte wie Linien, Rechtecke, Ovale, Texte und Bilder in Modelle eingefügt werden. Diese Objekte dienen nur zur visuellen Hervorhebung und

Dokumentation eines Modells. Sie haben keinerlei Auswirkungen auf die Semantik des Modells.

Durch Plugins werden weitere Inspektoren zum Inspektorpanel hinzugefügt. Die genaue Zusammensetzung des Inspektorpanels hängt von den geladenen Plugins ab.

Canvas/Zeichenfläche

Die Zeichenfläche des Prozesseditors steht im Zentrum der Arbeit eines Benutzers. Durch intuitive Gesten stellt der Anwender mit der Maus seine Modelle per "Drag and Drop" zusammen. SIB-Komponenten werden zuerst im Taxonomiebrowser ausgewählt und dann auf dem Canvas platziert. Zwischen SIBs werden Kanten mit der Maus *gezogen*. Nach einem Doppelklick auf eine Kante öffnet sich der *Brancheditor*, mit dem aus der Liste der verfügbaren Branches die Kantenbeschriftung vorgenommen wird. Der Benutzer kann beliebige SIBs, Kanten oder Zeichenelemente mit der Maus selektieren (*Einzelselektion*, *Multiselektion*, *Lasso-Funktion*). Über die Zwischenablage können selektierte Elemente kopiert oder in andere Modelle verschoben werden. Zu fast allen Editierfunktionen gibt es *Undo* und *Redo* Methoden. Über GraphSIBs werden die hierachischen Beziehungen zwischen Modellen hergestellt. Mit Makrofunktionen wie *Create Subgraph* und *Expand GraphSIB* sind sogar komplexe Operationen auf Modellen möglich. Das *Create Subgraph* Macro lagert den selektierten Teilbereich eines Modells in ein Untermodell aus und erzeugt automatisch ein GraphSIB. *Expand GraphSIB* bietet dazu eine inverse Funktion an.

Statuszeile

Die Statuszeile liefert dem Benutzer zusätzliche Informationen über den aktuellen Betriebszustand des Prozesseditors. Hier kann zwischen verschiedenen Profilen gewechselt werden. Ein Profil ist eine alternative Konfiguration des Editors. Mit verschiedenen Profilen kann der jABC Editor an wechselnde Arbeitssituationen angepasst werden.

Der Prozesseditor verwendet die Funktionen der Framework-Bibliothek und den ContentEditor. Eine Übersicht der verwendeten Bibliotheken zeigt die Abbildung 3.4 auf der nächsten Seite. Viele API-Methoden im Framework können durch Menüeinträge und entsprechende Eingabeformulare aus der Benutzeroberfläche ausgelöst werden. Die Funktionen der Editor-Bibliothek können den folgenden Kategorien zugeordnet werden:

Projekt-Container

Dieser Container verwaltet die Liste der Projektdefinitionen im Prozesseditor. Der Projektbrowser ist die grafische Darstellung des Projekt-Containers. Eine Projektdefinition beinhaltet die Projektwurzel

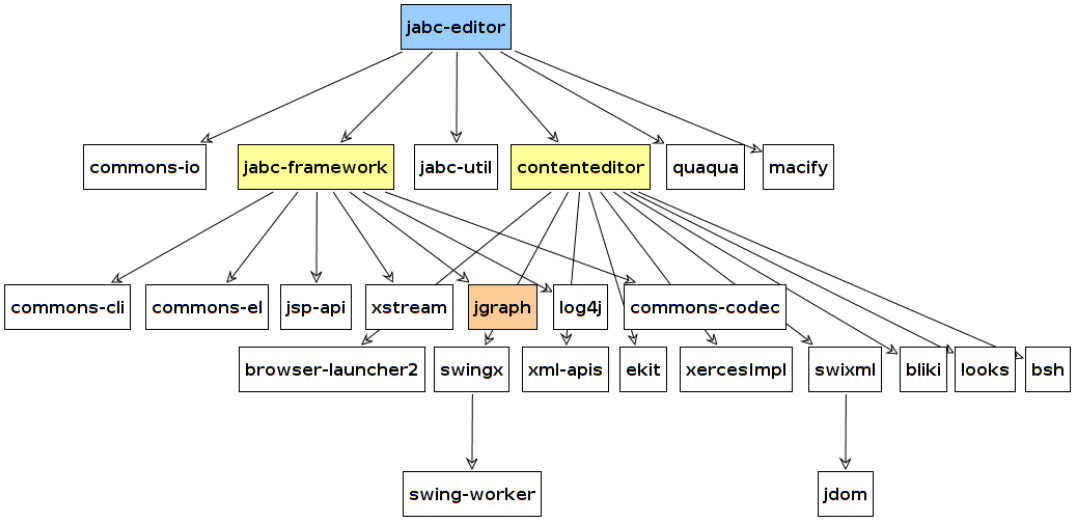

Abbildung 3.4: Maven Dependencygraph der Kernbibliotheken

im Dateisystem, die SIB-Pfade, einen anpassbaren Java-Klassenpfad sowie die Projekt-Annotation im AnnotationEditor (vgl. Abschnitt 4.3, Seite 62). Von allen Projektdefinitionen kann ein Projekt als das *aktive Projekt* selektiert werden. Für das aktive Projekt baut der Container eine Baumansicht über die enthaltenen Modelle auf. Weiterhin wird, entsprechend den SIB-Pfaden, mit Hilfe des SIB-Containers der Taxonomiebrowser mit der Liste der verfügbaren SIBs aktualisiert.

Inspektor-Container

Dieser Container verwaltet alle geladenen Inspektoren. Neben den Standard-Inspektoren werden von verschiedenen Plugins weitere Inspektoren hinzugefügt. Der Container informiert alle Inspektoren über die Selektionen, die der Benutzer im Canvas vornimmt. Die Inspektoren des Containers werden im Inspektorpanel der Benutzeroberfläche angezeigt. Zusätzlich speichert der Container den Anzeigestatus der einzelnen Inspektoren. Es gibt drei verschiedene Status: *im Panel*, *im eigenen Fenster* und *versteckt*. Beim Start des Prozesseditors wird der Status der Inspektoren wiederhergestellt.

Optionsdialog-Container

Eine komplexe Applikation wie der Prozesseditor besitzt eine Reihe von Konfigurationsoptionen. Dieser Container stellt mit dem erweiterbaren Optionsdialog eine modulare Möglichkeit bereit, damit Prozesseditor und Plugins einen gemeinsamen Dialog für das Setzen von Optionen anbieten können. Plugins haben die Möglichkeit, analog zum Inspektor-Container,

eigene Formulare anzumelden. Die eingestellten Werte werden zentral von der Konfigurations-API in den sogenannten *User-Properties* abgespeichert.

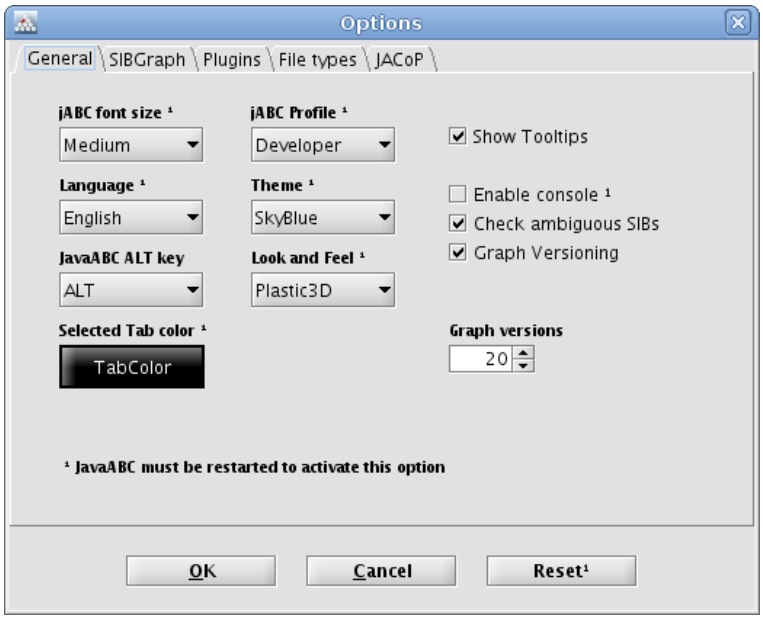

Abbildung 3.5: Optionsdialog

Menüaktionen

Zu jedem Menü- oder Toolbareintrag gehört eine Java-Aktion. Die Aktion steuert die Ausführung des ausgewählten Kommandos. Neben den eigentlichen API-Aufrufen können aus einer Aktion auch Dialoge aufgerufen werden. Die `Datei speichern unter...` Aktion zeigt beispielsweise erst den Dateiauswahldialog an, bevor über die Persistenz-API das aktuelle Modell als XML-Dokument gespeichert wird.

Konfigurations-API: Wie das Framework stellt auch der Prozesseditor eine reiche Palette verschiedener Subsysteme zur Verfügung. Die genaue Zusammensetzung des Editors wird nicht zur Compile- sondern erst zur Laufzeit bestimmt. Die Konfiguration des Editors besteht aus mehreren hierarchischen Propertydateien[9]. Welche Propertydateien ausgewertet werden bestimmt das aktuelle *Profil* des Prozesseditors. Alle Werte, die von den Vorgaben abweichen, werden in der *User-Property* abgelegt. Diese Datei liegt im Home-Verzeichnis des Anwenders. Die Werte der User-Property ersetzen die Standardwerte des Profils. Durch Profile können folgende Eigenschaften des Prozesseditors auf die Bedürfnisse des Anwenders angepasst werden:

[9]Eine Propertydatei ist eine Sammlung von Key/Value-Paaren

Menüsystem
Die Struktur und Zusammensetzung des Menü- und Toolbarsystems wird festgelegt. Es können neben den mitgelieferten Menüaktionen auch externe Aktionen eingebunden werden.

Editorsprache
Die vom Editor verwendete Sprache kann geändert werden. Die sprachabhängigen Textbausteine werden aus erweiterbaren Java-Sprachbundleklassen ermittelt. Mit zusätzlichen Klassen kann der Editor für weitere Sprachen übersetzt werden.

Zeichensatz-Eigenschaften
In der Konfiguration werden für alle GUI-Elemente verschiedene Vorgaben von Schriftgrößen und -formen festgelegt. Der Benutzer kann im Optionsdialog auswählen, welche Schriftvorgaben er verwenden möchte.

Look-and-Feel
Diese Eigenschaft legt das gesamte Aussehen des Prozesseditors fest. Es handelt sich dabei um eine Funktion der Swing-Bibliothek.

Inspektoren
Beim Start wertet der Inspektor-Container aus, welche Inspektoren initial unabhängig von Plugins geladen werden.

Plugins: Der Plugin-Container lädt die Plugins anhand der Konfiguration des aktuellen Profils. Ein Optionsdialog ermöglicht die Pflege der Pluginliste.

Projekte
Der Projekt-Container wertet bei der Initialisierung die aktuelle Konfiguration aus. Zu jedem Projekt existiert im Wurzelverzeichnis eine weitere Propertydatei, in der die Konfiguration des Projekts abgelegt wird (SIB-Pfade, Klassenpfad).

Optionsdialoge
Die Konfiguration bestimmt die initialen Dialoge im Optionsdialog, die von Plugins noch erweitert werden können.

Der Prozesseditor verfügt über einen eigenen *Property-Editor*, mit dem die aktuelle Konfiguration eingesehen oder angepasst werden kann. Der Editor zeigt an, welche Konfigurationswerte vom Prozesseditor ausgewertet wurden. Eine Filterfunktion ermöglicht ein komfortables Arbeiten und erleichtert die Suche in den Konfigurationswerten.

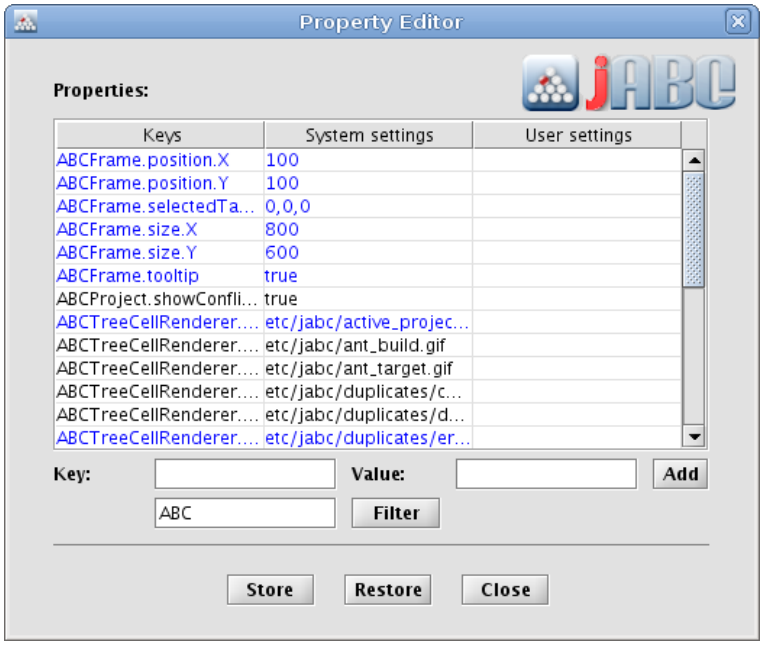

Abbildung 3.6: Property-Editor des jABCs

3.7 Plugins

Die Erweiterbarkeit des jABCs durch Plugins ist von zentraler Bedeutung (vgl. Anforderung T1: Framework-Architektur). Einige Anforderungen aus dem Kapitel 1 auf Seite 1 werden erst durch spezialisierte Plugins erreicht. Die Kommunikation zwischen einem Plugin und den SIBs erfolgt durch ein pluginspezifisches Interface, das sogenannte *Plugin-Interface*. Beim Laden eines Plugins wird das Plugin-Interface falls vorhanden beim SIB-Container registriert. Danach berücksichtigt der Container, dass ein Plugin Methoden aus diesem Interface aufrufen möchte. Damit keine Fehler dadurch entstehen, dass ein SIB diese Methoden nicht bereitstellt, werden bei entsprechenden SIBs solche Aufrufe ausgefiltert. Die Methoden werden dann durch funktionslose Implementierungen simuliert. Diese Kapselung der SIBs gegen den Zugriff durch Plugins wird durch ein Java-Proxyobjekt realisiert. Jede SIB-Instanz wird durch das `SIBProxy`-Objekt vor fehlerhaften Aufrufen geschützt und mit der Basisimplementierung für SIBs (`SIBBase`) verwoben.

jABC-Plugins werden daher in zwei Kategorien eingeteilt:

Semantik-Plugins
: verfügen über ein eigenes Plugin-Interface und implementieren eine eigene Semantik für dieses Interface.

Feature-Plugins
: dienen der allgemeinen Funktionserweiterung des jABC. Feature-Plugins sind von der jeweiligen SIB-Implementierung unabhängig und können mit allen SIB-Paletten eingesetzt werden.

Die folgenden Kapitel geben eine grobe Übersicht über einige der wichtigsten Plugins für das jABC. Die Plugins werden als eigenständige Projekte oder im Rahmen von Studien- oder Diplomarbeiten entwickelt.

3.7.1 Tracer

Das Tracer-Plugin [Doe06] implementiert eine ausführbare Semantik für Kontrollflussgraphen (vgl. Anforderung T5: Ausführbarkeit). Das Plugin-Interface des Tracers enthält eine `trace()`-Methode. Diese SIB-Methode wird vom Tracer während des Ausführens eines Modells aufgerufen. Die Ausführung kann interaktiv über einen Steuerdialog beeinflusst werden. Neben den Funktion zum *Starten*, *Pausieren* und *Stoppen* der Ausführung kann ein Modell auch im *Einzelschritt-Modus* abgearbeitet werden. Mit Haltepunkten (*Breakpoints*) kann die Ausführung an zuvor markierten Stellen unterbrochen und später fortgeführt werden. Für die Ausführung eines Prozessmodells ist kein vorheriger Übersetzungsschritt nötig (vgl. Codegenerator). Der Tracer interpretiert das aktuelle Prozessmodell und visualisiert den Fortschritt der Ausführung, indem der zurückgelegte Pfad im Modell farblich hervorgehoben wird. Die Tracersteuerung zeigt zusätzliche Details des Ausführungszustands aller Prozesse an:

- Liste der laufenden Prozessinstanzen (Thread-Liste)
- Anzeige des Kontextzustands einer Prozessinstanz
- Übersicht der zuletzt besuchten SIBs eines Prozesses
- Liste der SIBs, die auf ein Ereignis warten

Während der Ausführung von Tracer-SIBs wird vom Tracer ein Kontext für die Inter-SIB-Kommunikation bereitgestellt. Ein Kontext kann alle Arten von Informationsobjekten unter verschiedenen Schlüsselwerten speichern. SIBs können Werte im Kontext lesen, schreiben oder löschen. Mit speziellen *ControlSIBs* wird unter anderem auch die Erzeugung von neuen Kontexten und Threads während der Ausführung gesteuert. Das Tracer-Plugin enthält mit dem *MacroSIB*, dem *ThreadSIB* und dem *GraphSIB* drei verschiedene ControlSIBs für die Verzweigung zu anderen Modellen (Modellhierarchie). Parallele Prozesse können mit den ControlSIBs *ForkSIB* und *JoinSIB* verwaltet werden. Weitere ControlSIBs

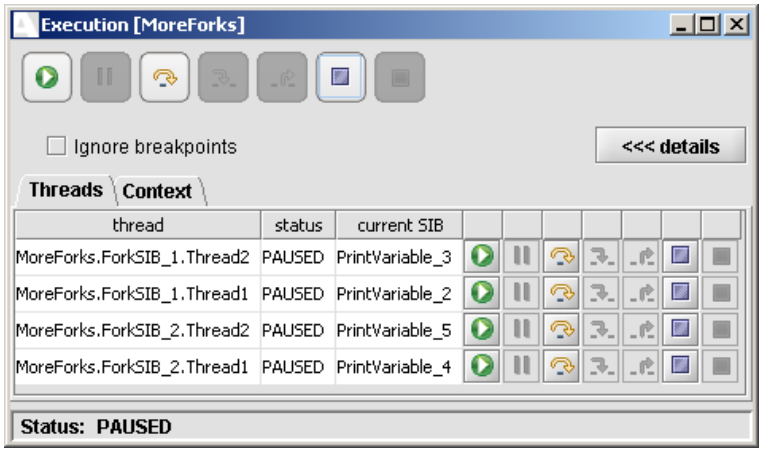

Abbildung 3.7: Tracersteuerung (aus [Doe09])

steuern das Verhalten von Tracermodellen beim Warten oder Eintreten von Ereignissen.

Über eine Kommandozeilenversion des Tracers können Modelle auch unabhängig von der Benutzeroberfläche gestartet werden. Das Tracer-Plugin bietet weiterhin eine Funktion an, sich auch nachträglich wieder mit einer laufenden Prozessinstanz zu verbinden. Damit können bereits laufende Prozesse überwacht und gesteuert werden (Monitoring).

3.7.2 LocalChecker

Die Aufgabe des LocalChecker-Plugins [Neu07] ist die Überprüfung von lokalen Anforderungen an ein SIB. In früheren Versionen wurden diese Tests ausschließlich zur Validierung von SIB-Parametern eingesetzt. Diese Beschränkung gilt für das LocalChecker-Plugin nicht mehr. Jedes SIB, das das `Localchecker`-Interface implementiert, stellt eine eigene Testmethode bereit. In dieser Testmethode kann beliebiger Code ausgeführt werden. Gängige Tests im LocalChecker prüfen folgende Eigenschaften:

- Wertebereich von Zahlparametern
- Angabe verpflichtender Werte (nicht leer)
- Konsistenz zwischen mehreren Parametern
- Existenz von ein- oder ausgehenden Kanten mit bestimmten Branchlabeln
- Existenz und Validität von SIB-Annotationen

Ist eine Bedingung im LocalChecker-Code nicht erfüllt, kann eine Rückmeldung an den Benutzer gegeben werden, die vom LocalChecker-Inspektor angezeigt wird. Die Meldungen können in aufsteigenden Prioritätskategorien klassifiziert werden. Der Benutzer kann sich durch Selektion die Fehlermeldungen von einzelnen oder allen SIBs anzeigen lassen.

Die Ausführung der LocalCheck-Prüfungen kann entweder manuell vom Anwender ausgelöst oder automatisch im Hintergrund ausgeführt werden. Bei der automatischen Prüfung wählt der LocalChecker abhängig von den letzten Änderung nur solche SIBs aus, bei denen die Testmethode erneut ausgeführt werden soll. Ein Queue-Mechanismus serialisiert alle anfallenden Testaufrufe und filtert Duplikate heraus.

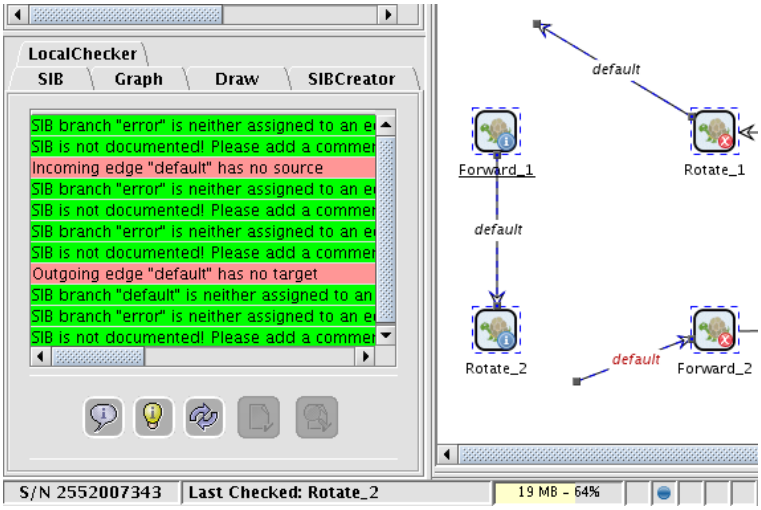

Abbildung 3.8: LocalChecker-Inspektor (aus [Neu09])

3.7.3 Modelchecker

Das Modelchecker-Plugin (GEAR) [BMRS08] dient zur Überprüfung von globalen Eigenschaften eines jABC-Modells (Anforderung G7: Korrektheit). Die zu prüfende Eigenschaft wird als temporallogische Formel[10] beschrieben [MOSS99]. Der Algorithmus des Modelcheckers prüft, ob die Formel für alle Knoten im Modell erfüllt ist. Diese Überprüfung eines Modells mit *Model Checking* wird auch als *Verifikation* bezeichnet. Temporallogische Formeln können auf sogenannte *atomare Propositionen* verweisen. Atomare Propositionen sind Eigenschaften

[10] GEAR verwendet dazu *CTL* oder *μ-Kalkül*

der Modellknoten. Im jABC können dazu alle Eigenschaften der SIB-Klassen wie Klassenname, SIB-Parameter usw. verwendet werden. Zusätzlich kann der Anwender beliebige Eigenschaften mit dem *AP-Manager* an die SIB-Knoten annotieren.

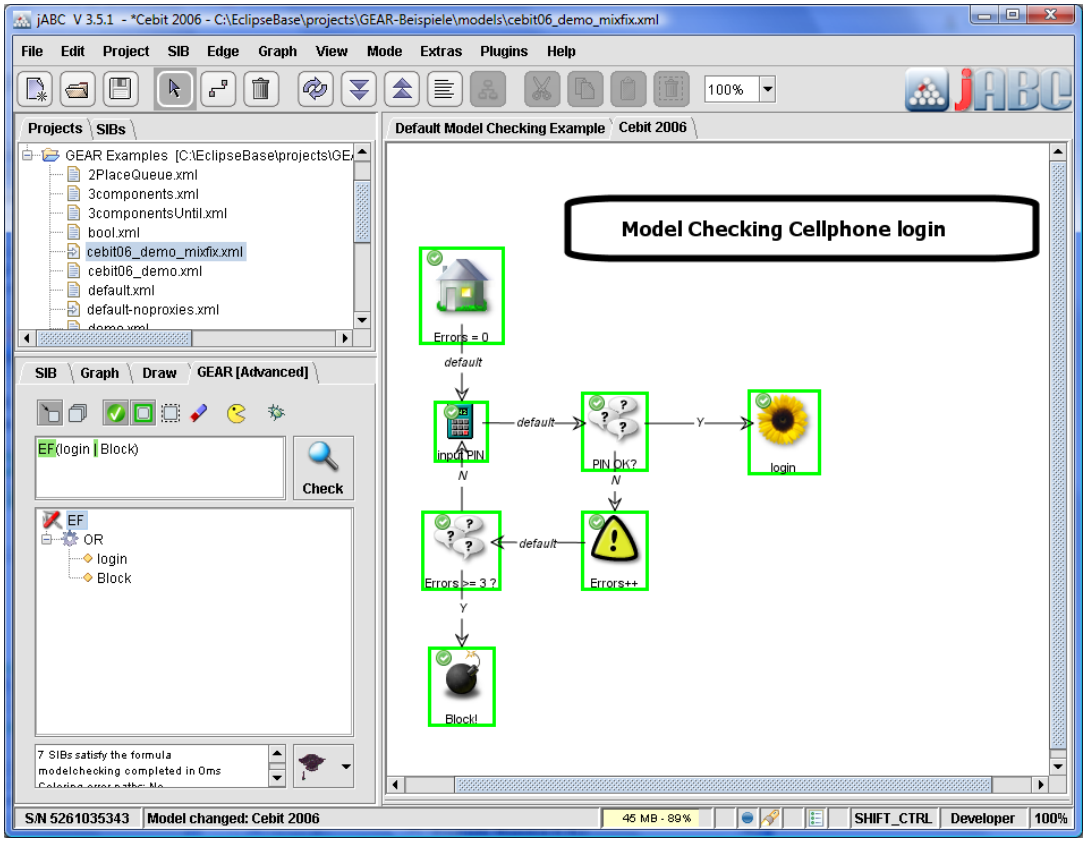

Abbildung 3.9: GEAR ModelChecker (aus [RB09])

Mit dem GEAR-Inspektor ist eine genaue Analyse der Formeln möglich. SIB-Knoten werden entsprechend markiert, ob diese die selektierte Formel oder Teilformel erfüllen oder nicht erfüllen. Mehrere Formeln werden von GEAR zusammen mit einer Beschreibung im jABC-Modell gespeichert. Eine tabellarische Übersicht informiert darüber, ob alle gespeicherten Formeln durch das Modell erfüllt wurden. Signalfarben heben solche Eigenschaften in der Übersicht hervor, bei denen die Formel nicht erfüllt ist. Der Anwender kann sich dann detailliert über die aufgetretenen Probleme informieren. Nach einer Modelländerung wird das *Model Checking* für jede Formel wiederholt.

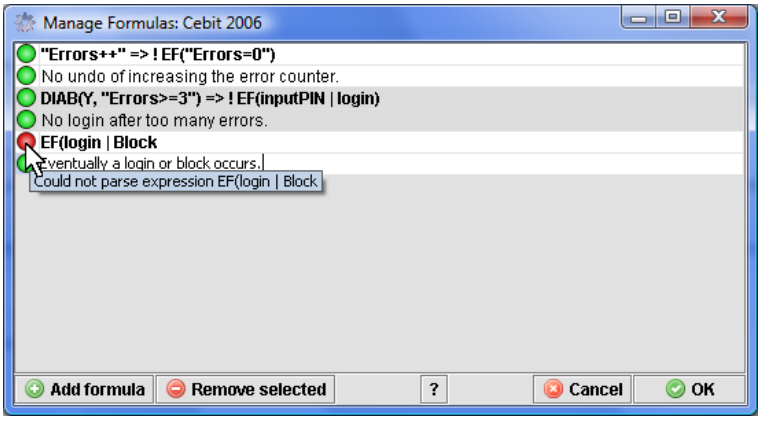

Abbildung 3.10: GEAR Formelübersicht (aus [RB09])

3.7.4 Codegenerator

Das Genesys-Projekt stellt das Codegenerator-Plugin des jABCs bereit [JMS08]. Das Projekt beinhaltet eine Reihe von verschiedenen Generatoren für unterschiedliche Zielsprachen und Aufgaben. Ein Genesys-Codegenerator verarbeitet ein oder mehrere jABC-Modelle und erzeugt daraus eine oder mehrere Source-Textdateien. Der so generierte Sourcecode kann dann von einem passenden Compiler übersetzt werden. Die Genesys-Generatoren sind selbst als jABC-Prozessmodelle implementiert. Durch einen initialen Bootstrapvorgang wurde der erste Genesys-Generator durch eine Tracer-Ausführung des entsprechenden Genesys-Modells erzeugt. Danach können neue Genesys-Generatoren mit den bereits fertiggestellten Generatoren kompiliert werden. Existierende Generatoren können sehr einfach erweitert oder für neue Zielplattformen oder -compiler überarbeitet werden (Anforderung: G2 - Agilität).

Mit Genesys wurden inzwischen eine Vielzahl von Generatoren für unterschiedliche Aufgaben entwickelt. Die folgende Liste gibt eine Übersicht einiger Generator-Varianten:

Java Class Extruder
: Dieser Generator erzeugt einen Java-Sourcecode, der zu einer Modellausführung im Tracer-Plugin identisch ist. Die Struktur des Modells wird dazu in eine entsprechende Java-Datenstruktur übersetzt. Es werden alle benötigten SIB-Komponenten instanziert und deren SIB-Parameter gesetzt. Danach führt der Traceralgorithmus die einzelnen SIBs aus und sucht dann einen entsprechenden Nachfolger.

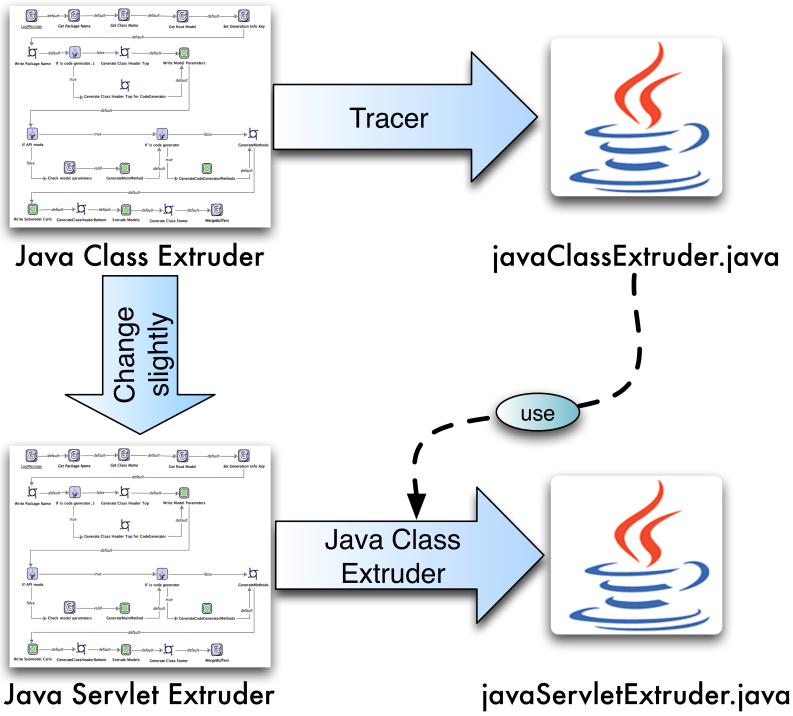

Abbildung 3.11: Entwicklung von Genesys-Generatoren (aus [JB09])

Java Class Pure Generator
Bei diesem Generator wird das jABC-Modell in eine Java-Klasse übersetzt, die keinerlei Abhängigkeiten mehr zum jABC-Framework besitzt. Hierzu müssen die SIB-Klassen ein anderes Interface als beim Tracer-Plugin implementieren.

Servlet Generator
Dieser Generator erzeugt im Unterschied zum Extruder oder Pure-Generator eine Servlet-Klasse. Damit können Web-Anwendungen mit dem jABC entwickelt werden.

C# Generator
Dieser Generator wandelt ein jABC-Modell in einen C#-Sourcecode um. Die SIBs repräsentieren dabei entsprechende C#-Methoden. Mit diesem Generator können auch andere .NET-Funktionen aufgerufen werden. Der Generator zeigt, dass die Zielplattformen des jABCs unabhängig von Java sind [Hös08].

3.8 Service Independent Building Blocks

Eine Komponente im jABC wird als *Service Independent Building Block* oder *SIB* bezeichnet. Dieser Name wurde von der Vorgängerversion, dem *Agent Building Center* (ABC) übernommen. Im ABC setzte sich die Definition einer Komponente noch aus verschiedenen Dateien und Formaten zusammen. Als Teil des Redesigns wurde das Komponentenmodell des jABCs komplett überarbeitet.

3.8.1 Komponentendefinition

Die folgenden Anforderungen aus dem Kapitel 1 auf Seite 1 wurden bei der Konzeption des neuen jABC-Komponentenmodells berücksichtigt:

- Anforderung G4: Plattformunabhängigkeit
- Anforderung G5: Universalität
- Anforderung G6: Skalierbarkeit
- Anforderung E2: Grafische Modellierung
- Anforderung E4: Einfaches Komponentenmodell
- Anforderung E5: Service-Orientierung
- Anforderung E6: Semantische Komponentenverwaltung
- Anforderung E7: Statusspezifische Implementierungen
- Anforderung T3: Service-Entkopplung
- Anforderung T5: Ausführbarkeit
- Anforderung T6: Codegenerierung
- Anforderung T7: Versionierbarkeit

Zentrale Idee des neuen Komponentenmodells ist es, eine Komponente mit genau einer Datei zu beschreiben. Die Informationen, die früher in verschiedenen Dateien vorlagen, werden nun zu einer Datei verschmolzen. Dadurch ist es nicht mehr möglich, dass eine Komponentendefinition inkonsistent oder unvollständig ist weil einzelne Dateien fehlen. Anstatt eine selbst definierte Syntax für die Komponentenbeschreibung zu verwenden, wird das neue Komponentenmodell nur noch in Java beschrieben. Es gilt die folgende Definition einer Komponente im jABC:

> *Eine SIB-Komponente ist eine Java-Bytecodeklasse, die mit der Annotation* `@SIBClass` *gekennzeichnet ist.*

Eine Annotation in Java ist die Möglichkeit, Metadaten in den Javasource einzubinden. Diese Metadaten können abhängig von der Definition der Annotation sowohl zur Compile- als auch zur Laufzeit ausgewertet werden. Als Parameter der `@SIBClass`-Annotation muss eine eindeutige *UID*[11] angegeben werden, die unveränderlich sein sollte. Anders als bei Java-Klassen, die über den Package- und Klassennamen angesprochen werden, wird eine SIB-Java-Klasse über deren UID referenziert. Eine Referenz von einem jABC-Modell zu einer UID bleibt intakt, auch wenn Klassen- oder Packagename einer SIB-Komponente verändert werden[12]. Durch diese Art der Referenzierung wird gewährleistet, dass SIB-Java-Klassen jederzeit neu organisiert oder umstrukturiert werden können, ohne dass alle Modelle, die die Komponente bereits referenzieren, angepasst werden müssen. Der Sourcecode einer SIB-Klasse wird zur Modellierungszeit **nicht** benötigt.

Bereits die Verwendung von Java und den auf Java basierenden Bibliotheken und Standards erfüllt die Anforderungen G4, G5, G6, E7, T5 und T7. Im jABC muss eine Komponente zusätzlich die folgenden Eigenschaften bereitstellen:

- Eindeutiger Komponentenname
- Klassifikation der Komponente
- Grafische Repräsentation / Piktogramm
- Parameter, Datentypen und initiale Werte
- Dokumentation aller Bestandteile der Komponente
- Verschiedene Kategorien von ausführbarem Code

Für eine SIB-Klasse werden diese Komponenten-Eigenschaften durch einfache Konventionen[13] auf Java-Sprachelemente abgebildet:

SIB	**Java-Klasse**
Name	Klassenname
Taxonomie	Package der Klasse
Parameter	Felder der Klasse
Piktogramm	Rückgabewert der Methode `getIcon()`
Dokumentation	Rückgabewert der Methode `getDocumentation()`
ausführbarer Code	durch Plugin-Interfaces definierte Methoden

[11] Unique Identifier (UID), ein eindeutiger alphanumerischer Schlüsselwert
[12] z.B. durch Refactoring
[13] in Anlehnung an das *Konvention over Konfiguration*-Paradigma des *Ruby on Rails*-Frameworks

Dieses auf Java basierende jABC-Komponentenmodell hat folgende Eigenschaften:

- SIB-Bytecodeklassen können mit Standard-Java-Compilern erzeugt werden.
- Java stellt eine breite Palette von Datentypen bereit, die als SIB-Parameter bzw. Klassenfelder verwendet werden können.
- SIB-Eigenschaften können mit der Java-Reflection-API ausgelesen werden (Name, Package, Interfaces, Annotations).
- Java-Bytecode kann auf nahezu allen Computersystemen ausgeführt werden.
- Mit Java-Bibliotheken kann der Funktionsumfang erweitert werden.
- Java bietet viele Möglichkeiten, mit anderen Technologien zusammenzuarbeiten.
- Java ist eine *General Purpose* Programmiersprache und bietet daher eine breite Funktionsvielfalt.
- Java-Sourcecode kann leicht versioniert werden.
- Java-Klassen können grafische Representationen generieren (`ImageIcon`).
- Eine Java-Klasse kann mehrere Interfaces implementieren.
- ClassLoader können Java-Klassen zur Laufzeit dynamisch laden.

3.8.2 SIB-Container

Der SIB-Container in der Framework-Bibliothek stellt komfortable Funktionen für den Umgang mit SIB-Java-Klassen zur Verfügung (vgl. Kapitel 3.5 auf Seite 26). Besonders die Verwendung von UIDs zur Referenzierung von Java-Klassen und das Neuladen von SIB-Klassen benötigen spezielle Hilfsmethoden. Der SIB-Container enthält Funktionen zum:

Scannen
: SIB-Klassen werden mit einem Scan-Algorithmus auch außerhalb des Klassenpfades gesucht und geladen. Zur Identifikation von SIBs müssen die Klassen nicht mit einem ClassLoader geladen werden. Das jABC verfügt über einen eigenen Bytecodeparser, der die notwendigen Informationen (die Annotation und die UID) sehr schnell auslesen kann. Der Zugriff auf SIBs erfolgt ausschließlich über die SIB-UID.

Instanzieren
Der SIB-Container ist dafür verantwortlich, eine neue SIB-Komponente zu instanzieren. Die verwendete Java-Komponente im Speicher besteht neben der SIB-Klasse noch aus dem `SIBClassWrapper`, dem `SIBProxy` und der `SIBBase` Implementierung (vgl. Kapitel 3.8.3).

Refresh
Durch den `SIBClassLoader` können SIB-Java-Klassen nach Änderungen neu geladen werden.

Dependency Injection
Der Zugriff auf externe jABC-Objekte (wie z.B. die zugehörige `SIBGraphCell`) wird durch *Dependency Injection* realisiert. Als Dependency Injection bezeichnet man einen indirekten Zugang zu benötigten Objekten. Anstatt aktiv im Code ein Objekt zu beschaffen, wird das benötigte Objekt von aussen gesetzt oder *injiziert*.

Java-Bibliotheken werden als JAR[14]-Dateien verteilt und genutzt. Zum Erstellen von JAR-Dateien ist im JDK ein JAR-Archiver enthalten. SIB-Komponenten werden in Paletten organisiert. Diese Paletten werden wie eine Java-Bibliothek in Form einer JAR-Datei gebündelt. In einem Projekt des jABC-Prozesseditors können diverse SIB-Pfade angegeben werden. Ein SIB-Pfad ist ein JAR mit SIB-Komponenten oder ein Ordner im Dateisystem. Der SIB-Container kann sowohl Ordner als auch JAR-Dateien nach SIB-Klassen durchsuchen (Scan-Funktion).

3.8.3 Objektstruktur einer SIB-Komponente

Eine SIB-Java-Klasse, die von einem SIB-Experten entwickelt wird, ist nur ein kleiner Bestandteil des entsprechenden Java-Objektes, das vom SIB-Container später bereitgestellt wird. Ein SIB-Objekt im Speicher setzt sich aus folgenden Bestandteilen zusammen:

SIB-Klasse
Die SIB-Klasse ist die Komponentendefinition. Das SIB wird mit der `@SIBClass`-Annotation markiert, die die UID des SIBs festlegt. Eine Objektinstanz einer SIB-Klasse stellt den Speicherplatz für die veränderlichen Parameter der SIB-Komponente bereit.

SIBClassWrapper
Der `SIBClassWrapper` analysiert die SIB-Klasse über die Java Reflection-

[14] Java-ARchive (JAR)

API und speichert die Ergebnisse. Dieser Wrapper kennt alle SIB-Parameter, Branches und implementieren Plugin-Interfaces. Er stellt Methoden zur Verfügung, um die Werte der Parameter einer SIB-Instanz zu aktualisieren. Alle Instanzen einer SIB-Java-Klasse verwenden den gleichen `SIBClassWrapper`.

SIBBase
Grundlegende Methoden, die von allen SIB-Komponenten bereitgestellt werden müssen, sind in `SIBBase` zusammengefasst. Die Methoden, die Werte in der SIB-Instanz verändern, delegieren diese Aufrufe an den entsprechenden `SIBClassWrapper`.

SIBProxy
Das `SIBProxy` delegiert alle Aufrufe an die entsprechende SIB-Instanz oder an ein `SIBBase`-Objekt. Wird eine Methode aufgerufen, die aus einem Plugin-Interface stammt, das ein SIB nicht unterstützt, wird der Aufruf **ohne** Fehlermeldung abgefangen und beendet. Dies entspricht der leeren Implementierung der entsprechenden Methode.

Das Erzeugen dieser Objektstruktur für eine neue SIB-Instanz wird automatisch durch den SIB-Container vorgenommen (`SIBProxyFactory.newInstance()`). Im Prozesseditor wird ein neues SIB automatisch beim Einfügen in ein Prozessmodell erzeugt:

```
SIBGraphModel model =
      GraphModelHandler.getInstance().createModel();
SIBGraphCell sib =
      model.insert(new Point2D.Float(0,0), "sib-uid");
```

Die Klasse `SIBGraphCell` repäsentiert die grafische Instanz eines SIBs im Prozessmodell. Die SIB-Objektstruktur ist über die Methode `getSIB()` erreichbar. Der Aufruf einer Methode aus einem Plugin-Interface erfolgt durch Casting des Objekts auf das entsprechende Interface (hier am Beispiel des Tracer-Plugins 3.7.1 auf Seite 36):

```
((Executable) mysib.getSIB()).trace(execEnvironment);
```

Das `SIBProxy`-Objekt verhindert eine `AS`, falls das SIB das angeforderte Interface nicht implementiert. Alle SIB-Basismethoden sind im Interface `SIB` definiert und werden analog aufgerufen:

```
((SIB) mysib.getSIB()).getUID();
```

3.8.4 SIB-Adapter Entwurfsmuster

Wie das vorige Kapitel gezeigt hat, sind SIB-Komponenten sehr einfache Java-Klassen. Beim Entwurf einer SIB-Java-Klasse sollten einige Empfehlungen berücksichtigt werden. Wird eine Empfehlung nicht berücksichtigt, führt dies nicht zwingend zu Fehlern, aber die Qualität der SIB-Komponenten kann darunter leiden.

Anforderung S1 - Keine Klassen-Abhängigkeiten zum Prozesseditor
: Durch Abhängigkeiten zwischen Klassen aus dem Prozesseditor und SIB-Komponenten kann das SIB z.B. nach dem Codegenerieren nicht mehr funktionieren. Beim Codegenerieren mit einem Extruder-Generator (vgl. Kapitel 3.7.4 auf Seite 40) darf ein SIB-Experte die Funktionen des Frameworks uneingeschränkt nutzen. Für Pure-generierbare Klassen muss in der Implementierung sogar auch darauf verzichtet werden.

Anforderung S2 - Zustandsloses Verhalten
: Die SIB-Komponenten sollten keinen internen Zustand in SIB-Parametern oder Felder der Java-Klasse speichern. Für diese Aufgabe stellen Plugins wie der Tracer einen Kontext bereit.

Anforderung S3 - Service Entkopplung
: Die Entkopplung zwischen SIB und Service ist wichtig, um mit Komponenten und Modellen auch dann noch arbeiten zu können, wenn der repräsentierte Service temporär nicht installiert oder erreichbar ist. SIB-Klassen sind Komponentenbeschreibungen in Java. Die Implementierung sollte auf Klassenniveau von der Komponentendefinition getrennt werden[15] (vgl. Anforderung T3).

Das *SIB-Adapter Entwurfsmuster* oder kurz *SIB-Pattern* entkoppelt die Service-Implementierung von der Komponentendefinition eines SIBs. Dieses Pattern unterteilt die SIB-Implementierung in drei Schichten:

SIB
: Die eigentliche SIB-Java-Klasse ist leichtgewichtig. Sie enthält die SIB Parameter und Branches. Die implementierten Plugin-Interfaces deligieren alle Aufrufe zu einer Adapterklasse.

Adapter
: Die Adapterklassen können von mehreren SIBs einer SIB-Palette gemeinsam genutzt werden. Dabei kapselt der Adapter eventuell notwendige technische Verbindung an den jeweiligen Service (z.B. Webservice, Corba oder JNI[16] Aufrufe)

[15] vergleichbar einem Interface und der Interfaceimplementierung
[16] Java Native Interface (JNI) ist eine Java API um native Bibliotheken aufzurufen

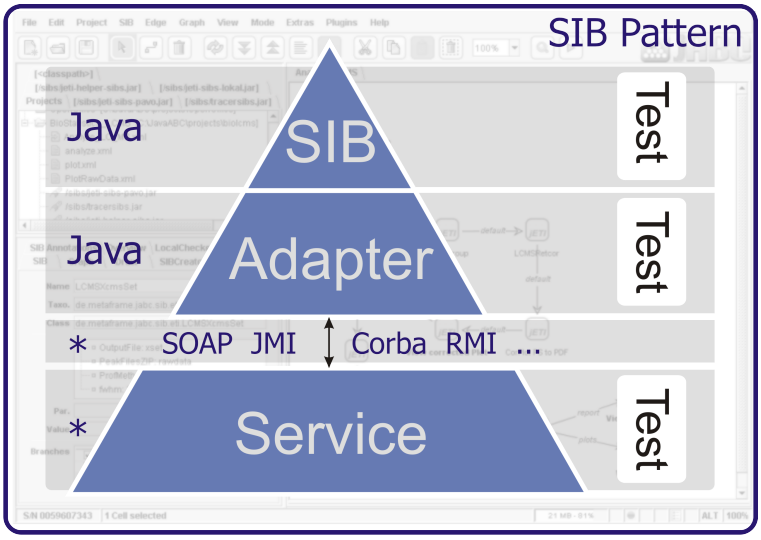

Abbildung 3.12: SIB-Adapter Entwurfsmuster

Service

Der Service ist eine beliebige Software-Funktionalität, die von einer Java-Adapterklasse aufgerufen werden kann. Für einen Service gilt also:

- Die Programmiersprache kann frei gewählt werden.
- Die Schnittstelle kann frei gewählt werden.
- Der Host kann frei gewählt werden.

Diese Definition eines SIB-Services ist so universell, dass nahezu jede andere Spezifikation einer Software-Komponente eingeschlossen ist.

SIB-Paletten werden als JAR-Archiv gebündelt. Bei SIBs, die nach dem SIB-Pattern entworfen wurden, enthält das Archiv sowohl SIBs als auch die dazugehörigen Adapter. Das jABC lädt über den SIB-Container angeforderte SIB-Java-Klassen. Dabei ist es wichtig, dass alle von der SIB-Klasse referenzierten Klassen, die durch die Importanweisungen im Java Sourcecode definiert wurden, im Klassenpfad erreichbar sind. Werden in der SIB-Klasse keine Service-Objekte sondern nur die Adapter-Klassen referenziert, kann die SIB-Klasse auch dann geladen werden, wenn der eigentliche Service nicht verfügbar ist. Wird über ein Plugin die Ausführung einer Komponente angefordert, für die auf dem aktuellen Computer Bestandteile fehlen (z.B. Bibliotheken), so erzeugt die JVM eine Exception (für fehlende Java-Klassen z.B. eine `ClassNotFoundException`). Plugins wie der Tracer fangen solche Fehlermeldungen ab und beenden die Ausführung mit einer entsprechenden Meldung.

Die drei Schichten der SIB-Patterns können voneinander unabhängig getestet werden. Der Test der Serviceschicht sollte vom jeweiligen Service-Provider vorgenommen werden. Dieser Test ist vollkommen unabhängig davon, ob der Service als SIB genutzt werden soll. Der Test der Adapterklasse muss sicherstellen, dass der Adapter die API des Services korrekt anwendet. Da in der SIB-Java-Klasse keinerlei Logik enthalten ist, wird in der Regel dafür kein Unittest benötigt. Zum Testen der SIB-Komponente können mit dem jABC verschiedene Testmodelle erstellt werden, die dann ausgeführt werden.

3.8.5 SIB-Beispiel

Das folgende Beispiel zeigt eine sehr einfache SIB-Komponente. Das `MailSIB` kann eine E-Mail an einen Empfänger versenden. Der entsprechende Service wird durch die JavaMail-Bibliothek bereitgestellt [SUN09a]. Das SIB verfügt über die folgenden Parameter:

To: der Empfänger der E-Mail

From: der Absender der E-Mail

Subject: der Betreff der E-Mail

Message: die Nachricht

Das SIB wird über den `Default`-Branch verlassen, falls die Mail erfolgreich versendet wurde. Tritt irgendein Fehler auf, wird die `Error`-Kante verwendet. Das SIB unterstützt das Tracer-Plugin und den LocalChecker. Bei der Ausführung mit dem Tracer wird die E-Mail versendet. Der LocalChecker prüft, ob die verwendete E-Mail Adressen das übliche Format einhalten.

MailSIB.java

```
package de.jabc.sib;

import de.jabc.adapter.MailAdapter;
import de.metaframe.jabc.sib.Executable;
import de.metaframe.jabc.sib.LocalCheck;
import ... // some imports deleted

@SIBClass("a5690335-30b4-4f6e-8b26-71a897af4c44")
public class MailSIB implements LocalCheck, Executable {

  // immutable SIB-Branches
   public static final String[] BRANCHES = new String[]{
    SIB.DEFAULT, SIB.ERROR};
```

```
  // SIB-Parameter
  public String to      = "john.doe@mail.com";
  public String from    = "jane.doe@mail.com";
  public String subject = "Subject";
  public String message = "Messagetext";

  // Tracer-Interface
  public String trace(ExecutionEnvironment execEnv) {
    return MailAdapter.sendMail(to, from, subject, message);
  }

  // Basic E-Mail regexp pattern
  static final String EMAIL_PATTERN = ".+@.+\\.[a-z]+";

  // Localchecker-Interface
  public void checkSIB(SIB sib) {
    if (Pattern.matches(EMAIL_PATTERN, to))
      SIBUtilities.addError(sib,
        "Illegal receiver E-Mail adress!");
    if (Pattern.matches(EMAIL_PATTERN, from))
      SIBUtilities.addError(sib,
        "Illegal sender E-Mail adress!");
  }

  // Helper class for Icons
  static IconCache ICON = new IconCache(MailSIB.class);
  // Helper class for I18n SIB documentation
  static Documentation DOC = new Documentation(MailSIB.class);

  // get the documentation from a propertyfile
  public String getDocumentation(DocType type, String param) {
    return DOC.get(this, type, param);
  }

  // read Icon from IconCache
  public Object getIcon() {
    return ICON.get("mail-sib");
  }
}
```

Das MailSIB nutzt den `MailAdapter`, um über die JavaMail-API eine Mail zu versenden. In diesem Fall benötigt der Adapter noch einen entsprechenden SMTP-Host, über den die E-Mail versendet werden soll. Da der SMTP-Host vom

entsprechenden Provider abhängig ist, wurde dieser nicht als SIB-Parameter definiert. Der Wert (*mail.smtp.host*) wird aus einer entsprechenden Propertydatei (*javamail.properties*) ausgelesen. Wird das SIB später in Modellen verwendet, kann der SMTP-Host angepasst werden, ohne das entsprechende Prozessmodell anzupassen oder neu zu kompilieren.

MailAdapter.java

```
package de.jabc.adapter;

import javax.mail.Message;
import ...

// Adapter class for MailSIB
public class MailAdapter {

  static ClassLoader cl = MailAdapter.class.getClassLoader();
  static final String PROPERTIES = "javamail.properties";

  // Send a mail via JavaMail-API
  public static String sendMail(String to, String from,
      String subject, String message) {

    try {
      // read properties
      Properties props = new Properties();
      props.load(cl.getResourceAsStream(PROPERTIES));

      // prepare the email
      Session session = Session.getDefaultInstance(props);
      Message msg = new MimeMessage(session);

      msg.setFrom(new InternetAddress(from));
      msg.setRecipient(Message.RecipientType.TO,
        new InternetAddress(to));
      msg.setSubject(subject);
      msg.setContent(message, "text/plain");

      // Send the mail
      Transport.send(msg);

      // OK
      return SIB.DEFAULT;

    } catch (Exception e) {
```

```
        // Error
        return SIB.ERROR;
      }
    }
}
```

3.9 Fehlertoleranz

Durch die Framework-Architektur kann der Funktionsumfang des jABCs in Abhängigkeit der installierten Plugins sehr unterschiedlich ausfallen. Das erweiterbare Prozessmodell des jABCs erlaubt es den Plugins sogar, eigene Daten und Objekte mit einem Modell zu speichern. Wenn Prozessmodelle zwischen verschiedenen jABC-Installationen ausgetauscht werden, könnte dies zu Problemen führen. Folgender Grundsatz wurde daher beim Design des jABCs berücksichtigt:

> *Jedes Prozessmodell kann mit jeder jABC Installation unabhängig von den installierten Plugins geöffnet, bearbeitet und verlustfrei wieder gespeichert werden.* (Anforderung T4: Fehlertoleranz)

Um diese Anforderung jederzeit zu gewährleisten mussten mehrere Problemfälle betrachtet und gelöst werden.

Fehlende/Inkompatible SIBs

SIBs in einem Prozessmodell werden beim Speichern in ein XML-Dokument über die SIB-UID referenziert. Beim Parsen und Laden eines Prozessmodells kann es vorkommen, dass kein SIB mit der angegebenenen UID gefunden wird. Für diese Fehlersituation wurde das ProxySIB entwickelt. Das ProxySIB ist das einzige SIB, das beliebige Parameter und Branches akzeptiert. Dies ist nur durch eine Sonderbehandlung des ProxySIBs im SIB-Container und im Persistenzmechanismus möglich. Ein SIB wird auch dann durch ein ProxySIB ersetzt, falls die im XML vorliegenden Parameter nicht mit den Feldern der vorhandenen SIB-Java-Klassen zusammenpassen. Alle gesetzten Parameter eines ProxySIBs können wie gewohnt im SIB-Inspektor bearbeitet werden. Beim Speichern werden die Parameter zusammen mit der ursprünglichen SIB-UID wieder im XML-Dokument abgelegt. Sobald das ursprüngliche SIB wieder verfügbar ist, wird es beim nächsten Öffnen des Modells automatisch wieder geladen. ProxySIBs werden durch ein besonderes Icon dargestellt

und die Statusleiste des Prozesseditors zeigt an, sobald beim Laden ein SIB durch ein ProxySIB ersetzt werden mußte.

Fehlende Plugins

Plugins haben im jABC die Möglichkeit, das Prozessmodell durch eigene Objekte zu erweitern. Diese Objekte werden als *UserObjects* bezeichnet. UserObjects können als *persistent* oder *temporär* gekennzeichnet werden. Persistente UserObjects müssen von der Persistenz-API im XML-Dokument des Modells mitgespeichert werden. Der Persistenz-Mechanismus des jABC basiert auf der Bibliothek XStream [Joe08]. XStream verwendet ein modulares, erweiterbares Konverterkonzept zur Speicherung beliebiger Javaobjekte. Jeder Konverter ist für die Serialisierung und Deserialisierung von bestimmten Objektklassen spezialisiert. Für alle gängigen Javaobjekte[17] sind entsprechende Konverter bereits in XStream vorhanden. Das jABC fügt eigene Konverter für Prozessmodelle und SIBs hinzu. Plugins können weitere Konverter bereitstellen. Wird ein Prozessmodell geladen, das UserObjects von einem Plugin enthält welches in diesem jABC nicht vorhanden ist, können Konverter oder Objekte fehlen, die zur Deserialisierung des Modells benötigt werden. Die Persistenz-API des jABCs verwendet dazu einen *partiellen Deserialisierungmechanismus*. Stößt der Algorithmus auf einen Teilbaum im XML, der von keinem Konverter deserialisiert werden kann, wird das entsprechende XML-Fragment als String in einem Hilfsobjekt namens `MissingObject` abgelegt. Beim Speichern wird dieses Fragment unverändert wieder im XML abgelegt. Wird das Modell danach wieder in einem jABC mit dem benötigten Plugin eingelesen, sind die vom Plugin abgelegten Objekte unverändert vorhanden.

Fehlende Modelle

GraphSIBs werden im jABC dazu verwendet, andere Modelle als SIB zu referenzieren. Die Referenz wird analog zu den SIB-UID über eine generierte Model-ID gespeichert. Fehlt das referenzierte Modell, ist die Verzweigung zu diesem Modell nicht möglich. Das GraphSIB wird im XML-Dokument wie jedes andere SIB serialisiert, sodass ein fehlendes Modell bei der Deserialisierung eines GraphSIBs keinerlei Auswirkungen hat. GraphSIBs, zu denen das referenzierte Prozessmodell nicht gefunden wurde, werden durch ein besonderes Piktogramm gekennzeichnet. Sobald das fehlende Modell wieder verfügbar ist, ist die Referenz im Ursprungsmodell wieder intakt.

Fehlende Services

Mit dem SIB-Adapter Entwurfsmuster kann eine lose Kopplung zwischen SIB und dem zugehörigen Service leicht umgesetzt werden (vgl Kapi-

[17] Strings, Zahlen, Listen, Maps usw.

tel 3.8.4 auf Seite 47).

Defekte XML-Dokumentstruktur

Die Struktur des XML-Dokuments, in dem das jABC-Prozessmodell abgelegt wird, könnte zum Beispiel durch defekte Blöcke eines Massenspeichers beschädigt sein. In diesem Fall gibt es für das jABC keine Möglichkeit, dieses Dokument wieder herzustellen. Einige moderne Filesysteme wie ZFS [SUN09c] verwenden CRC-Summen, um solche Fehler festzustellen. Bei redundanter Speicherung können defekte Blöcke automatisch durch Kopien ersetzt werden (ZFS Self-Healing[18]). In Java und dem jABC gibt es keine Möglichkeit, defekte XML-Dokumente zu retten. Der Prozesseditor bietet als Schutz eine lokale Versionierung der XML-Dokumente an. Dabei werden ältere Versionen des XML-Prozessmodells nicht überschrieben, sondern mit fortlaufenden Versionsnummern umbenannt. Der jABC-Anwender kann bei Defekten auf eine kontinuierliche Versionsgeschichte von Prozessmodellen zurückgreifen. Die Anzahl der insgesamt gespeicherten Versionen ist begrenzbar. Eine spezielle Funktion des Prozesseditors (*Purge Model*) hilft beim Beseitigen von überflüssigen Sicherheitskopien.

[18] http://opensolaris.org/os/community/zfs/demos/selfheal/

4 ContentEditor

Das ContentEditor-Projekt entstand aus der Notwendigkeit, Zusatzinformationen an SIBs hinterlegen zu können, die von Plugins automatisch ausgewertet werden können. In früheren Versionen des jABCs gab es einen Rich-Text-Editor[1], mit dem zu jedem SIB ein HTML-Text gespeichert werden konnte. In diesen, als *SIB-Doc* bezeichneten HTML-Texten konnten die grundlegenden HTML-Tags wie Überschriften, Aufzählungen, Tabellen und Fonteigenschaften verwendet werden. SIB-Doc-Informationen wurden vom jABC direkt im Prozessmodell gespeichert. Ein gravierendes Problem der SIB-Docs war es, dass in den HTML-Texten wichtige Informationen mit Fließtext gemischt wurden.

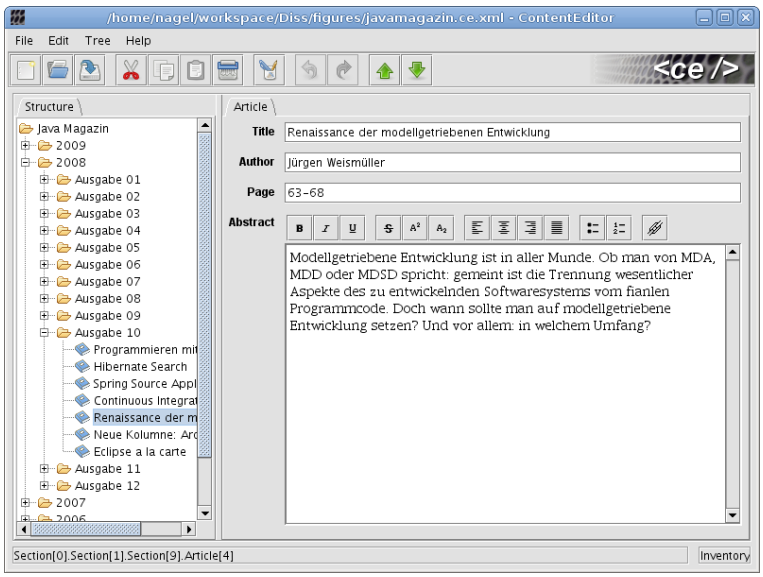

Abbildung 4.1: Zeitschriftverzeichnis im ContentEditor

Betrachten wir hierzu als Beispiel die Definition von Benutzerrechten, die zur Ausführung eines Services in einem Kontrollflussgraph notwendig sind. Diese

[1] Ekit: ein in Java realisierter HTML-Editor [How07]

Information muss von einem Codegenerator in geeigneter Weise in den Sourcecode übertragen werden. Die Art und Weise hängt dabei vom verwendeten Sicherheitskonzept und -framework ab. Deshalb können die Codegeneratoren des Genesys-Plugins [JMS08], die selbst als Prozessmodelle realisiert wurden, auf unterschiedliche Konzepte und Frameworks einfach angepasst werden. Die Definition des Benutzerrechts als SIB-Parameter direkt im SIB abzulegen scheitert daran, dass dann jedes SIB für alle denkbaren Arten von Zusatzinformationen passende SIB-Parameter bereitstellen muss. Im SIB-Doc können dagegen beliebige Information hinterlegt werden. Steht die Information allerdings an einer undefinierten Stelle im SIB-Doc, ist eine automatische Auswertung unmöglich.

Wie dieses Beispiel zeigt, wurde für das jABC ein Editor für verschiedene Arten von Zusatzinformationen benötigt. Dieser Editor sollte allerdings kein Bestandteil des jABCs sein, sondern als eigenständiges Projekt realisiert werden, das auch unabhängig vom jABC eingesetzt werden kann. Folgende Anforderungen sollte der ContentEditor erfüllen:

Anforderung C1 - Strukturierte Informationen
: Das System soll unterschiedliche Informationen speichern können, die in einer hierarchischen Struktur angeordnet sind. Die verschiedenen Arten von Informationen werden nach Name und Datentyp unterschieden. Für die Eingabe der Informationen in einer grafischer Benutzeroberfläche sollten den verschiedenen Datentypen adäquate Editorkomponenten zugeordnet werden können.

Anforderung C2 - Flexibler Zugriff
: Die Informationen sollen flexibel manuell oder automatisiert zugreifbar sein. Die Zugriffe können über den Namen, Datentyp, Inhalt oder Position innerhalb der Struktur erfolgen.

Anforderung C3 - Regelkonformität
: Das System soll die Struktur der Informationen anhand von Regeln überprüfen können. Bei Regelverletzungen muss dem Anwender eine Hilfestellung angeboten werden, um die Regelkonformität wiederherzustellen.

Anforderung C4 - Fehlertoleranz
: Auch Informationsstrukturen, die nicht den angegebenen Regeln entsprechen, müssen gespeichert werden können, falls eine Korrektur der Fehler nicht augenblicklich möglich ist.

Im jABC bezeichnen wir eine solche Information als *Annotation*[2]. Eine Annotation ist eine Zusatzinformation, die nicht als SIB-Parameter gespeichert wird

[2]Dieser Begriff wurde an Java-Annotationen angelehnt.

und von verschiedenen Plugins ausgewertet werden kann. Damit eine Annotation automatisch ausgewertet werden kann, müssen folgende Eigenschaften festgelegt sein:

Eigenschaft A1 - Name
Jede Annotation soll über einen eindeutigen Namen angesprochen werden können.

Eigenschaft A2 - Datentyp
Der Datentyp legt fest, welche Information in der Annotation gespeichert werden kann. Weiterhin wird über den Datentyp auch die grafische Komponente festgelegt, mit der die Information editiert wird.

Eigenschaft A3 - Verbindlichkeit
Annotationen können als verbindlich oder freiwillig gekennzeichnet werden. Diese Eigenschaft wird als Teil der Strukturregeln festgelegt.

Eigenschaft A4 - Häufigkeit
Ob es einen oder mehrere Werte für eine Annotation gibt, wird ebenso als Teil der Strukturregeln hinterlegt.

4.1 Architektur

Der ContentEditor wurde als selbständiges Projekt realisiert. Die Informationen im ContentEditor werden als Baum gespeichert (Anforderung C1: Strukturierte Informationen). Ein austauschbares SIB-Modell im ContentEditor steuert:

- aus welchen Objekten der Baum zusammengesetzt ist (A1, A2).
- wann die Struktur eines Baums regelkonform ist (A3, A4).

Dazu wurde eine neue jABC-Modellsemantik definiert, die Railroad-Diagramme zur Beschreibung kontextfreier Grammatiken nutzt. Diese Semantik wird als *ContentEditor-Grammatik* bezeichnet. Die *Terminale* und *Nicht-Terminale* der Grammatik entsprechen den Knoten des Baums. Der Anwender kann den Baum frei aus den Elementen zusammensetzen, die in der Grammatik enthalten sind. Alle Baumknoten können Informationen aufnehmen, unabhängig davon, ob diese in der Grammatik Terminale oder Nicht-Terminale sind. Welche Art von Information gespeichert wird, hängt vom Typ des Baumknotens ab.

Die Eigenschaften A1 und A2 werden durch unterschiedliche Knotentypen im Baum realisiert. Die Baumknoten werden von spezialisierten SIB-Komponenten

produziert, aus denen die zugehörige Grammatik zusammengesetzt ist. Der Name eines Knotens (Eigenschaft A1) wird als Parameter des entsprechenden SIBs festgelegt. Der Datentyp wird durch die verwendete SIB-Komponente bestimmt (Eigenschaft A2). Die Baumknoten verfügen jeweils über spezialisierte Editierfunktionen. Auf einer Baumstruktur wird die Anforderung C2: Flexibler Zugriff leicht durch rekursive Suchmethoden erfüllt.

Die Anforderung C3: Regelkonformität wird durch die zugehörige ContentEditor-Grammatik geprüft. Diese Grammatik legt die Regeln fest, mit denen der Baum validiert wird. Die Eigenschaften A3: Verbindlichkeit und A4: Häufigkeit werden wie in Railroad-Diagrammen durch die grafischen Regeln festgelegt. Eine Regel legt die Eigenschaft A3 fest, indem es einen "Umweg" um ein entsprechendes SIB gibt oder nicht. Entsprechend gibt es für die Eigenschaft A4 einen Weg zurück, um ein SIB mehrfach zu besuchen.

Zur Prüfung der Regelkonformität (Anforderung C3) wird der Baum im ContentEditor als *Strukturbaum* aufgefasst, der aus *Terminalen* und *Nicht-Terminalen* besteht. Ausgehend von der Wurzel prüft der ContentEditor rekursiv, ob die Kindknoten im Baum den Regeln der Grammatik entsprechen. Zur Vereinfachung des Algorithmus müssen die Grammatikregeln eine festgelegte, einfache Form einhalten. Sobald ein Knoten gefunden wird, der nicht einer Regel entspricht, wird dieser und der Teilbaum darunter als fehlerhaft markiert. Damit der Benutzer diese Fehler im Strukturbaum bereinigen kann, wird für selektierte Knoten der besuchte Pfad in der Grammatik hervorgehoben. So kann der Benutzer anhand der Grammatik nachverfolgen, welche Knoten an der selektierten Stelle verwendet werden dürfen (vgl. Abbildung 4.2 auf Seite 61).

Der Editor bietet eine Reihe von Funktionen an, um die Baumstruktur zu bearbeiten. Dies sind: *Erzeugen*, *Löschen*, *Duplizieren* oder *Verschieben* von Knoten im Baum. Die resultierende Baumstruktur kann nach einer Operation nicht mehr regelkonform sein. Beim Laden und Speichern der Struktur wird die ContentEditor-Grammatik **nicht** verwendet. Die aktuelle Datenstruktur des Baums wird als XML-Datei gespeichert (serialisiert). Diese Funktion wird von der Bibliothek XStream [Joe08] bereitgestellt. So können auch fehlerhafte Baumstrukturen problemlos gespeichert werden (Anforderung C4: Fehlertoleranz).

4.2 ContentEditor-Grammatik

Eine Grammatik für den ContentEditor wird als jABC-Graph mit Hilfe einer spezialisierten SIB-Palette modelliert. Die SIBs unterstützen neben dem

`ContentEditor`-Interface zusätzlich das `LocalChecker`-Interface. Über Localcheck-Code wird geprüft, ob das Modell die Anforderungen an eine Content-Editor-Grammatik erfüllt. Die SIB-Palette besteht aus folgenden SIBs:

NodeSIBs

Ein NodeSIB repräsentiert einen Baumknoten in der Grammatik. Verschiedene Datentypen werden durch unterschiedliche NodeSIBs realisiert. Jedes NodeSIB kann Baumknoten über die Methode `createNode()` produzieren. Um die Daten eines Baumknotens bearbeiten zu können, implementiert jeder Baumknoten die Methode `getEditor()`, die eine passende Swing-Editorkomponente erzeugt. Durch die Erweiterung der Content-Editor-SIB-Palette um neue NodeSIBs können weitere Datentypen oder Editorkomponenten hinzugefügt werden.

TextSIB
: speichert eine Zeichenkette, die über eine Textbox editiert wird. Die Editorkomponente kann ein- oder mehrzeilig verwendet werden (`JTextArea, JTextField`).

RichTextSIB
: verwendet einen Rich-Text-Editor zur Eingabe einer HTML Zeichenkette (`Ekit`).

NumberSIB
: speichert eine ganze Zahl. Zur Eingabe wird ein spezialisiertes Textfeld verwendet (`JFormattedTextField`).

DateSIB
: entspricht dem NumberSIB für Datumseingaben (`JXDatePicker`).

PasswordSIB
: verwendet ein Textfeld mit einer verdeckten Eingabe für Passwörter (`JPasswordField`).

ListBoxSIB
: realisiert die Auswahl eines Textes aus einer Menge von Vorgaben (`JComboBox`).

ExternalSIB
: verwaltet einen Link zu einer externen Datei oder Ressource. Der Link wird im URL-Format gespeichert. Das GUI-Element verwaltet zusätzlich eine Liste von unterschiedlichen Editoren, die zum Bearbeiten des Links aufgerufen werden können.

TableSIB
: realisiert eine Tabelle mit einer variablen Spaltenbreite. Zur Laufzeit können Zeilen gelöscht oder hinzugefügt werden (`JXTable`).

CompoundSIB
: ist eine zusammengesetzte Annotation, die aus mehreren NodeSIBs

besteht. Als GUI wird ein automatisch generiertes Formular verwendet, das aus den verschiedenen GUI-Elementen der einzelnen NodeSIBs besteht. Die Werte des CompoundSIBs werden als Liste der Werte des enthaltenden NodeSIBs abgespeichert. Die Verwendung von CompoundSIBs innerhalb von CompoundSIBs wird derzeit nicht unterstützt.

SwiXmlSIB
ist eine Alternative zum CompoundSIB. Anstatt eines generierten Formulars wird hier ein per XML spezifizierter Dialog verwendet. Für die Generierung des Swing Dialogs und als XML-Dialekt wird die SwiXML [Wol09] Bibliothek verwendet (`SwiXml`).

RefSIB
mit RefSIBs werden Links zu weiteren NodeSIBs definiert. In der ContentEditor-Grammatik wird über RefSIBs die Vater-Kind-Beziehung definiert.

AcceptSIB
markiert das Ende einer Grammatik Regel. Im ContentEditor wird beim Erreichen eines AcceptSIBs die Verarbeitung der nächst höheren Ebene des Baums fortgeführt (Rekursionsende).

NoopSIB
Sind funktionslose SIBs, die nur zur Vereinfachung der Kantenabfolgen dienen.

Der ContentEditor analysiert beim Start die konfigurierte Grammatik. Diese Grammatik legt durch die enthaltenden NodeSIBs fest, welche Knotenarten im Baum vorkommen können (*Create Node*). Nach jeder Aktion im Content-Baum (Hinzufügen, Duplizieren, Verschieben oder Löschen) wird anhand der Grammatik überprüft, ob der aktuelle Baum gemäß der Grammatik alle Regeln weiterhin erfüllt. Knoten, die nicht zu den Regeln passen, werden als fehlerhaft markiert.

Ein *Track* ist eine Regel einer ContentEditor-Grammatik. Zur Vereinfachung des Validierungsalgorithmus wird eine gewisse Form der Tracks vorausgesetzt. Die Wurzel des ContentEditor-Baums wird durch das StartSIB der Grammatik bestimmt. Die Grammatik muss deterministisch sein. Aus dem Strukturbaum muss eindeutig der Pfad der Grammatik hervorgehen. Für die ContentEditor-Grammatiken gelten die folgenden einfachen Regeln, die als Localcheck-Code in den SIBs hinterlegt sind:

- NodeSIBs haben mindestens eine ausgehende Kante.
- AcceptSIBs haben keine ausgehende Kante.
- Von jedem SIB kann ein AcceptSIB erreicht werden.

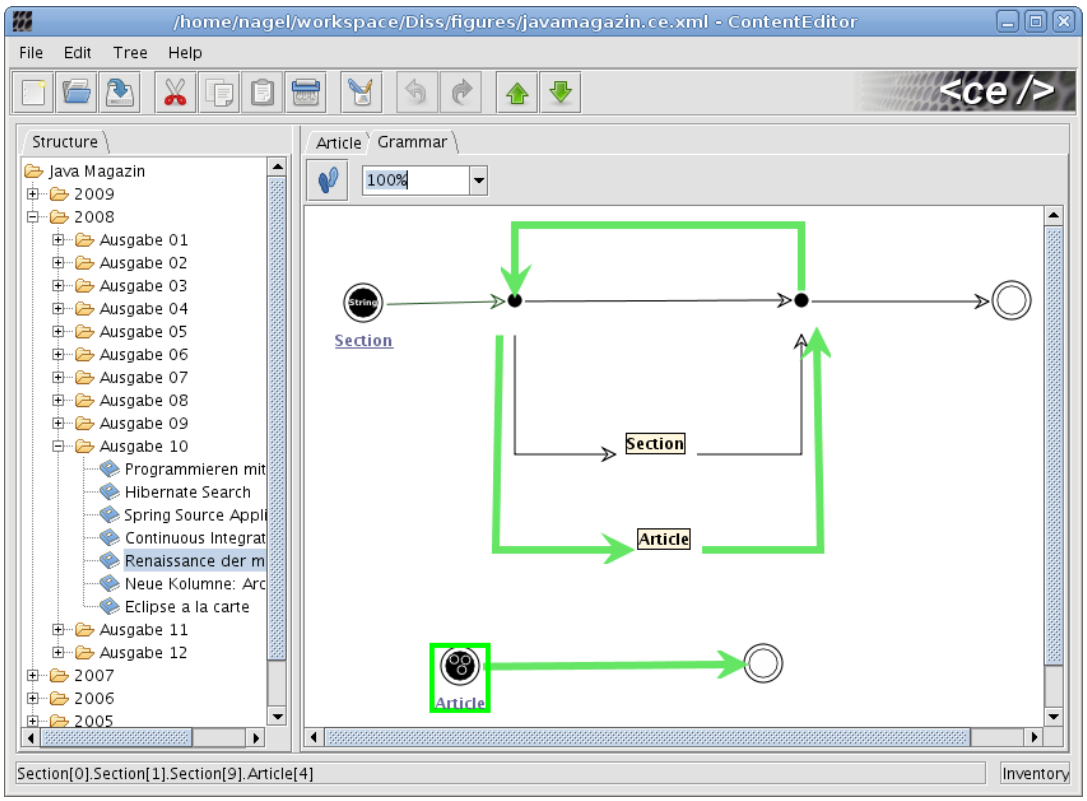

Abbildung 4.2: Validierung einer Grammatik

- Es gibt nur genau ein NodeSIB, das als StartSIB markiert ist.
- Jedes NodeSIB hat einen eindeutigen Namen.
- Zu jedem RefSIB gibt es genau ein über den Namen referenziertes NodeSIB.
- Die Grammatik ist deterministisch.

Abbildung 4.3 auf der nächsten Seite zeigt ein Beispiel einer Grammatik zur Definition der Webservice-Parameter des jETI-Systems. Die Abbildung 4.6 auf Seite 65 zeigt dazu einen mit dieser Grammatik erzeugten ContentEditor-Baum.

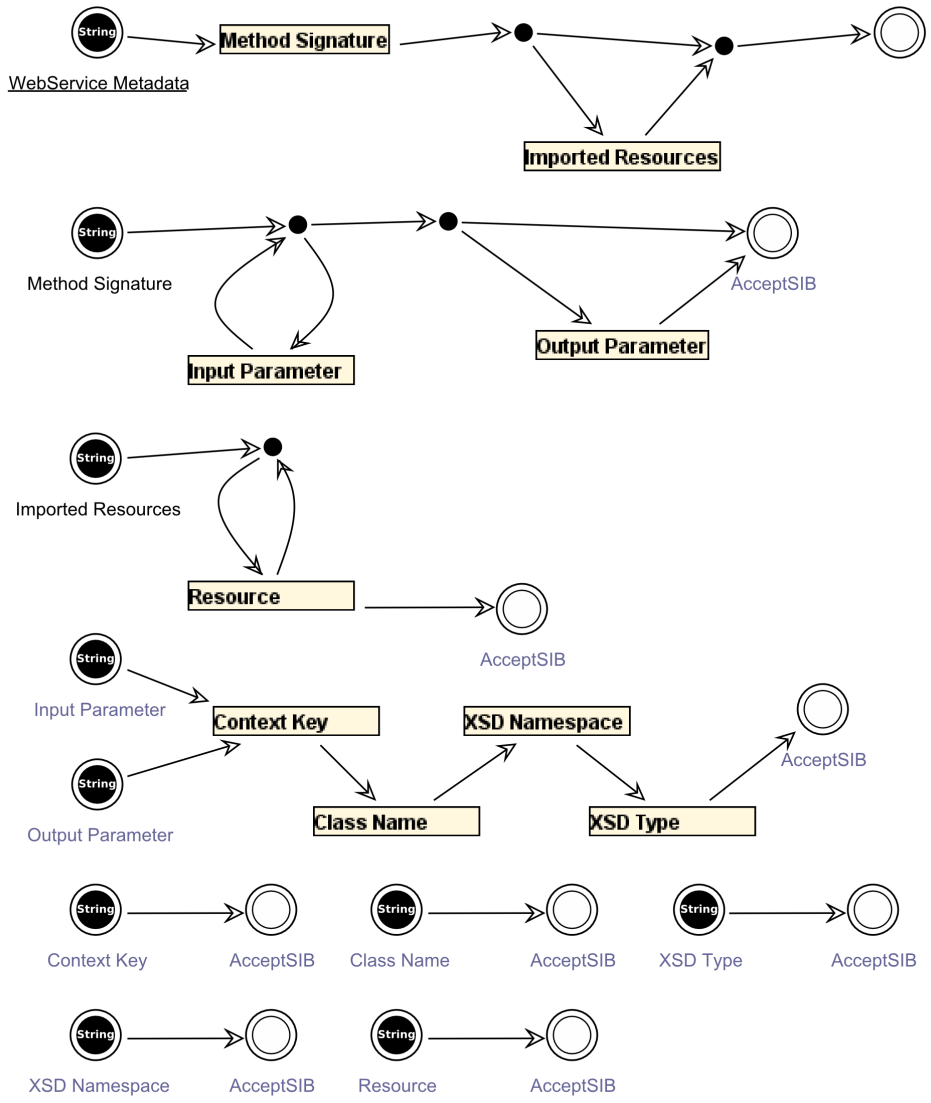

Abbildung 4.3: Beispiel einer ContentEditor-Grammatik

4.3 Produktlinien

Mit dem ContentEditor wurden eine Reihe von spezialisierten Editoren für unterschiedliche Aufgaben erstellt. Neben den verschiedenen Grammatiken unterscheiden sich die Produktlinien auch durch einige angepasste Funktionen.

XML-Editor

Der XML-Editor ist ein Editor für beliebige XML-Dateien. Die zugehörige ContentEditor-Grammatik zeigt die typischen Elemente wie `element`, `#text`, `#cdata` und `#comment`.

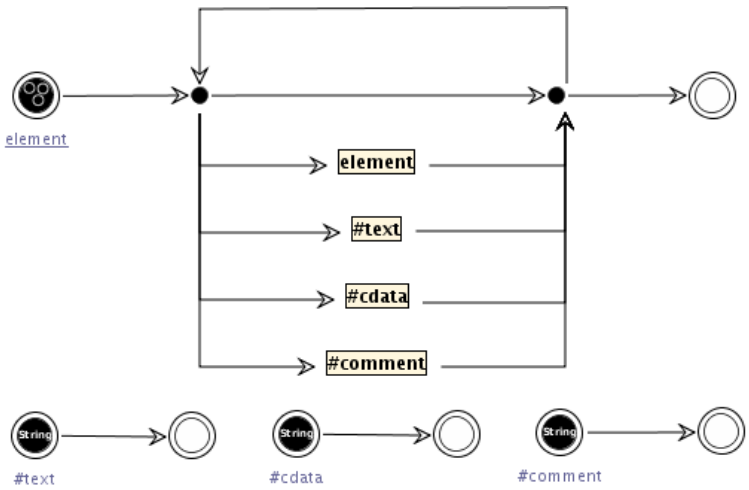

Abbildung 4.4: ContentEditor-Grammatik für XML

Für diesen Editor wurden die Funktionen *Datei öffnen* und *Datei speichern* alternativ implementiert. Anstatt das ContentEditor-XML-Format zu verwenden[3], wird im XML-Editor direkt der DOM-Parser[4] Xerces [Apa07c] eingesetzt. Der vom Parser erzeugte DOM-Baum wird beim Öffnen in die Baum-Datenstruktur des ContentEditors übersetzt. Die Validierung der Grammatik des XML-Editors gewährleistet hierbei allerdings nur die allgemeine Struktur eines XML-Dokumentes. Eventuell vorhandene DTD- oder XML-Schema-Definitionen werden zur Zeit noch nicht ausgewertet.

PWManager

Beim PWManager handelt es sich um einen einfachen Passwortmanager. Ein Passworteintrag ist ein ContentEditor-Knoten, der sich aus mehreren elementaren Bestandteilen zusammensetzt (CompoundNode bestehend aus: Name, User, URL, Password und Kommentar). Das GUI-Formular ist mit dem SwiXml-Knoten realisiert worden. Über die Toolbar sind zusätzliche Funktionen verfügbar:

[3]Das ContentEditor XML Format wird mit der Bibliothek XStream erzeugt [Joe08]
[4]Document Object Model (DOM) ist eine gängige API für den Zugriff auf XML-Dokumente

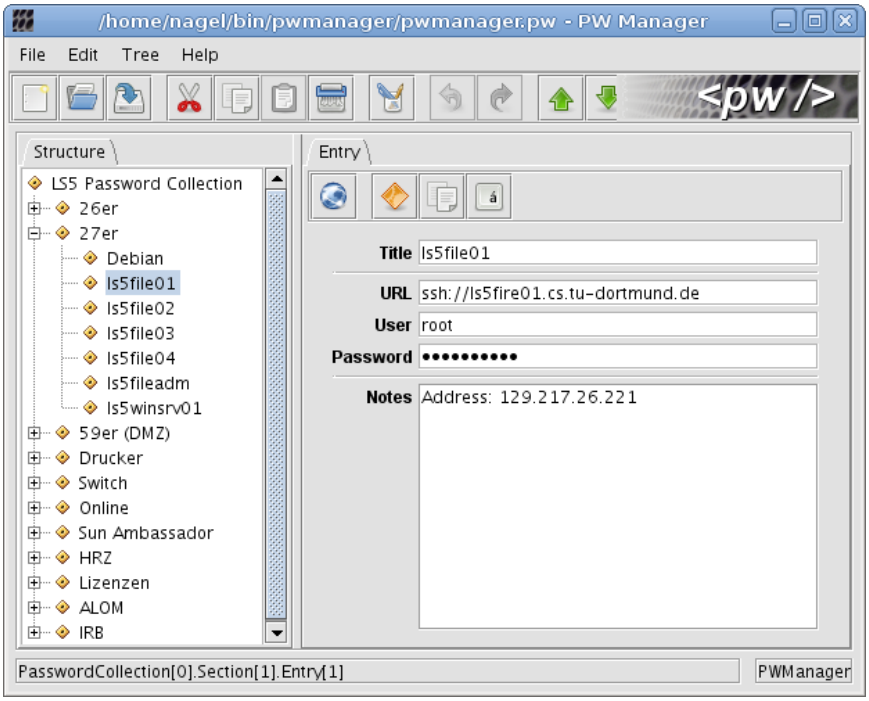

Abbildung 4.5: SwiXml-Passworteintrag im PWManager

- Browser mit der angegebenen URL öffnen
- Passwort aufblenden
- Passwort in die Zwischenablage kopieren
- zufälliges Passwort generieren

Auch beim PWManager wurden die Funktionen *Datei öffnen* und *Datei speichern* alternativ implementiert. Das ContentEditor-XML-Format wird beim PWManager zusätzlich noch verschlüsselt. Dazu wurde ein weiterer Dialog programmiert, der die Eingabe eines Passworts beim Öffnen und Speichern ermöglicht.

AnnotationEditor

Der AnnotationEditor ist die Variante des ContentEditors für den Einsatz im jABC. Beim AnnotationEditor erfolgt die Verwaltung der verwendeten Grammatik innerhalb des jABC. Es werden vier verschiedene Grammatiken unterschieden:

- die Grammatik für SIB-Komponenten
- die Grammatik für Kanten
- die Grammatik für ein Prozessmodell
- die Grammatik für ein Projekt

Die Definition der Grammatiken kann für jedes jABC-Projekt separat erfolgen. Die vom jABC mitgelieferten Grammatiken sind Vorlagen, die an die Anforderungen des jeweiligen Projekts angepasst werden müssen. Für das erwähnte Beispiel zur Definition von Benutzerrechten im Prozesseditor muss die Grammatik für die SIB-Komponenten um einen Knoten zur Angabe der Benutzerrechte erweitert werden. Dabei kann durch die Struktur der Grammatik definiert werden, ob dieser Knoten optional oder verpflichtend ist und einfach oder mehrmalig vorkommen darf. Durch die Validierung der Grammatik wird dem Benutzer im AnnotationEditor visuell angezeigt, falls er gegen die Regel der Grammatik verstoßen hat.

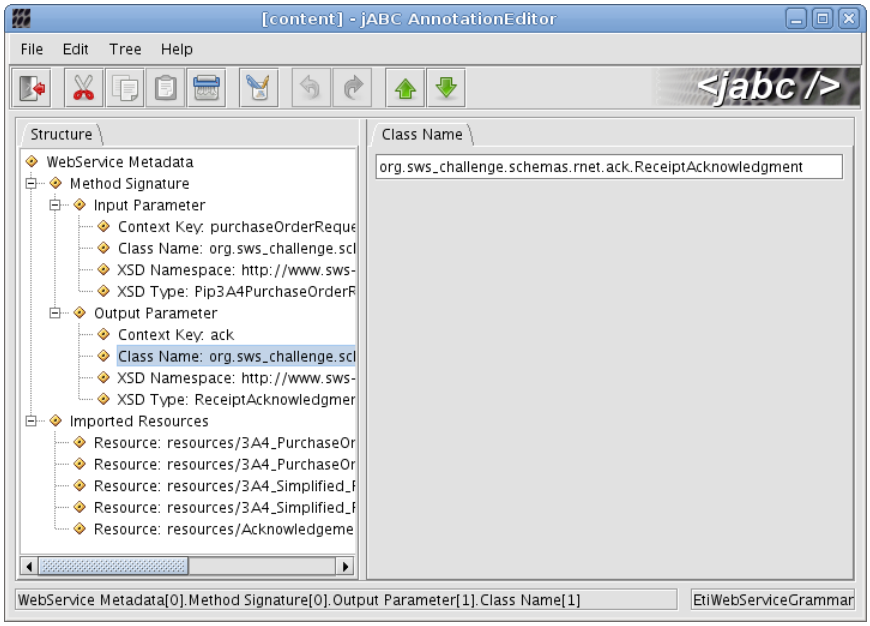

Abbildung 4.6: jETI Webservice Annotation

Zusätzlich verwaltet das jABC die laufenden Instanzen des AnnotationEditors. Es können mehrere AnnotationEditoren zu verschiedenen Graphbestandteilen gleichzeitig geöffnet werden. Wird ein AnnotationEditor zu einem bereits gestarteten Bestandteil angefordert, wird keine neue Instanz erzeugt, sondern das Fenster der bereits gestarteten Instanz wieder in den Vordergrund gestellt.

Auch beim AnnotationEditor wurde der Funktionsumfang angepasst. Es existiert keine *Beenden*, *Datei öffnen* oder *Datei speichern* Funktion. Es kann nur das Fenster des AnnotationEditors geschlossen werden. Der Content-Baum wird immer automatisch an der entsprechenden Komponente des Prozessmodells gespeichert. Zusätzlich gibt es die Funktionen *Annotation exportieren* und *Annotation importieren*, die im Prinzip den Funktionen *Datei öffnen* und *Datei speichern* entsprechen.

Die Abbildung 4.6 auf der vorherigen Seite zeigt den AnnotationEditor im jETI-Plugin. Dort werden die Zusatzinformationen zur Beschreibung eines Webservices verwendet. Das jETI-Plugin stellt einen Generator zur Verfügung, der aus dem Prozessmodell unter Berücksichtigung dieser Annotationen einen entsprechenden Webservice generiert [KMK+08].

5 Projektbeispiel

Bei der Entwicklung von Softwareprojekten werden die Sourcedateien üblicherweise in Revision-Control-Systemen (RCS) versioniert. Die verschiedenen Entwickler aktualisieren über das RCS ihre lokalen Versionen des Sourcecodes. Dabei gibt es drei wesentliche Funktionen:

Checkout: Mit dieser Funktion wird eine initiale Kopie aller Sourcecode-Dateien eines Repositories in einem Arbeitsverzeichnis angelegt. Beim Checkout müssen eine Benutzerkennung und das Passwort angegeben werden, mit dem der Zugriff auf das Repository autorisiert wird. Beide Angaben werden von Subversion gespeichert, sodass in nachfolgenden Schritten diese nicht mehr angegeben werden müssen.

Update: Mit der Updatefunktion werden die letzten Änderungen aus dem Sourcecode-Repository auf die Sourcedateien im Arbeitsverzeichnis angewendet. Dabei werden in der Regel die Dateien nicht einfach ersetzt, sondern die Änderungen werden zeilen- oder wortweise in die Kopien übernommen. Dies hat den Vorteil, dass eigene Änderungen an einer Sourcedatei, die noch nicht ins Repository übernommen wurden, nicht verloren gehen.

Commit: Mit der Commitfunktion werden die Änderungen der Sourcedateien im Arbeitsverzeichnis in das Softwarerepository übertragen und freigegeben. Andere Entwickler können dann mit der Updatefunktion diese Änderungen in ihre Kopien übernehmen.

Eine goldene Regel bei der Arbeit mit RCS-Systemen ist, dass nur Versionen in das Repository *commited* werden dürfen, die fehlerfrei übersetzt werden können und alle Unittests erfolgreich absolvieren. Anderenfalls würden Fehler durch *Updates* in alle anderen Arbeitskopien reproduziert und so alle Entwickler in ihrer Arbeit behindert. Wie diese Regel mit dem jABC durch einen automatischen *Continuous Integration*-Prozess (CI) kontrolliert werden kann, zeigt dieses Beispiel.

Beispielszenario

Beim dem gewählten Szenario handelt es sich um die reale Entwicklungsumgebung des Lehrstuhls und des jABC-Projektes. Alle Softwareprojekte werden in verschiedenen Repositories eines Subversion-Servers (SVN [Col07]) versioniert. Ein LDAP-Server authentifiziert die Zugriffe auf die SVN-Repositories. Im jABC-Projekt wird Mantis [Man09] als Bugtracker eingesetzt. Dieser webbasierte Bugtracker verfügt über eine zusätzliche Webservice-Schnittstelle, über die Bugmeldungen auch programmgesteuert erzeugt werden können. Alle Teilprojekte des jABCs verfügen über automatisierte Maven-Scripte [Apa07b] zum Kompilieren und Testen. Der SVN-Server bietet die Möglichkeit, vor oder nach verschiedenen Kommandos ein Skript auszuführen. Diese Skripte werden als *Hook Scripts* bezeichnet. Für dieses Beispiel werden wir das *Postcommit Script* nutzen, das nach jedem *Commit* Kommando ausgeführt wird.

Im Folgenden wird ein *Continuous Integration*-Prozess vorgestellt, der nach jedem Commit-Kommando für ein bestimmtes Repository ausgeführt wird. Dieser Prozess lädt alle aktuellen Versionen der Sourcedateien aus dem Repository herunter und prüft dann, ob diese fehlerfrei per Maven compiliert werden können. Meldet Maven einen Fehler, soll eine neue Bugmeldung im Mantis Bugtracker erzeugt werden. Im LDAP-Verzeichnis wird danach festgestellt, ob eine Handynummer für den Benutzer hinterlegt wurde, der das Commit ausgelöst hatte. Liegt eine Handynummer im LDAP vor, wird eine SMS an den betreffenden Benutzer gesendet, die über das Problem informiert. Falls keine Handynummer hinterlegt wurde, wird eine entsprechende E-Mail an den Benutzer gesendet. Sind bei der Übersetzung keine Fehler aufgetreten, werden anschließend alle Unittests aufgerufen. Auch hier wird beim Auftreten von Fehlern eine Bugmeldung erzeugt und der Entwickler benachrichtigt.

SIB-Palette

Für dieses Projekt wurde eine eigene kleine SIB-Palette entwickelt. Neben den SIB-Parametern und Branches wird ein Teil der Konfiguration aus einer Propertydatei entnommen. Die zugehörigen Schlüssel und Werte können als Annotationen im AnnotationEditor gesetzt werden. Das PropertyPlugin speichert diese Werte dann in einer entsprechenden Datei. Die folgende Übersicht entspricht der SIB-Dokumentation.

WorkingFolderSIB		Dieses SIB prüft, ob das Arbeitsverzeichnis existiert und eine Subversion-Steuerdatei (.svn) enthält.
Branch	exists	Das Arbeitsverzeichnis existiert und enthält eine Subversion-Steuerdatei.
	not found	Das Arbeitsverzeichnis fehlt oder enthält keine Subversion-Steuerdatei.
	config error	Der Propertywert WORK_DIR ist nicht gesetzt.
Property	WORK_DIR	Absoluter Pfad des Arbeitsverzeichnisses.
SvnSIB		Dieses SIB führt ein Subversion-Kommando (`svn`) aus.
Parameter	Command	Es kann das Kommando `update` oder `checkout` ausgewählt werden.
Branch	OK	Das `svn`-Kommando wurde fehlerfrei ausgeführt.
	error	Das `svn`-Kommando wurde nicht fehlerfrei ausgeführt.
	config error	Die Propertywerte SVN, SVN_UPDATE oder SVN_CHECKOUT sind nicht gesetzt.
Property	SVN	Das SVN-Binary
	SVN_UPDATE	Die Argumente für das `update`-Kommando.
	SVN_CHECKOUT	Die Argumente für das `checkout`-Kommando.
MvnSIB		Dieses SIB führt ein Maven-Kommando (`mvn`) aus.
Parameter	Command	Es kann das Kommando `clean`, `compile` oder `test` ausgewählt werden.
Branch	OK	Das `mvn`-Kommando wurde fehlerfrei ausgeführt.
	error	Das `mvn`-Kommando wurde nicht fehlerfrei ausgeführt.
	config error	Der Propertywert MVN ist nicht gesetzt.
Property	MVN	Absoluter Pfad des `mvn`-Binaries.
	WORK_DIR	Absoluter Pfad des Arbeitsverzeichnisses.

MantisSIB		Dieses SIB erzeugt eine neue Mantis-Bugmeldung.
Parameter	Summary	Die Zusammenfassung des Bugs.
	Description	Die Beschreibung des Bugs.
Branch	OK	Die Mail wurde abgesetzt.
	config error	Die Propertywerte MNT_SERVER, MNT_USER, MNT_PWD oder MNT_PROJECT sind nicht gesetzt.
Property	MNT_SERVER	IP oder DNS-Name des Mantis-Servers.
	MNT_USER	Mantisbenutzer, mit dem der Bug erzeugt wird.
	MNT_PWD	Passwort des Mantisbenutzers.
	MNT_PROJECT	Projektname, in das der Bug eingestellt werden soll.
LdapSIB		Dieses SIB führt eine LDAP-Suche nach der Handynummer und der E-Mailadresse des Benutzers aus, der vom *Postcommit*-Skript übergeben wurde. Die Handynummer oder E-Mail Adresse werden in den Properties gespeichert.
Branch	mobile	Die Suche hat einen Benutzer gefunden, bei dem eine Handynummer eingetragen wurde.
	mail	Die Suche hat einen Benutzer gefunden, bei dem keine Handynummer aber eine Mailadresse eingetragen wurde.
	not found	Die Suche hat keinen Benutzer gefunden.
	config error	Die Propertywerte LDAP_SERVER oder LDAP_QUERY sind nicht gesetzt.
Property	LDAP_SERVER	IP oder DNS-Name des LDAP-Servers.
	LDAP_QUERY	LDAP-Query für die Benutzersuche.
	MOBILE	Die Handynummer.
	MAIL_TO	Die E-Mailadresse.

MailSIB		Dieses SIB versendet an die festgelegte Empfängeradresse eine E-Mail mit entsprechendem Betreff.
Parameter	Subject	Der Betreff der E-Mail.
	Message	Die Mitteilung der E-Mail.
Branch	OK	Die Mail wurde abgesetzt.
	config error	Die Propertywerte SMTP_SERVER, MAIL_FROM oder MAIL_TO sind nicht gesetzt.
Property	SMTP_SERVER	IP oder DNS-Name des SMTP-Servers.
	MAIL_FROM	E-Mail Absenderadresse.
	MAIL_TO	E-Mail Empfängeradresse.
SmsSIB		Dieses SIB versendet eine SMS mit entsprechender Mitteilung.
Parameter	Message	Die Mitteilung der SMS.
Branch	OK	Die SMS wurde abgesetzt.
	config error	Die Propertywerte SMS_SERVER oder MOBILE sind nicht gesetzt.
Property	SMS_SERVER	IP oder DNS-Name des Servers.
	MOBILE	Handynummer des Empfängers der SMS.
Log4jSIB		Dieses SIB versendet eine Logmeldung über die Bibliothek Log4j.
Parameter	Level	Es kann zwischen den Leveln `info`, `warning`, `error` und `fatal` gewählt werden.
	Message	Die Mitteilung der Log-Meldung.
Branch	OK	Die Meldung wurde abgesetzt.

Prozessmodell

Aus der im vorigen Abschnitt vorgestellten SIB-Palette wurde der folgende Prozess modelliert, der nach jedem *Commit* ausgeführt werden soll. Der Prozess in Abbildung 5.1 auf der nächsten Seite gliedert sich in vier wesentliche Bereiche:

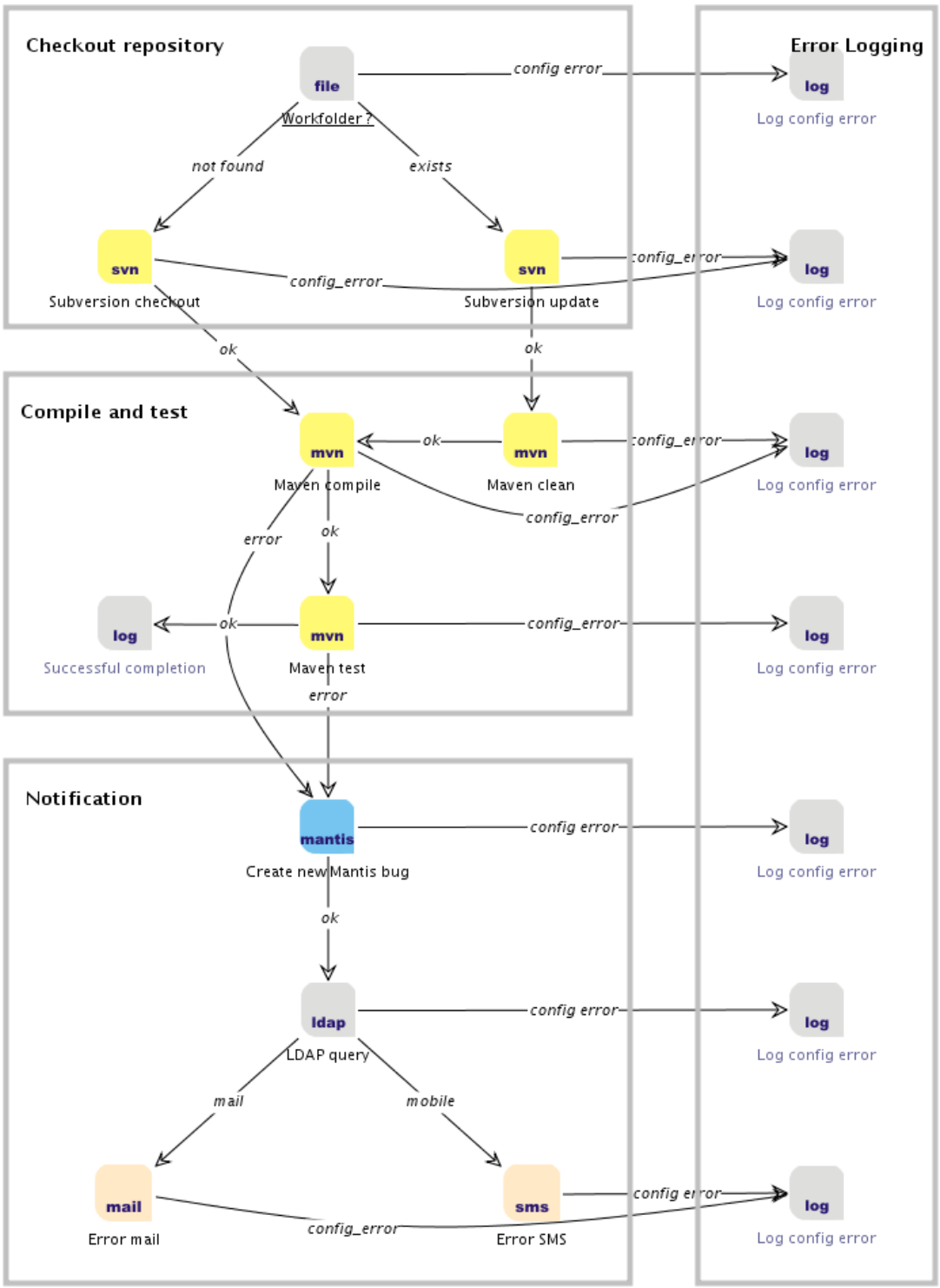

Abbildung 5.1: jABC-Modell des CI-Prozesses

Checkout repository: In diesem Abschnitt wird die lokale Arbeitskopie auf den letzten Stand des Repositories aktualisiert. Beim ersten Aufruf wird das Subversion *checkout*-Kommando ausgeführt. Bei jedem weiteren Aufruf wird das Repository nur noch mit dem *update*-Kommando aktualisiert. Durch ein Update werden Ressourcen gespart und die Ausführungszeit ist in der Regel deutlich kürzer. Da auf der lokalen Arbeitskopie kein Entwickler arbeitet, ist dies auch problemlos möglich. Anderenfalls müsste man bei jeder Ausführung des CI-Prozesses vor dem *checkout*-Befehl das Arbeitsverzeichnis komplett löschen.

Compile und test: In diesem Teil des Prozesses wird die eigentliche Überprüfung der aktuellen Repositoryversion vorgenommen. Durch die automatisierten Maven-Buildskripte ist dies sehr einfach möglich. Maven überprüft die von den Entwicklern definierten Abhängigkeiten zwischen den Sourcedateien und den benötigten Bibliotheken und lädt diese bei Bedarf automatisch herunter. Auch der Klassenpfad des Java-Compilers und der Testklassen werden von Maven automatisch zusammengestellt. Durch verschiedene *Targets* werden unterschiedliche Teile des Buildprozesses angestoßen. In diesem Prozess werden folgende Maven-Targets verwendet:

maven clean: Das *Clean-Target* löscht alle Reste eines vorhergehenden Buildvorgangs. Danach liegt eine *saubere* Kopie des Arbeitsverzeichnisses vor.

maven compile: Mit dem *Compile-Target* werden alle Sourcedateien vom Compiler übersetzt. Vorher werden bei Bedarf benötigte Bibliotheken aktualisiert oder heruntergeladen. Meldet der Java-Compiler einen Syntaxfehler, bricht Maven mit einer entsprechenden Meldung den Buildvorgang ab.

maven test: Das *Test-Target* setzt voraus, dass das *Compile-Target* vorher ausgeführt wurde. Falls dies nicht der Fall ist, wird das *Compile-Target* automatisch aufgerufen. Erst wenn dieser Schritt erfolgreich beendet wurde, werden die Testklassen übersetzt und aufgerufen. Auch hier meldet Maven einen Fehler, falls ein Test scheitert.

Notification: Dieser Abschnitt benachrichtigt den Entwickler, falls bei der Übersetzung oder beim Test des Repositories ein Fehler aufgetreten ist. Zuerst wird über die Webservice-Schnittstelle von Mantis eine neue Bugmeldung angelegt (siehe Abbildung 5.2 auf der nächsten Seite). Die Meldung enthält die notwendigen Angaben, um welchen Fehler es sich handelt. Danach wird der Entwickler zusätzlich direkt informiert. Im LDAP-Verzeichnis wird dazu die Handynummer oder die E-Mailadresse des Entwicklers nachgeschlagen, der als *Commiter* von Subversion übergeben wurde. Falls eine Handynummer vorliegt, wird die

Benachrichtigung per SMS bevorzugt (siehe Abbildung 5.3 auf Seite 77).

Logging: In diesem Bereich werden alle Konfigurationsfehler, die im Prozess auftreten können, an die Logging-Bibliothek log4j [Apa07a] übergeben. Diese Meldungen werden dann abhängig von der Log4j-Konfiguration in entsprechenden Logdateien abgelegt.

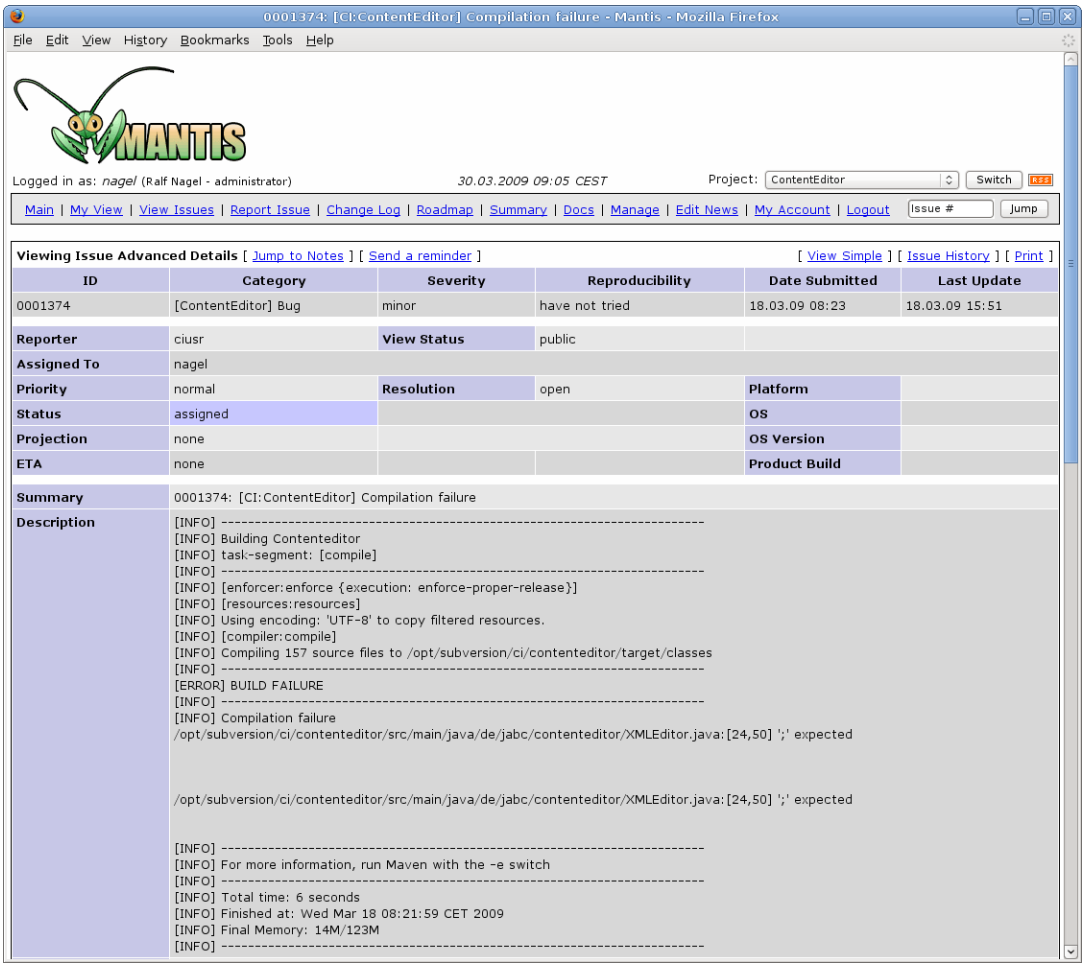

Abbildung 5.2: CI Mantis-Bugmeldung

Die nachfolgenden Tabellen entsprechen den SIB-Annotationen des Prozessmodells.

Workfolder ?	
WorkingFolderSIB	Es wird geprüft, ob zu einem früheren Zeitpunkt bereits ein *svn checkout* ausgeführt wurde. Das Arbeitsverzeichnis wird über das Property WORK_DIR konfiguriert. Der SVN-Benutzer benötigt Lese- und Schreibrechte, da dieser Prozess als SVN-Subprozess ausgeführt wird.
WORK_DIR	/opt/subversion/ci

Subversion checkout	
SvnSIB	Es wird eine aktuelle Kopie des Repositories angelegt. Username, Password und Repositorypfad müssen als Argumente des *checkout*-Kommandos angegeben werden. Bei Verwendung einer verschlüsselten Verbindung muss voher ein passendes Zertifikat installiert werden.
SVN_CHECKOUT	checkout –username ciusr –password ******** https://ls5svn/svn/jabc-core/trunk/contenteditor $WORK_DIR

Subversion update	
SvnSIB	Das Arbeitsverzeichnis wird mit der letzten Version aus dem Repository aktualisiert. Vorher muss das Arbeitsverzeichnis einmal per *svn checkout* angelegt werden.
SVN	/usr/bin/svn
SVN_UPDATE	update $WORK_DIR

Maven clean	
MvnSIB	Das Maven *clean* Kommando löscht alle Reste von vorhergehenden Übersetzungsprozessen im `target` Verzeichnis.
MVN	/usr/bin/mvn

Maven compile	
MvnSIB	Das Repository wird übersetzt (*mvn compile*).

Maven test	
MvnSIB	Es werden alle Tests ausgeführt (*mvn test*).

LDAP query	
LdapSIB	Dieses SIB sucht im LS5-LDAP-Baum nach dem Commit-Benutzer. Es werden die E-Mailadresse und die Handynummer (Mobil) ausgelesen. Die E-Mailadresse ist im LS5-LDAP immer vorhanden. Die Handynummer ist dagegen optional.
LDAP_SERVER	ls5ldap1.cs.tu-dortmund.de
LDAP_BASE	ou=people,dc=ls-5,dc=de
LDAP_QUERY	(&(objectclass=inetOrgPerson)(uid=$1))

Create new Mantis bug	
MantisSIB	Erzeugt eine neue Bugmeldung im ContentEditor-Projekt. Das *Assignment* wird in Mantis durch die Bug-Kategorie festgelegt.
MNT_SERVER	leo.cs.tu-dortmund.de
MNT_USER	ociusr
MNT_PWD	********
MNT_PROJECT	ContentEditor

Send mail	
MailSIB	Sendet eine Mail über den SMTP Host der IRB (waldorf).
SMTP_SERVER	waldorf.cs.tu-dortmund.de
MAIL_FROM	no-reply@ls5.cs.tu-dortmund.de

Send SMS	
SmsSIB	Sendet eine SMS über ein kostenpflichtiges Mail2SMS-Gateway.
SMS_SERVER	smtp.arcor.de

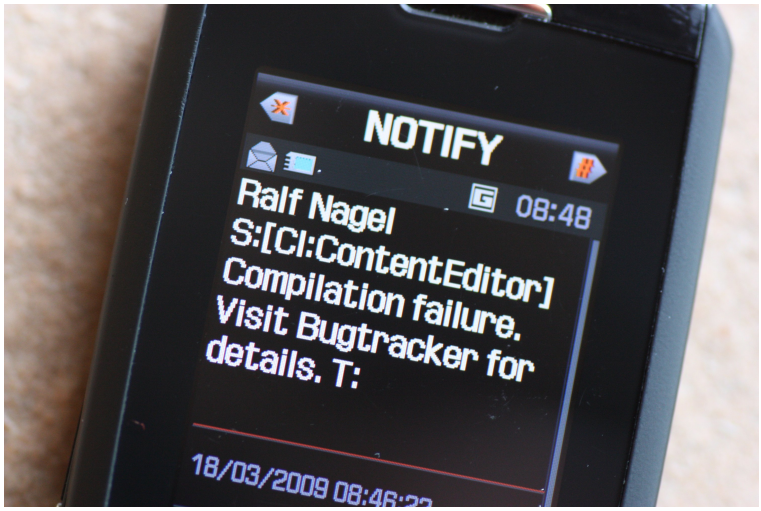

Abbildung 5.3: SMS mit CI-Bugmeldung

Properties

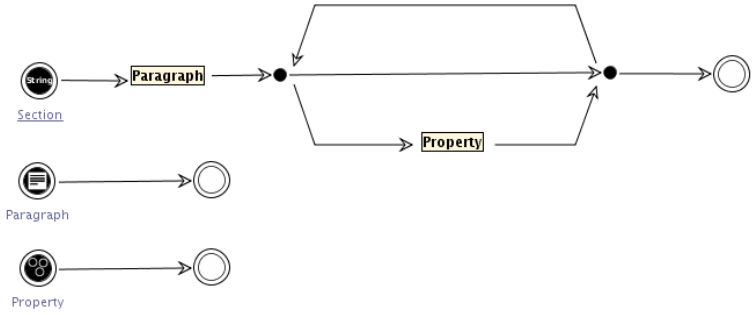

Abbildung 5.4: Annotation-Grammatik des CI-Prozesses

In diesem Beispiel werden die technischen und serverabhängigen Prozessparameter über Propertywerte konfiguriert. Die Werte werden zur Laufzeit aus einer entsprechenden Propertydatei ausgelesen. Wird der CI-Prozess auf einen anderen Server installiert, müssen nur die Werte der Propertydatei angepasst werden. Zur Demonstration werden die initialen Propertywerte direkt im Prozessmodell hinterlegt. Dazu wurde eine sehr einfache Grammatik für den AnnotationEditor entwickelt. Jedes SIB soll mit genau einem Paragraph-Knoten und beliebig vielen Property-Knoten dokumentiert werden können. Der Paragraph-Knoten ist ein RichTextSIB und stellt einen HTML-Editor für die Fließtextdokumentation des SIBs bereit. Der Property-Knoten ist ein CompoundSIB, das intern aus zwei

TextSIBs zusammengesetzt ist. Das erste TextSIB speichert den Schlüssel eines Propertys, das zweite TextSIB nimmt den entsprechenden Wert auf. Die in der Abbildung 5.4 auf der vorherigen Seite gezeigte Grammatik wurde im jABC zur Dokumentation des SIBs festgelegt.

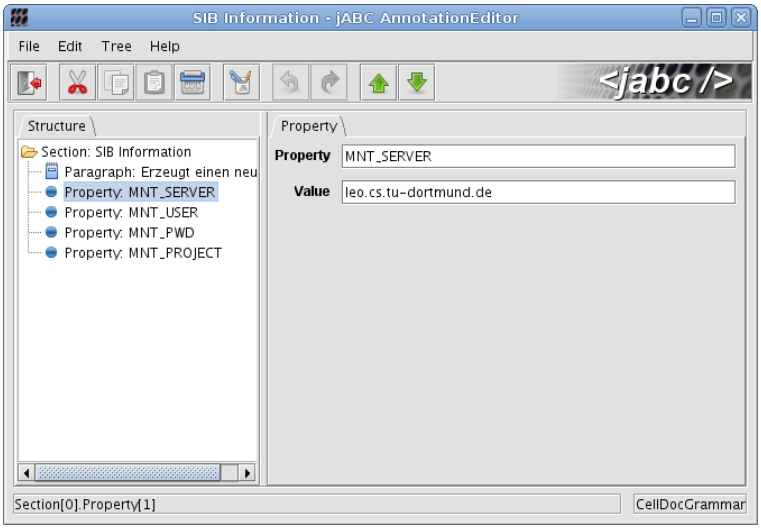

Abbildung 5.5: Annotationen des *LDAP query*-SIBs

Mit dem PropertyPlugin können die Propertywerte, die im AnnotationEditor festgelegt wurden, in eine Propertydatei geschrieben werden (vgl. Abbildung 5.5). Das PropertyPlugin erweitert das jABC nur um einen Eintrag im Pluginmenü. bei aufruf des Plugins wird über einen Filechooser der Dateiname einer Propertydatei bestimmt (`actionPerformed()`) und dann die Werte aus dem Prozessmodell dorthin übertragen (`generate()`). Das Plugin wurde auf ein Minimum beschränkt, um den Rahmen dieser Arbeit nicht zu sprengen.

PropertyPlugin.java

```
package de.jabc.plugin.propertyplugin;

import java.awt.event.ActionEvent;
import ... // some imports deleted

// This plugin will create a property file from
// the annotations in the current model.
public class PropertyPlugin implements Plugin {

  // Display name of this Plugin
  public String getPluginName() {
```

```
  return "Property-Plugin";
}

// Register the menu action in the plugin menu
public void start() {
  MenuHandler menuHandler = MenuHandler.getInstance();
  JMenu pluginMenu = menuHandler.getMenu(MENU_PLUGINS);
  menuHandler.addAction(pluginMenu,
        new CreatePropertiesAction());
}

// This method fills the property object and stores it
public void generate(File file) throws Exception {
  Properties props = new Properties();
  if (file.exists())
    // if the file exists load the previous values
    props.load(new FileInputStream(file));
  SIBGraphCell[] cells;
  cells = ABCFrame.getSIBGraphModel().getSIBGraphCells();
  // Outer loop: visit all SIBs
  for (int i = 0; i < cells.length; i++) {
    SIB sib = cells[i].getSIB();
    // Get the AnnotationEditor content
    Content content = (Content) sib.getUserObject(ABC_CE);
    if (content == null)
      continue;
    // Find all 'Property' nodes in the content tree
    ContentEditorNode[] node =
          content.find("Property", NODENAME);
    // Inner loop: copy all property values for this SIB
    for (int j = 0; j < node.length; j++) {
      // the compound node is stored as a map
      Map<String, String> value;
      value = (Map<String, String>) node[j].getValue();
      props.put(value.get("Property"), value.get("Value"));
    }
  }
  props.store(new FileOutputStream(file), null);
}

// Action class for the Swing menuentry
class CreatePropertiesAction extends DefaultAbstractAction {

  public CreatePropertiesAction() {
```

```
      putValue(Action.NAME, "Create Propertyfile");
    }

    public void actionPerformed(ActionEvent event) {

      // Create a file chooser
      final JFileChooser fc = new JFileChooser();
      // Select the filename for the property file
      if (fc.showSaveDialog(null) == APPROVE_OPTION)
        try {
          generate(fc.getSelectedFile());
        } catch (Exception e) {
          e.printStackTrace();
        }
      super.actionPerformed(event);
    }
  }
}
```

Eine vom PropertyPlugin erzeugte Datei sieht dann in etwa wie folgt aus:

ci.properties

```
#Created by Property-Plugin
#Wed Mar 18 10:36:59 CET 2009
LDAP_BASE=ou\=people,dc\=ls-5,dc\=de
LDAP_QUERY=(\\&(objectclass\=inetOrgPerson)(uid\=$1))
LDAP_SERVER=ls5ldap1.cs.tu-dortmund.de
MAIL_FROM=no-reply@ls5.cs.tu-dortmund.de
MNT_PROJECT=ContentEditor
MNT_PWD=********
MNT_SERVER=leo.cs.tu-dortmund.de
MNT_USER=ociusr
MVN=/usr/bin/mvn
SMS_SERVER=smtp.arcor.de
SMTP_SERVER=waldorf.cs.tu-dortmund.de
SVN_CHECKOUT=checkout --username ciusr --password ******** \
  https\://ls5svn/svn/jabc-core/trunk/contenteditor $WORK_DIR
SVN_UPDATE=update $WORK_DIR
SVN=/usr/bin/svn
WORK_DIR=/opt/subversion/ci
```

Verifikation

Um die Korrektheit des Prozessmodells zu gewährleisten, werden sowohl lokale als auch globale Eigenschaften durch das jABC automatisch geprüft. Die Prüfung von lokalen Eigenschaften übernimmt das LocalChecker-Plugin, die globalen Eigenschaften werden mit dem GEAR Modelchecker verifiziert.

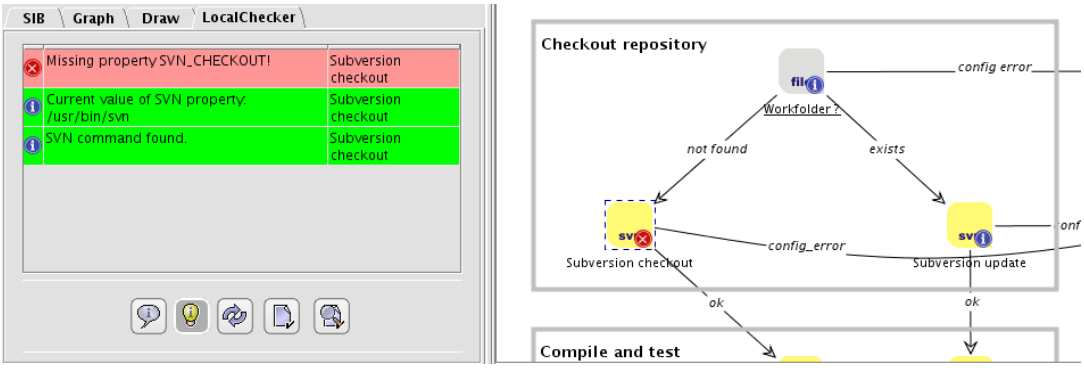

Abbildung 5.6: LocalCheck-Meldungen eines SvnSIBs

Alle SIBs des Continuous-Integration-Modells verfügen über LocalCheck-Code. Da die Parameter der SIB-Palette so konstruiert wurden, dass keine fehlerhaften Eingaben möglich sind (im Gegensatz zum MailSIB im Abschnitt 3.8 auf Seite 42), müssen die SIB-Parameter nicht durch den LocalChecker geprüft werden. Alle technischen Parameter wurden in diesem Beispiel in eine Propertydatei ausgelagert. Für die Ausführung im Tracer müssen diese Werte korrekt gesetzt sein. Daher werden in den SIBs die aktuellen Werte der Propertydatei ausgelesen und soweit möglich geprüft, ohne den eigentlichen Service auszuführen.

Als Beispiel wird hier der Localcheck-Code des *SvnSIBs* vorgestellt. Durch diesen Code werden folgende Eigenschaften geprüft:

- Ist das SVN-Property gesetzt?
- Ist das vom SVN-Property referenzierte Kommando ausführbar?
- Sind die SVN-Argumente in Abhängigkeit vom ausgewählten `command`-Parameter gesetzt ?

Zur Illustration geben die Prüfungen in diesem Beispiel auch bei erfolgreicher Prüfung eine Meldung in der Kategorie `Info` aus. Alle Fehlermeldungen hingegen werden als `Error` klassifiziert.

`checkSIB()`-Methode im SvnSIB.java

```
// localcheck code
public void checkSIB(SIB sib) {
  // Check 1) Is the SVN property defined ?
  String svn = CIProperties.getSvn();
  if (svn == null)
    SIBUtilities.addError(sib, "Missing property SVN!");
  else
    SIBUtilities.addInfo(sib,
      "Current value of SVN property: " + svn);
  // Check 2) Is the SVN binary executable ?
  if (!new File(svn).canExecute())
    SIBUtilities.addError(sib,
      "SVN command is not executable! " + svn);
  else
    SIBUtilities.addInfo(sib, "SVN command found.");
  // Check 2) Are the SVN command arguments defined ?
  String svnargs = "SVN_" + command.getSelectedValue();
  CIProperties props = CIProperties.getInstance();
  if (props.getProperty(svnargs) == null)
    SIBUtilities.addError(sib,
      "Missing property " + svnargs + "!");
  else
    SIBUtilities.addInfo(sib,
      "Current value of " + svnargs + " property: " +
        props.getProperty(svnargs));
}
```

Mit globalen Eigenschaften kann die Struktur des Prozessmodells geprüft werden. Dies ist insbesondere für die kontinuierliche Weiterentwicklung des Prozesses von Bedeutung. Durch die automatische Verifikation globaler Eigenschaften können gewisse Details nicht übersehen werden. In diesem Beispiel werden nur sehr einfache Aussagen über das Modell geprüft. Die Formeln werden dazu im GEAR-Formularmanager hinterlegt. Vorher wurden einige SIBs mit dem AP-Manager mit *atomaren Propositionen* markiert. Abbildung 5.7 auf der nächsten Seite zeigt die Arbeit mit GEAR.

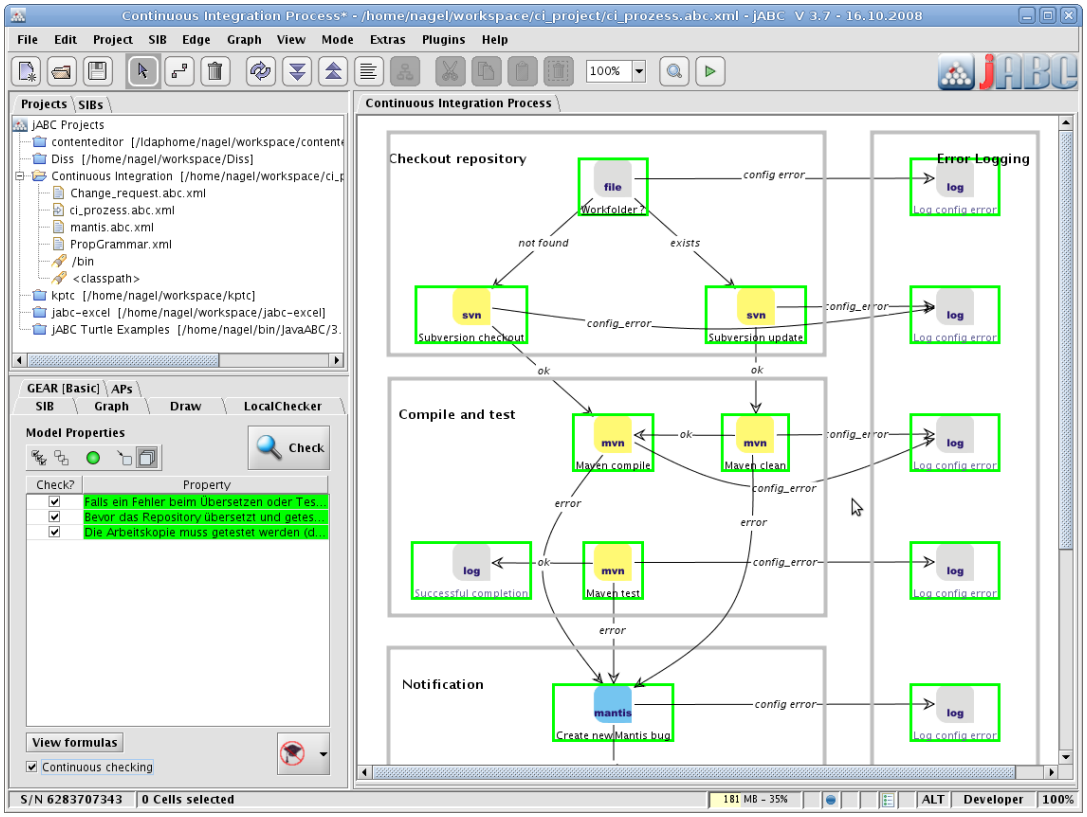

Abbildung 5.7: Model Checking mit GEAR

GEAR-Formel	Start => AF((SVN | Error) & AF(MVN | Error))
Beschreibung	Bevor das Repository übersetzt und getestet wird, muss eine Arbeitskopie ausgecheckt oder aktualisiert werden.
GEAR-Formel	Start => AF(MVNTest | Error)
Beschreibung	Die Arbeitskopie muss getestet werden (dies schließt den Übersetzungsvorgang automatisch mit ein).
GEAR-Formel	MVN => <error>AF(Message | Error)
Beschreibung	Falls ein Fehler beim Übersetzen oder Testen gefunden wird, muss der Entwickler per Mail oder SMS benachrichtigt werden.

Hierarchie

Der *Continuous Integration*-Prozess ist auch ein Bestandteil eines übergeordneten Prozesses. In diesem Beispiel verwenden wir dazu einen abstrakten *Change Request*-Prozess (siehe Abbildung 5.8). Solche abstrakten Prozesse werden auch als *Domänenmodelle* bezeichnet. Die Prozesshierarchie kann im jABC beliebig erweitert werden. Neben dem *Change Request*-Prozess könnte es ebenso einen *New feature*-Prozess und viele weitere Prozesse geben.

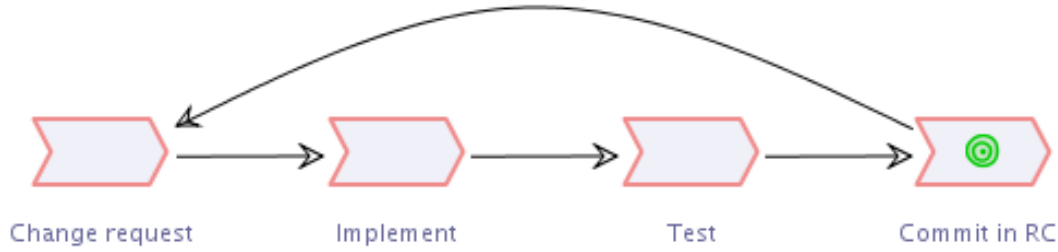

Abbildung 5.8: Übergeordneter *Change Request*-Prozess

Analog zu einem übergeordneten Prozess können auch einzelne Prozessschritte weiter verfeinert werden. Zum Erzeugen einer neuen Bugmeldung müssen über die Mantis-Webserviceschnittstelle verschiedene Aufrufe nacheinander erfolgen. Dies kann ebenfall als Prozess modelliert werden. Solche SIBs, die bereits sehr nah an der eigentlichen (Java-) Implementierung orientiert sind, werden im jABC als *MicroSIBs* bezeichnet (siehe Abbildung 5.9).

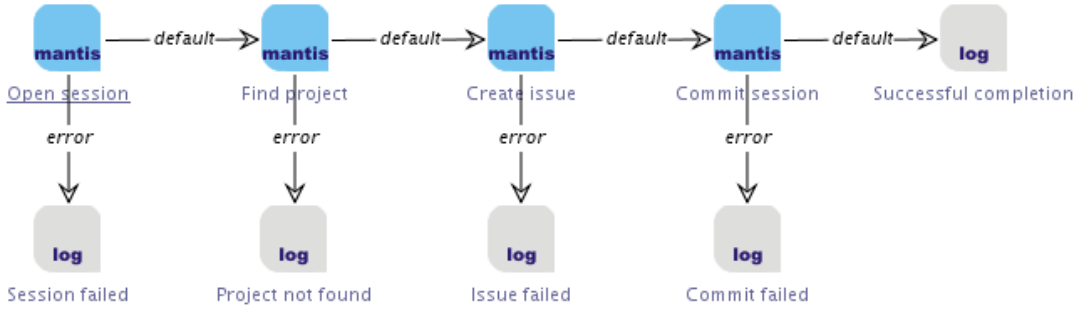

Abbildung 5.9: Prozessverfeinerung: Mantis-Fehlermeldung erzeugen

Subversion-Integration

Nachdem alle Teile des Continuous-Integration Prozesses vorliegen, muss der Prozess noch in den Subversion-Server integriert werden. In der Testphase wird der Prozess im Tracer interaktiv ausgeführt, bis alle Fehler und Probleme beseitigt wurden. Vorher wurde mit dem PropertyPlugin ein passende Propertydatei angelegt. Die Abbildung 5.10 zeigt den Tracer bei der Arbeit.

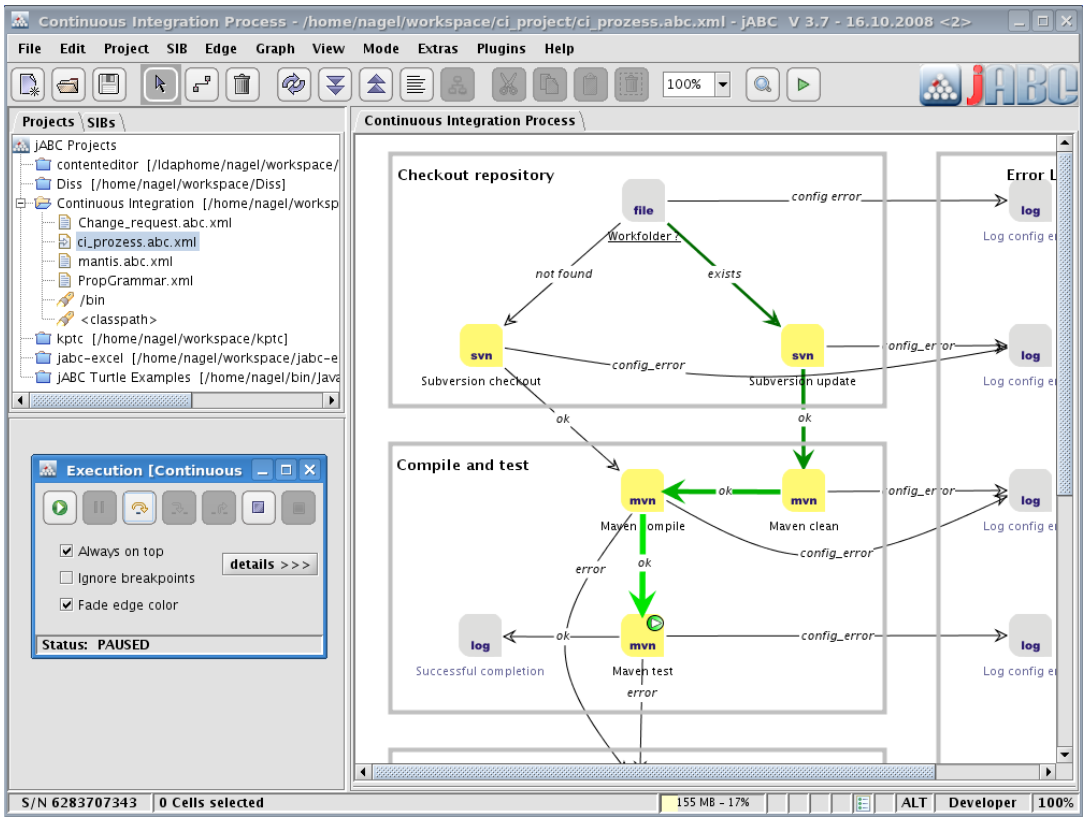

Abbildung 5.10: Tracerausführung des CI-Prozesses

Zur Integration in den Subversion-Server nutzen wir das *Postcommit Script*. Dieses Skript wird vom Subversion-Server nach jedem Commit-Befehl ausgeführt. Das Skript startet dazu eine einfache Java-Klasse, die den eigentlichen Prozess vom Tracer ausführen lässt.

post-commit.sh

```
#!/bin/sh

COMMITTER=$1
JAVA_HOME="/usr/lib/jvm/java-6-sun"; export JAVA_HOME
CLASSPATH="/opt/subversion/ci/bin:..."

java -cp $CLASSPATH de.jabc.ci.Launcher $COMMITTER
```

Schließlich übernimmt diese Java-Klasse die Aufgabe, den CI-Prozess mit dem Tracer-Plugin zu starten:

Launcher.java

```
package de.jabc.ci;

import de.metaframe.jabc.framework.execution.*;
import ... // some imports deleted

// Launch the CI process
public class Launcher {

  public static void main(String[] args) throws Exception {
    if (args.length < 1) {
      System.err.println(
        "usage: java de.jabc.ci.Launcher <commiter>");
      System.exit(-1);
    }
    // Load the processmodel
    ClassPathSIBLoader sibloader = new ClassPathSIBLoader();
    SIBGraphModel model = SIBGraphModelFactory.read(sibloader,
      new File("ci-process.xml"));
    // Start the tracer
    ExecutionController ec = new ExecutionController(model);
    ec.getGlobalExecutionContext().put("COMMITTER", args[0]);
    ec.execute();
  }
}
```

6 Verwandte Arbeiten

In diesem Kapitel werden die gängigen alternativen Methoden zur grafischen Prozessdefinition insbesondere im Hinblick auf die im Kapitel 1.1 auf Seite 3 aufgestellten grundlegenden Anforderungen beleuchtet. Der Fokus liegt hierbei auf allgemeinen Ansätzen, die von der jeweiligen Anwendungsdomäne unabhängig sind. Anwendungsspezifische Ansätze, wie beispielsweise der in [NM08] beschriebene zur chemischen Verfahrenstechnik, werden hier nicht betrachtet.

Der Abschnitt IDE steht stellvertretend für hybride Methoden, bei denen mehrere Bestandteile in einem Rahmenwerk lose zusammengeführt werden. Dieser Trend zu hybriden, auf den jeweiligen Anwendungsfall und Entwickler zugeschnittenen Entwicklungsumgebungen scheint sich in der letzten Zeit zunehmend zu etablieren. Sie sind richtungsweisend für Entwicklungsysteme, die sich an unterschiedliche Rahmenbedinungen anpassen lassen.

BPEL

Die *Business Process Execution Language* (BPEL) ist ein bekannter, im Jahr 2002 von IBM, Bea und Microsoft entwickelter XML-Dialekt zur Beschreibung von Geschäftsprozessen [A+03]. BPEL-Prozesse setzen sich aus Webservice-Aufrufen zusammen und werden selbst als Webservice verwendet (*Webservice-Komposition*) [WCL+05]. Für das Erstellen von BPEL-Dokumenten existieren verschiedene grafische Editoren von unterschiedlichen Herstellern wie z.B. das Netbeans Enterprise Pack [SUN09b] (siehe Abbildung 6.1 auf der nächsten Seite). Als grafische Notation setzen solche Editoren in der Regel die *Business Process Modeling Notation* (BPMN) ein [WM08].

Zur Ausführung eines BPEL-Prozesses wird eine passende *BPEL-Engine* benötigt. Häufig sind BPEL-Editoren und Engines verschiedener Hersteller untereinander nicht kompatibel, sodass durch die Wahl eines Editors auch die zugehörige Engine verwendet werden muss oder umgekehrt. Die Einsetzbarkeit von BPEL für allgemeine Geschäftsprozesse wird dadurch eingeschränkt, dass ausschließlich Webservices verwendet werden können. Reale Abläufe in Geschäftsprozessen

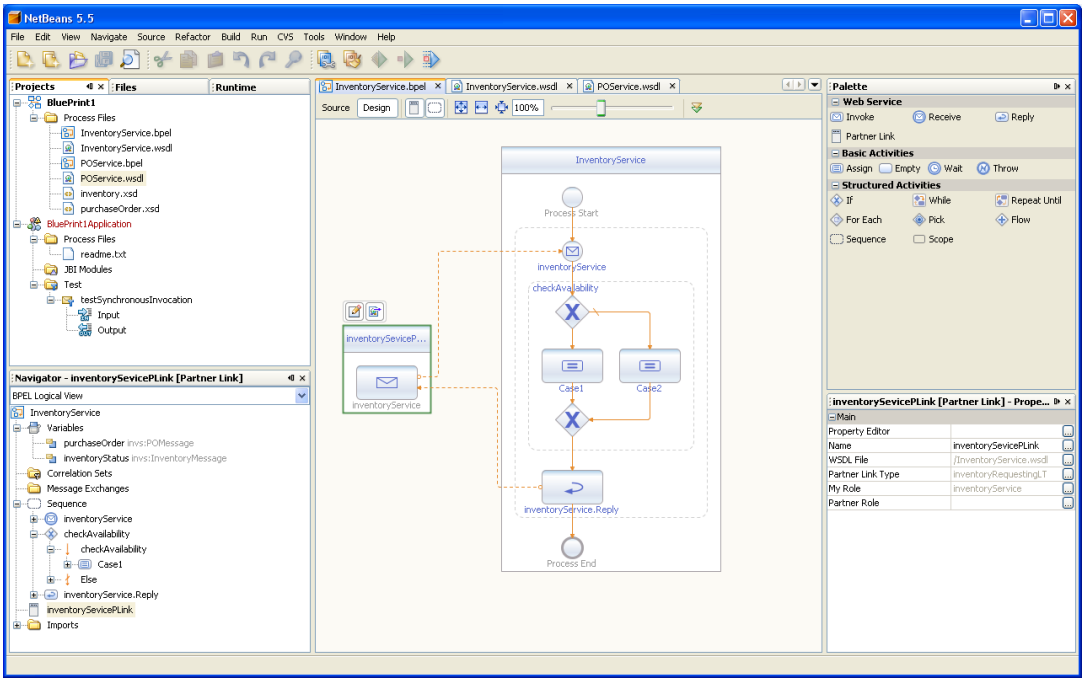

Abbildung 6.1: BPEL-Designer in Netbeans 5.5 (aus: [Rom07])

können auf diese Weise nur über unglückliche Umwege abgebildet werden. Als Beispiel könnte in einem *Supply Chain*-Prozess ein Teilschritt daraus bestehen, dass ein LKW eine bestimmte Ware vom Zwischenlager zur Produktion bringt. Bei dieser realen Aktion handelt es sich offensichtlich nicht um einen Webservice und ist daher in BPEL nicht direkt darstellbar (vgl. Anforderung G5: Universalität). Auch die Interaktion mit den Anwendern kann in BPEL nicht abgebildet werden. Dazu wurden in den letzten Jahren verschiedene Erweiterungen vorgeschlagen. Ein Ansatz ist *BPEL for Java technology* (BPELj) [IBM07]. In BPELj kann Java Code als sogenannte *Java Snipptes* in die BPEL-Beschreibung eingebettet werden. Hier stellt sich natürlich die Frage, ob Java-Fragmente in XML-Dokumenten noch sinnvoll sind. Eine andere Alternative ist *BPEL4People*, das rollenbasierte menschliche Interaktionen zusammen mit Webservices in BPEL kombiniert.

BPEL ist ein Hilfsmittel für die technische Realisierung von bestimmten Software-Prozessen. Die Möglichkeit, *Long Running Transactions* und *Compensation* zu beschreiben, kann bei der Umsetzung von Prozessen genutzt werden. Allerdings eignet sich BPEL nicht dazu, von einem *Business Developer* eingesetzt zu werden (vgl. Anforderung G1: Intuitivität). Es wird eine hohes Maß an technischem Wissen über die verwendeten Webservices vorausgesetzt. In BPEL

können nur Webservices verwendet werden, für die bereits eine Beschreibung in der *Web Services Description Language* (WSDL) [W3C01] vorliegt. In frühen Phasen einer Prozessdefinition, wenn die unterstützenden Systeme noch gar nicht existieren, kann auch dies zu Problemen führen.

ARIS

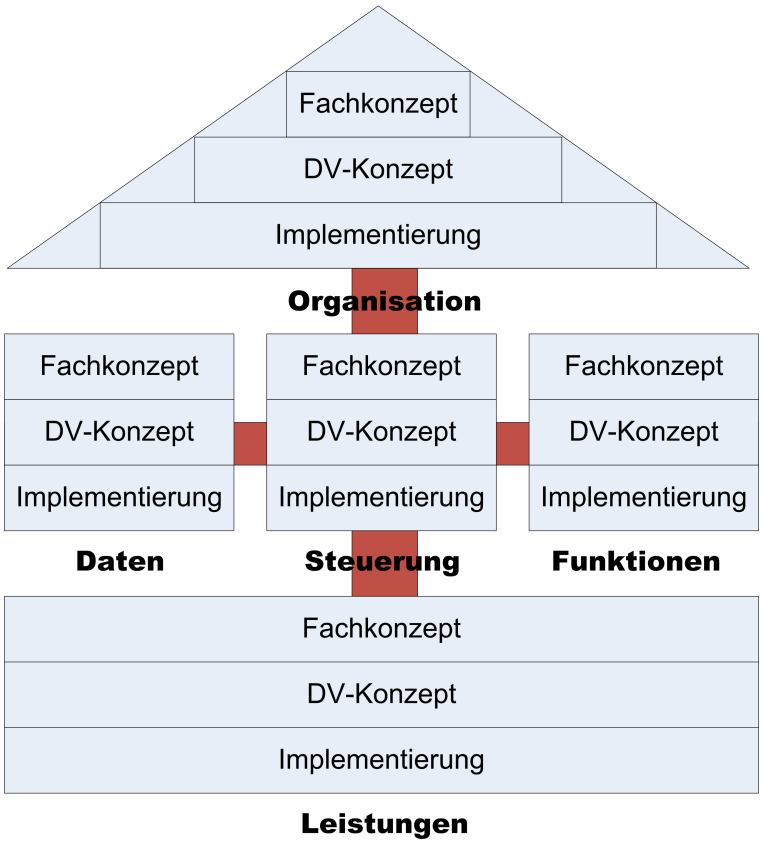

Abbildung 6.2: Aris-Haus nach A. W. Scheer

Das Aris-Konzept (*Architektur integrierter Informationssysteme*) und die dazu passende Software (*ARIS-Toolset*) wird von Prof. August-Wilhelm Scheer seit 1993 entwickelt [Fra08]. Ein Softwaresystem wird danach in fünf unterschiedlichen Sichten definiert: *Funktions-, Daten-, Organisations-, Steuerungs-* und der *Leistungssicht* (vgl. Abbildung 6.2). Jede dieser Sichten wird wiederum in drei Phasen konzipiert: dem *Fachkonzept*, dem *DV-Konzept* und der *Implementierung*. Zum Teil werden für die unterschiedlichen Phasen der genannten Sichten

unterschiedliche grafische Notationen verwendet. Weite Verbreitung hat die *erweitere Ereignisgesteuerte Prozessketten*-Notation (eEPK) gefunden, die in der Steuerungssicht verwendet wird und das Zusammenspiel der anderen Sichten koordiniert, obwohl diese Semantik konzeptionelle Schwächen aufweist.

Wesentlicher Einsatzbereich des ARIS-Toolsets ist das Customizing von SAP-Systemen. Dazu werden die Prozesse entsprechend den Anforderungen in ARIS modelliert. Das nachfolgende Customizing ist dann allerdings eine manuelle Übertragung dieser Modelle in eine entsprechende SAP-Konfiguration. Dabei müssen individuelle Entscheidungen getroffen werden, wie jedes Prozessdetail in SAP umgesetzt wird (vgl. Anforderung G2: Agilität). Mit dem *ARIS UML Designer* wurde das ARIS-Konzept an UML angebunden, um die Modellierung der Geschäftsprozesse mit der Softwarerealisation zu koppeln.

Das ARIS-Toolset besteht aus einer weitverzweigten Palette von verschiedenen Anwendungen, die jeweils nur eine spezielle Aufgabe im ARIS-Konzept übernehmen. Die folgende Tabelle gibt eine Übersicht über die einzelnen Produkte, ohne im Detail auf die jeweilige Aufgabe näher einzugehen.

ARIS Design Platform	**ARIS Implementation Platform**
ARIS Business Architect	ARIS Business Architect for SAP
ARIS Business Designer	ARIS BI Modeler
ARIS Business Publisher	ARIS SOA Architect
ARIS IT Architect	ARIS Business Rules Designer
ARIS IT Inventory	ARIS UML Designer
ARIS Defense Solution	ARIS for Interstage BPM
ARIS Quality Management Scout	ARIS for Microsoft BizTalk
ARIS Archimate Modeler	
ARIS Strategy Platform	**ARIS Controlling Platform**
ARIS Business Optimizer	ARIS Process Performance Manager
ARIS BSC	ARIS Risk & Compliance Manager
ARIS Business Simulator	ARIS Process Event Monitor
ARIS Six Sigma	ARIS Performance Dashboard

Viele Ansatzpunkte aus dem ARIS-Konzept sind bei der Realisierung des jABCs ebenfalls untersucht worden. Ein Schwachpunkt ist die Schwierigkeit der konsistenten Modellierung. Durch die insgesamt 15 unterschiedlichen Phasen kann es bei agilen Systemen leicht zu Abweichungen zwischen Fachkonzept und der

Implementierung kommen. Ein Anwender muss sich erst intensiv in das Modellierungskonzept und die Werkzeugpalette von ARIS einarbeiten, bis er die ersten Modelle erstellen kann (vgl. Anforderung G1: Intuitivität).

ARIS-Modelle müssen von Entwicklern in entsprechende Softwarekomponenten umgesetzt werden. Durch permanente Anforderungsänderungen an Geschäftsprozesse und die daraus resultierende Weiterentwicklung der Modelle müssen Änderungen konsistent in die Implementierung übernommen werden. Eine manuelle Übernahme der Änderungen durch Entwickler ist dabei aufwändig und fehleranfällig. Zusätzlich besteht die Gefahr, dass Änderungen direkt am Code vorgenommen werden, ohne das dazugehörige Modell anzupassen (vgl. Anforderung G3: Konsistenz). Das *Round Trip*-Problem kann nur von Systemen befriedigend gelöst werden, die ein Modell automatisch ohne manuellen Eingriff in Sourcecode transformieren (vgl. Kapitel 3.7.4 auf Seite 40).

UML

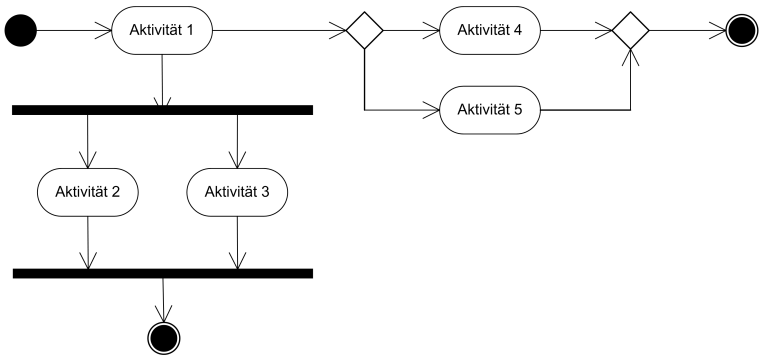

Abbildung 6.3: UML-Aktivitätsdiagramm

Die *Unified Modeling Language* (UML) [Fow04] ist eine grafische Notation für die Spezifikation von Softwaresystemen und wird von der Object Management Group (OMG) definiert. UML definiert eine Reihe von verschiedenen Diagrammtypen mit unterschiedlichen Semantiken. Bekannte Typen sind *Klassen-, Use Case-* oder *Sequenz-Diagramme*. Je nach Diagrammtyp sind verschiedene Zeichenelemente mit unterschiedlichen Bedeutungen zulässig. Diese Elemente können zusätzlich beschriftet und mit Kanten verbunden werden. UML wird im *Rational Unified Process* (RUP) [Hes01] und dem dazugehörigen *Rational Rose*-Werkzeug von IBM eingesetzt.

Verschiedene Hersteller bieten grafische Editoren für UML an. Neben dem Zeichnen von UML-Diagrammen können viele dieser Editoren auch Sourcecode- Templates aus Diagrammen erzeugen, die anschließend von Softwareentwicklern vervollständigt werden müssen. Auch Projekte wie *AndroMDA* beschäftigen sich mit der Codegenerierung aus UML-Diagrammen [And09]. Bei AndroMDA erzeugen verschiedene templatebasierte Generatoren (*Cartridges*) Sourcecode aus XMI-Dateien für unterschiedliche Zielsprachen und -architekturen. XMI ist ein UML-Exportformat, das von allen gängigen UML-Werkzeugen unterstützt wird. Die Mehrheit der Cartridges erzeugen Sourcecoderahmen, die von einem Entwickler weiter vervollständigt werden müssen. Nur für triviale Anwendungsfälle wie ER-Diagramme ist eine vollständige Generierung des Sourcecodes möglich.

Häufig genannter Kritikpunkt an UML ist die lose Kopplung und die unpräzise Semantik der einzelnen Diagrammtypen. In der Regel reichen die Informationen aus den UML-Diagrammen nicht für den Programmierer aus, um einen funktionstüchtigen Sourcecode zu erstellen. Zusätzliche Informationen sind daher notwendig, die entweder undokumentiert oder an UML vorbei ausgetauscht werden. Für komplexe Systeme wie ein ERP-System in einem Industrieunternehmen nimmt die Anzahl der benötigten Diagramme stark zu. Als Folge wird es für den Modellierer immer schwieriger, das Zusammenspiel aller Diagramme zu kontrollieren.

Für die Definition von Geschäftsprozessen ist UML ungeeignet, da bereits ein detailliertes Wissen über die Implementierung vorausgesetzt wird. Damit ein Anwendungsexperte UML zur Definition von Prozessen nutzen kann, müsste er sich in die Semantik der UML-Diagrammtypen und die verwendeten Technologien einarbeiten (vgl. Anforderung G1: Intuitivität). Auch ist der agile Einsatz von UML schwierig und kann zu Inkonsistenzen führen. Durch das Überarbeiten von UML-Diagrammen kann bereits vervollständigter Sourcecode verloren gehen. Je nach Änderung im Diagramm muss dann vom Programmierer von Hand der Source zwischen der alten und der neu generierten Version übertragen und gemischt werden. Das verleitet dazu, nachträgliche Änderungen nur noch am Sourcecode vorzunehmen und die dazugehörigen UML-Diagramme nicht mehr zu pflegen (vgl. Anforderung G3: Konsistenz).

WF

Die *Workflow Foundation* (WF) von Microsoft ist ein Bestandteil des *.NET-Frameworks* zum Entwurf und zur Ausführung von Workflows auf Windows-Plattformen. Mit dem *Visual Studio* können neben dem grafischen Entwurf auch

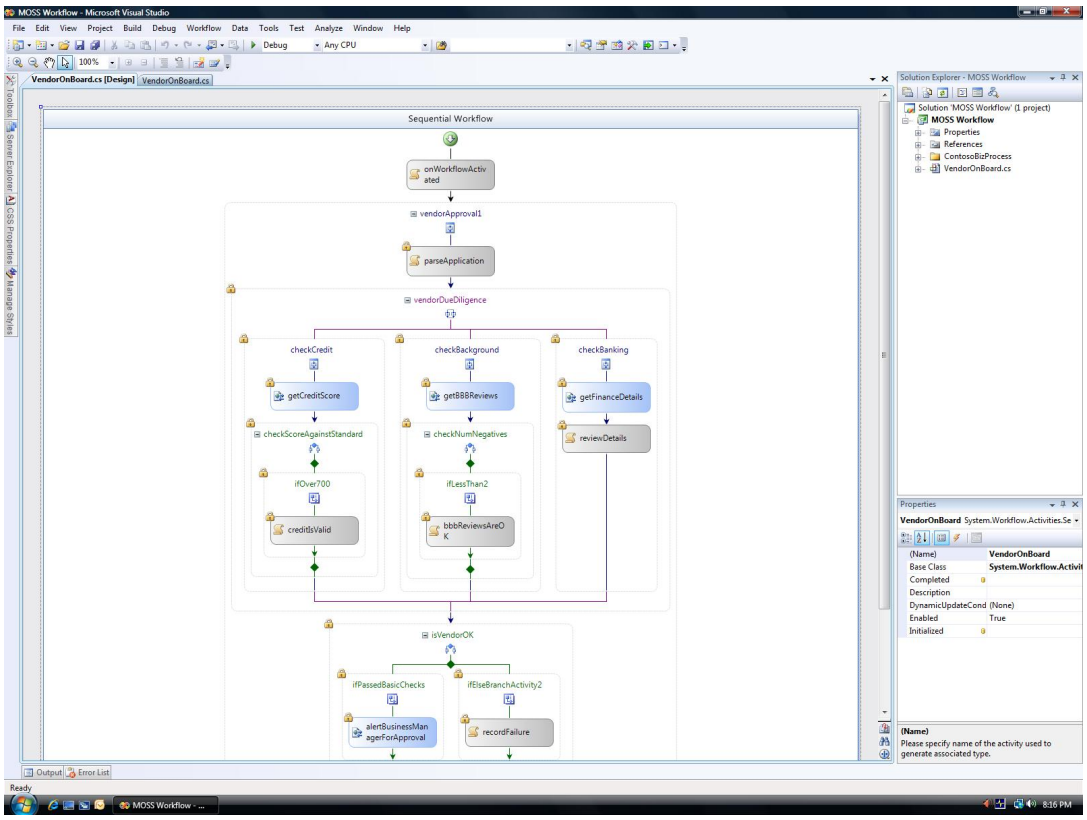

Abbildung 6.4: Visual Studio Workflowdesigner (aus: [Mic07])

alle *.NET*-Sprachen zur Definition und Implementierung von Workflows genutzt werden (siehe Abbildung 6.4). Im Grunde handelt es sich bei der Workflow Foundation allerdings eher um eine Bibliothek als um eine Prozessengine und ist daher stark mit dem .NET-Framework verwoben (vgl. Anforderung G5: Universalität). Weiterhin kann WF nur auf aktuellen Microsoft-Betriebssystemen eingesetzt werden (vgl. Anforderung G4: Plattformunabhängigkeit).

In Visual Studio existieren verschiedene Templates für WF-Prozesse. Diese werden in zwei Basiskategorien unterschieden: den *Sequential Workflow* und den *State Machine Workflow*. Der *State Machine Workflow* realisiert einen endlichen Automaten mit Events, persistenten Zuständen und diversen Zustandsübergängen. Der *Sequential Workflow* mit Schleifen und Bedingungen entspricht einem Kontrollflussgraphen. Workflows werden als XML-Dokument im XAML-Format[1] gespeichert. Das folgende Beispiel zeigt einen *Sequential Workflow* als XAML-Dokument mit einer Schleife und zwei geschachtelten Bedingungen (aus

[1]Extensible Application Markup Language

[Ali07]).

MyWorkFlow.xoml

```
<SequentialWorkflowActivity x:Class="WFApplication.MyWorkflow"
    x:Name="MyWorkflow"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns=
  "http://schemas.microsoft.com/winfx/2006/xaml/workflow">
    <WhileActivity x:Name="whileActivity1">
        <WhileActivity.Condition>
            <RuleConditionReference ConditionName="Cond1" />
        </WhileActivity.Condition>
    <IfElseActivity x:Name="ifElseActivity1">
      <IfElseBranchActivity x:Name="ifElseBranchActivity1">
          <CodeActivity x:Name="codeActivity1"
              ExecuteCode="codeActivity1_ExecuteCode" />
    </IfElseBranchActivity>
      <IfElseBranchActivity x:Name="ifElseBranchActivity2">
        <CodeActivity x:Name="codeActivity2"
            ExecuteCode="codeActivity2_ExecuteCode" />
    </IfElseBranchActivity>
    </IfElseActivity>
    </WhileActivity>
</SequentialWorkflowActivity>
```

Die Workflow Foundation hat sich im Vergleich zu Aris oder BPEL bisher kaum durchgesetzt. Haupteinsatzgebiet scheint die Automation von Microsoft-Produkten wie Office zu sein. In den Sharepoint Server 2007 wurde daher eine Runtimeversion der Workflow Foundation integriert. Andere Serverprodukte von Microsoft wurde inzwischen auch entsprechend ausgestattet.

IDE

Auch *Integrierte Entwicklungsumgebungen* wie Eclipse [Ecl07] oder Netbeans [SUN07b] unterstützen mit Plugins den Entwurf und die Implementierung von Geschäftsprozessen. Mit UML-Plugins oder BPEL-Erweiterungen (siehe Abbildung 6.1 auf Seite 88) kann sich der Entwickler ein auf sich persönlich zugeschnittenes Entwicklungswerkzeug zusammenstellen. So können sogar innerhalb eines Entwicklungsteams z.B. UML-Plugins von unterschiedlichen Herstellern verwendet werden. Weiterhin bieten IDEs Funktionen wie das Versionieren

in RCS Systemen, Refactoring von Sourcecode oder elementares Codegenerieren bereits an. Die Entwicklung neuer Plugins mit grafischer Semantik wird durch spezialisierte Frameworks wie das *Eclipse Graphical Modeling Framework* (GMF) [Ecl09a] und dem *Eclipse Modeling Framework Project* (EMF) [Ecl09b] besonders erleichtert. Projekte wie *openArchitectureWare* (oAW) [oAW09] haben diese Basis für sich entdeckt und entwickeln daraus eigene Codegeneratoren. Für den Softwareentwickler sind IDEs ein gewohntes Werkzeug und trotz zum Teil instabiler Plugins werden solche Umgebungen häufig eingesetzt. Für einen *Business Developer* sind diese Werkzeuge wegen der technischen Ausrichtung in der Regel allerdings unbrauchbar (vgl. Anforderung G1: Intuitivität).

Auch das jABC nutzt die Mächtigkeit und Funktionsvielfalt von aktuellen IDEs aus (vgl. Kapitel 3.2 auf Seite 20). Allerdings gibt es im jABC eine strikte Trennung zwischen dem SIB-Experten, der mit einer beliebigen IDE SIBs produziert und dem Anwendungsexperten, der mit dem jABC Prozessmodelle definiert. Diese Aufteilung in verschiedene Rollen ist ein Grund dafür, warum der jABC-Prozesseditor nicht als IDE-Plugin sondern als eigenständige Anwendung realisiert wurde.

Zusammenfassung

Die folgende Tabelle gibt einen zusammenfassenden Überblick, welche grundlegenden Anforderungen von den angesprochenen Systemen berücksichtigt werden (vgl. Kapitel 1.1 auf Seite 3). Vor allem die Anforderungen Agilität (G2) und Konsistenz (G3) können nicht von solchen Methoden befriedigend gelöst werden, die keine vollautomatische Codegenerierung anbieten. Bei einer halbautomatischen Codegenerierung werden nur Fragmente des Sourcecodes erstellt, die von Programmierern vervollständigt werden müssen. Bei späteren Änderungen des Modells müssen entsprechende Anpassungen im bestehenden Sourcecode manuell nachgepflegt werden. Dies reduziert die Agilität und erhöht das Risiko von Inkonsitenzen.

Anforderung	jABC	BPEL	ARIS	UML	WF	IDE
Intuitivität (G1)	●	◉	⊙	⊙	◉	○
Agilität (G2)	●	◉	⊙	⊙	◉	⊙
Konsistenz (G3)	●	◉	⊙	⊙	○	⊙
Plattformunabh. (G4)	●	●	●	●	○	◉
Universalität (G5)	●	⊙	◉	●	⊙	●
Skalierbarkeit (G6)	●	◉	●	●	●	●
Korrektheit (G7)	●	○	⊙	⊙	○	◉

Legende

- ○ nicht erfüllt
- ⊙ kaum erfüllt
- ◉ annähernd erfüllt
- ● vollständig erfüllt

7 Einsatzbereiche und Ausblick

7.1 Einsatzbereiche

Das jABC wurde in den vergangenen Jahren in einer Vielzahl unterschiedlicher Forschungs- und Industrieprojekten eingesetzt. Die nachfolgende Übersicht zeigt das breite Spektrum der Anwendungen.

Prozessaufnahme in einem Zulieferbetrieb der Flugzeugindustrie
In diesem Projekt wurden die Prozessabläufe in einem Zulieferbetrieb der Flugzeugindustrie als jABC-Modelle aufgenommen und dokumentiert. Diese Modelle dienten zur Analyse und Optimierung der bestehenden Prozesse.

Entwicklung eines Dokumentmanagement-Prozesses
Beim internationalen Wareneinkauf von Möbeln am Weltmarkt muss mit einer Vielzahl von unterschiedlichen nationalen Warenbegleit- und Zolldokumenten umgegangen werden. In diesem Projekt wurde anstatt der sonst in diesem Unternehmen üblichen RUP-Spezifikation das jABC eingesetzt. Die Erfahrungen daraus wurden später veröffentlicht [HMM+08].

Marketingplanung in einer Textilkette
Für die Planung von Marketingaktionen in Zeitungen und Fernsehen wurde in diesem Projekt in einer großen Textilkette ein Prozess aufgenommen, welche Schritte ein neues Produkt von der ersten Planung bis zum Verkauf in den Läden durchläuft. Im Zentrum des Prozesses steht dabei ein Kalender, der die zeitlichen Abläufe koordiniert.

Kapazitätsplanung in einem Telekommunikationsunternehmen
Ziel dieses Projekts war eine verbesserte Kapazitätsplanung der Service-Mitarbeiter in einem großen deutschen Telekommunikationsunternehmen. Durch zyklische Schwankungen variiert die Anzahl der Reparatur- und Installationsaufträge. Mit der Aufnahme dieses Prozesses können die Vorhersageparameter präzisiert werden.

CeBIT Präsentation: Exam and Go
Dieses Projekt wurde speziell für die CeBIT 2007 entwickelt. Es demonstriert einen automatisierten Anmeldungeprozess für Prüfungen an einer

Universität. Dazu wurden verschiedene Subsysteme orchestriert: ein Universitäten-Verwaltungssystem, ein DMS, ein Kalender, eine OCR-Software und ein MFP-Gerät[1] zum Scannen und Drucken.

jETI

Die *Electronic Tool Integration Platform* (jETI) ermöglicht das installationsfreie Experimentieren mit Softwarewerkzeugen aus unterschiedlichsten Kategorien [MNS05a]. *Bio-jETI* ist eine thematische Spezialisierung der Plattform im Bereich der Bioinformatik-Forschung [LMS08].

jITE

Das *Integrated Test Enviroment* führt durch jABC-Modelle gesteuerte Tests von Webanwendungen aus, die manuell entworfen oder mittels des *Web Recorders* vorher aufgezeichnet wurden [RMSM08].

OCS

Der *Online Conference Service* ist ein webbasiertes Entscheidungsunterstützungssystem zur Auswahl wissenschaftlicher Veröffentlichungen für Konferenzen. Das System koordiniert die Einreichungen und den Begutachtungsprozess. Das noch mit der Vorgänger Version entwickelte System wird zur Zeit einem Redesign mit dem jABC unterzogen [KM06].

Connect-IT

Mit dieser speziellen Version des jABCs können Schüler und Studenten sehr einfach eigene Strategien für das bekannte *Vier-gewinnt* Spiel entwickeln und testen. Im Turniermodus treten dann verschiedene Strategien gegeneinander an und es wird ein Sieger ermittelt [Sve08].

Firewall-Plugin

Mit dem Firewall-Plugin können Firewallregeln, die mit dem *Firewall Builder* definiert wurden, im jABC bearbeitet und simuliert werden. Die Simulation der Firewallregeln und die Generierung von TCP/IP-Testpaketen ermöglicht das einfache und frühzeitige Testen von Firewallregeln unabhängig von der eingesetzten Hard- und Software [Win08].

XSLT-Plugin

Das XSLT-Plugin ermöglicht die Transformation von strukturierten Daten per XSLT. In drei Ebenen können mit unterschiedlichen Semantiken XSLT-Transformationen mit dem jABC modelliert werden. Ein Codegenerator übersetzt die Modelle in XSLT, die dann ausgeführt werden können [Neu08].

MindstormsNXT

In dieser Arbeit wurde ein Codegenerator für den *MindstormsNXT*-Microcontroller von Lego entwickelt. So kann der NXT-Controller mit dem jABC

[1] MFP: Multifunction Peripheral

grafisch programmiert werden. In Schulen wird das *MindstormsNXT*-Baukastensystem häufig zur spielerischen Konstruktion und Programmierung von Robotern eingesetzt [Sch07].

DBSchema-Plugin
Mit diesem Plugin können ER-Diagramme für relationale Datenbanken entwickelt werden. Das Plugin kann diese Schemata dann in JDBC-kompatiblen Datenbanken anlegen oder existierende Schemata importieren. Das Plugin kann sogar Schemaänderungen vornehmen, obwohl die Tabellen bereits mit Daten gefüllt sind [Win06].

7.2 Ausblick

Aus den Erfahrungen beim Einsatz des jABCs in Forschungs- und Industrieprojekten entstanden neue Anforderungen oder Änderungswünsche, die in die nächste Generation des jABCs einfließen werden. Dieses Kapitel gibt einen groben Überblick über die Weiterentwicklungen.

Vollständige Flexibilisierung

Das jABC kann durch die Framework-Architektur und den modularen Aufbau vielseitig an unterschiedliche Aufgaben angepasst werden. Viele Eigenschaften des jABCs werden erst zur Laufzeit durch die Konfigurations-API festgelegt. Es gibt allerdings einige wenige Bestandteile, die bisher nicht austauschbar sind. Dazu gehören die Zeichenfläche und die verwendeten Gesten beim Zeichnen von Modellen. Eine neue Anforderung für die Weiterentwicklung könnte die vollständige Flexibilisierung und Austauschbarkeit aller jABC-Bestandteile sein.

Update-Server

Für die aktuelle Version des jABCs können über ein *Maven*-Projekt unterschiedliche jABC-Installer mit verschiedenen Plugins, SIB-Paletten und Projekt-Zusammenstellungen erzeugt werden. Die zunehmende Anzahl der Plugins macht dies zu einer zeitaufwändigen Aufgabe. Zu dem Aufwand gehört neben der eigentlichen Release-Phase und dem Integrationstest auch die Softwareverteilung der unterschiedlichen Distributionen an entsprechende Empfänger. Mit einem Update-Server könnten Updates an bereits existierende jABC-Installationen ausgeliefert werden. Dadurch kann in vielen Fällen auf die Zusammenstellung eines speziellen Installers verzichtet werden.

Tests der grafischen Benutzeroberfläche

Das Testen von grafischen Benutzeroberflächen ist häufig sehr schwierig oder unmöglich. Einige Projekte beschäftigen sich mit der Automatisierung von Benutzerinteraktionen durch eine Maus oder Tastatur. Dabei ist die Kontrolle des erwarteten Ergebnisses meist das Problem in einem solchen Test. Beim jABC ist vor allem das Testen von Erweiterungen der Zeichenfläche durch neue Gesten oder Popupmenüs schwer möglich. Durch spezielle Textalternativen kann das automatische Testen der grafischen Benutzeroberfläche vereinfacht werden.

Vereinheitlichung der Benutzeroberfläche

Durch unterschiedliche Autoren ist die Bedienung der verschiedenen Plugins und Dialoge uneinheitlich. Zusätzlich wurden zum Teil auch unterschiedliche Icon-Paletten verwendet. Für den Anwender ist dies besonders in der Eingewöhnungsphase verwirrend. Mit einem verbindlichen Styleguide kann das Aussehen und die Bedienung aller Dialoge leicht vereinheitlicht werden.

7.3 Zusammenfassung

In dieser Arbeit wurde das jABC vorgestellt, welches ein Java-Framework zur modellgetriebenenen Beschreibung von Prozessen nach dem *One-Thing-Approach* ist. Durch Plugins kann das System um neue Funktionen erweitert werden. Auch die Semantik der Graphen ist nicht festgelegt und kann durch Plugins ausgetauscht werden. Das Komponentenmodell des jABC basiert auf der Programmiersprache Java und ist besonders einfach und universell. In einer Vielzahl von Diplomarbeiten und Projekten wurde das Potential des Systems gezeigt.

Literaturverzeichnis

[A+03] Tony Andrews et al. *Business Process Execution Language for Web Services*. 2nd public draft release, Version 1.1, May 2003.

[AC96] T. Margaria V. Braun A. Claßen, B. Steffen. Tool Coordination in METAFrame. Technical report, Fakultät für Mathematik und Informatik, Universität Passau, 1996.

[Ali07] Ali Iqbal Khan. Introduction to XAML in Windows Workflow Foundation. Internet, `http://www.codeproject.com/KB/WF/XAML_WF.aspx`, 2007.

[And09] AndroMDA Team. AndroMDA. Internet, `http://www.andromda.org/`, 2009.

[Apa07a] Apache Software Foundation. Apache Log4j Projekt. Internet, `http://logging.apache.org/log4j/`, 2007.

[Apa07b] Apache Software Foundation. Apache Maven Projekt. Internet, `http://maven.apache.org/`, 2007.

[Apa07c] Apache Software Foundation. Xerces2 Java Parser. Internet, `http://xerces.apache.org/xerces2-j/`, 2007.

[BBC+97] Michael von der Beeck, Volker Braun, Andreas Claßen, A. Dannecker, C. Friedrich, Dirk Koschützki, Tiziana Margaria, F. Schreiber, and Bernhard Steffen. Graphs in METAFrame: The Unifying Power of Polymorphism. In *TACAS '97: Proceedings of the Third International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, pages 112–129, London, UK, 1997. Springer-Verlag.

[BM06] Markus Bajohr and Tiziana Margaria. MaTRICS: A service-based management tool for remote intelligent configuration of systems. *ISSE*, 2(2):99–111, 2006.

[BM08] Markus Bajohr and Tiziana Margaria. High Service Availability in MaTRICS for the OCS. In *ISoLA*, pages 572–586, 2008.

[BMRS07a] Marco Bakera, Tiziana Margaria, Clemens D. Renner, and Bernhard Steffen. Property-driven functional healing: Playing against undesired behavior. In *Business Process Engineering, Proceedings of the CONQUEST 2007*, 2007.

[BMRS07b] Marco Bakera, Tiziana Margaria, Clemens D. Renner, and Bernhard Steffen. Verification, Diagnosis and Adaptation: Tool supported enhancement of the model-driven verification process. In Yamine Ait-Ameur, Frederic Boniol, and Virginie Wiels, editors, *ISoLA Workshop*, 2007.

[BMRS08] Marco Bakera, Tiziana Margaria, Clemens D. Renner, and Bernhard Steffen. Model Checking with the jABC Framework - From METAGame to GEAR. . In *Intern. Symposium on Quality Engineering for Embedded Systems (QEES)*, 2008.

[BS95] A. Claßen B. Steffen, T. Margaria. Incremental Formalization: a Key to Industrial Success. Technical report, Fakultät für Mathematik und Informatik, Universität Passau, 1995.

[Col07] CollabNet. Subversion Version Control System. Internet, `http://subversion.tigris.org/`, 2007.

[Ded05] Edin Dedagic. Erstellung einer bidirektionalen Integrationsarchitektur zwischen dem jABC Framework und der Eclipse Plattform zur dualen Steuerung von Modellierungsschritten als Graph oder Sourcecode. Diplomarbeit, Lehrstuhl für Programmiersysteme, Universität Dortmund, Oktober 2005.

[Deu06] Deutsches Rotes Kreuz. Wiederbelebungs-Richtlinien 2006. In *Ausbilder-INFO, Nr. 37 01/2006*, page 2, 2006.

[Doe06] Markus Doedt. Erweiterung der jABC Framework Bibliothek um eine durch entsprechenden Hotspot individuell anpassbare Ausführungsschicht. Diplomarbeit, Lehrstuhl für Programmiersysteme, Universität Dortmund, Mai 2006.

[Doe09] Markus Doedt. Handbuch Tracer-Plugin. Internet, `http://jabc.cs.tu-dortmund.de/manual/index.php/Tracer_Manual`, 2009.

[Ecl07] Eclipse Foundation. Eclipse - an open development platform. Internet, `http://www.eclipse.org/`, 2007.

[Ecl09a] Eclipse Foundation. Eclipse Graphical Modeling Framework. Internet, `http://www.eclipse.org/modeling/gmf/`, 2009.

[Ecl09b] Eclipse Foundation. Eclipse Modeling Framework Project. Internet, `http://www.eclipse.org/modeling/emf/`, 2009.

[Fow04] Martin Fowler. *UML konzentriert.* Addison-Wesley, München [u.a.], 2004.

[Fra02] David Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing.* John Wiley & Sons, Inc., New York, NY, USA, 2002.

[Fra08] Frank R. Lehmann. *Integrierte Prozessmodellierung mit ARIS*. dpunkt Verlag, 1. aufl. edition, 2008.

[FSMZ95] Burkhard Freitag, Bernhard Steffen, Tiziana Margaria, and Ulrich Zukowski. An Approach to Intelligent Software Library Management. In *DASFAA*, pages 71–78, 1995.

[GNU06] GNU. cvs - Concurrent Versions System. Internet, `http://www.nongnu.org/cvs/`, 2006.

[Hes01] Wolfgang Hesse. RUP - A process model for working with UML. In *Unified Modeling Language: Systems Analysis, Design and Development Issues*, pages 61–74. IGI Global, 2001.

[HMM+06] M. Hörmann, Tiziana Margaria, T. Mender, Ralf Nagel, M. Schuster, Bernhard Steffen, and H. Trinh. The jABC Approach to Collaborative Development of Embedded Applications. In *CCE '06, Int. Workshop on Challenges in Collaborative Engineering*, April 2006. (Industry Day), Prag (CZ).

[HMM+08] Martina Hörmann, Tiziana Margaria, Thomas Mender, Ralf Nagel, Bernhard Steffen, and Hong Trinh. The jABC Approach to Rigorous Collaborative Development of SCM Applications. In *ISoLA*, pages 724–737, 2008.

[How07] Howard Kistler. Ekit: a free open source Java HTML editor applet and application. Internet, `http://www.hexidec.com/ekit.php`, 2007.

[Hös08] Stefan Hösel. Entwicklung eines Plug-ins für das jABC-Framework zur Integration von .NET mittels C. Diplomarbeit, Service und Software Engineering, Universität Potsdam, April 2008.

[IBM07] IBM, Bea Systems. BPELJ: BPEL for Java technology. Internet, `http://www.ibm.com/developerworks/library/specification/ws-bpelj/`, 2007.

[JB09] Sven Jörges and Benjamin Bentmann. Handbuch Genesys-Plugin. Internet, `http://jabc.cs.tu-dortmund.de/manual/index.php/Genesys_(Developer_Manual)`, 2009.

[JGr07] JGraph Ltd. Java Graph Visualization and Layout. Internet, `http://www.jgraph.com/`, 2007.

[JKN+06] Sven Jörges, Christian Kubczak, Ralf Nagel, Tiziana Margaria, and Bernhard Steffen. Model-Driven Development with the jABC. In *HVC - IBM Haifa Verification Conference*, Haifa, Israel, October 2006. IBM.

[JKPM07] Sven Jörges, Christian Kubczak, Felix Pageau, and Tiziana Margaria. Model Driven Design of Reliable Robot Control Programs Using the jABC. *Engineering of Autonomic and Autonomous Systems, IEEE International Workshop on*, 0:137–148, 2007.

[JMN+08] Georg Jung, Tiziana Margaria, Ralf Nagel, Wolfgang Schubert, Bernhard Steffen, and Horst Voigt. SCA and jABC: Bringing a Service-Oriented Paradigm to Web-Service Construction. In *ISoLA*, pages 139–154, 2008.

[JMS06] Sven Jörges, Tiziana Margaria, and Bernhard Steffen. FormulaBuilder: a tool for graph-based modelling and generation of formulae. In *ICSE*, pages 815–818, 2006.

[JMS08] Sven Jörges, Tiziana Margaria, and Bernhard Steffen. Genesys: Service-Oriented Construction of Property Conform Code Generators. *Innovations in Systems and Software Engineering*, 4(4):361–384, 2008.

[Joe08] Joe Walnes and XStream Committers. XStream, a simple library to serialize objects to XML. Internet, `http://xstream.codehaus.org/`, 2008.

[Jul09] Julien Ponge. IzPack solution for packaging, distributing and deploying applications. Internet, `http://izpack.org/`, 2009.

[KM06] Martin Karusseit and Tiziana Margaria. A Web-Based Runtime-Reconfigurable Role Management Service. *Automated Specification and Verification of Web Systems, International Workshop on*, 0:53–60, 2006.

[KMK+08] Christian Kubczak, Tiziana Margaria, Matthias Kaiser, Jens Lemcke, and Bjoern Knuth. Abductive Synthesis of the Mediator Scenario with jABC and GEM. In *EON*, 2008.

[KMSN07] Christian Kubczak, Tiziana Margaria, Bernhard Steffen, and Stefan Naujokat. Service-Oriented Mediation with jETI/jABC: Verification and Export. *Web Intelligence and Intelligent Agent Technology, International Conference on*, 0:144–147, 2007.

[KMWS07] Christian Kubczak, Tiziana Margaria, Christian Winkler, and Bernhard Steffen. An Approach to Discovery with miAamics and jABC. *Web Intelligence and Intelligent Agent Technology, International Conference on*, 0:157–160, 2007.

[KNMS06] Christian Kubczak, Ralf Nagel, Tiziana Margaria, and Bernhard Steffen. The jABC Approach to Mediation and Choreography. In *Semantic Web Service Challenge Phase I-III Workshops*. DERI, Stanford University, March-November 2006.

[Knu64] Donald E. Knuth. Backus Normal Form vs. Backus Naur Form. *Commun. ACM*, 7(12):735–736, 1964.

[LMS06] Anna-Lena Lamprecht, Tiziana Margaria, and Bernhard Steffen. Data-Flow Analysis as Model Checking Within the jABC. In *CC*, pages 101–104, 2006.

[LMS08] Anna-Lena Lamprecht, Tiziana Margaria, and Bernhard Steffen. Seven Variations of an Alignment Workflow - An Illustration of Agile Process Design and Management in Bio-jETI. In *ISBRA*, pages 445–456, 2008.

[Man09] Mantis. Mantis Bug Tracker. Internet, `http://www.mantisbt.org/`, 2009.

[Mar07] Tiziana Margaria. Service Is in the Eyes of the Beholder. *Computer*, 40(11):33–37, 2007.

[MBRS08] Tiziana Margaria, Marco Bakera, Harald Raffelt, and Bernhard Steffen. Synthesizing the Mediator with jABC/ABC. In Raul Garcia-Castro, Asunción Gómez-Pérez, Charles J. Petrie, Emanuele Della Valle, Ulrich Küster, Michal Zaremba, and M. Omair Shafiq, editors, *EON*, volume 359 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.

[Mic07] Microsoft. TechEd Developers and TechEd IT Forum 2007 Virtual Pressroom. Internet, `http://www.microsoft.com/emea/presscentre/teched07/imagegallery.mspx`, 2007.

[MNS02] Tiziana Margaria, Oliver Niese, and Bernhard Steffen. Demonstration of an automated integrated test environment for web-based applications. In *Proceedings of the 9th International SPIN Workshop on Model Checking of Software*, pages 250–253, London, UK, 2002. Springer-Verlag.

[MNS05a] Tiziana Margaria, Ralf Nagel, and Bernhard Steffen. Remote Integration and Coordination of Verification Tools in JETI. In *Proc. of the 12th IEEE Int. Conf. on the Engineering of Computer-Based Systems (ECBS 2005)*, pages 431–436. IEEE Computer Society, 2005.

[MNS05b] Tiziana Margaria, Ralf Nagel, and Bernhard Steffen. Remote Integration and Coordination of Verification Tools in jETI. In *ECBS*, pages 431–436, 2005.

[MOSS99] Markus Müller-Olm, David A. Schmidt, and Bernhard Steffen. Model-Checking: A Tutorial Introduction. In *SAS '99: Proceedings of the 6th International Symposium on Static Analysis*, pages 330–354, London, UK, 1999. Springer-Verlag.

[Moz09] Mozilla.org. Rhino: JavaScript for Java. Internet, `http://www.mozilla.org/rhino/`, 2009.

[MS04] Tiziana Margaria and Bernhard Steffen. Lightweight coarse-grained coordination: a scalable system-level approach. *Int. J. Softw. Tools Technol. Transf.*, 5(2):107–123, 2004.

[MS06] Tiziana Margaria and Bernhard Steffen. Service Engineering: Linking Business and IT. *Computer*, 39(10):45–55, 2006.

[MS07] Tiziana Margaria and Bernhard Steffen. LTL Guided Planning: Revisiting Automatic Tool Composition in ETI. *Software Engineering Workshop, Annual IEEE/NASA Goddard*, 0:214–226, 2007.

[MS08] Tiziana Margaria and Bernhard Steffen. Agile it: Thinking in user-centric models. In *ISoLA*, pages 490–502, 2008.

[MSUW04] Stephen J. Mellor, Kendall Scott, Axel Uhl, and Dirk Weise. *MDA Distilled: Principles of Model-Driven Architecture*. Addison Wesley Professional, März 2004.

[MWK+07] Tiziana Margaria, Christian Winkler, Christian Kubczak, Bernhard Steffen, Marco Brambilla, Stefano Ceri, Dario Cerizza, Emanuele Della Valle, Federico Michele Facca, and Christina Tziviskou. The SWS Mediator with WEBML/WEBRATIO and JABC/JETI: A Comparison. In *ICEIS (4)*, pages 422–429, 2007.

[NE09] Charles Oliver Nutter and Thomas Enebo. JRuby. Internet, `http://jruby.codehaus.org/`, 2009.

[Neu07] Johannes Neubauer. LocalChecker Plugin for the jABC. Studienarbeit, Lehrstuhl für Programmiersysteme, Universität Dortmund, July 2007.

[Neu08] Johannes Neubauer. Entwicklung eines jABC Plugins zur grafischen Beschreibung von Transformationen strukturierter Daten. Diplomarbeit, Lehrstuhl für Programmiersysteme, Universität Dortmund, April 2008.

[Neu09] Johannes Neubauer. Handbuch LocalChecker-Plugin. Internet, `http://jabc.cs.tu-dortmund.de/manual/index.php/LocalChecker_Manual`, 2009.

[NM08] Manfred Nagl and Wolfgang Marquardt, editors. *Collaborative and Distributed Chemical Engineering. From Understanding to Substantial Design Process Support - Results of the IMPROVE Project*, volume 4970 of *Lecture Notes in Computer Science*. Springer, 2008.

[NR02] Markus Nüttgens and Frank J. Rump, editors. *EPK 2002 - Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten,*

Proceedings des GI-Workshops und Arbeitskreistreffens (Trier, November 2002). GI-Arbeitskreis Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten, 2002.

[oAW09] oAW Team. openArchitectureWare MDA/MDD generator framework. Internet, `http://www.openarchitectureware.org/`, 2009.

[OSG09] OSGi Alliance. OSGi Alliance. Internet, `http://www.osgi.org`, 2009.

[RB09] Clemens Renner and Marco Bakera. Handbuch GEAR-ModelChecker. Internet, `http://jabc.cs.tu-dortmund.de/manual/index.php/GEAR`, 2009.

[RMSM08] Harald Raffelt, Tiziana Margaria, Bernhard Steffen, and Maik Merten. Hybrid test of web applications with webtest. In *TAV-WEB*, pages 1–7, 2008.

[Rom07] Roman Strobl. Faster Enterprise Application Development with NetBeans 5.5. Internet, `http://www.netbeans.org/kb/articles/faster-enterprise-app.html`, 2007.

[RSM07] Harald Raffelt, Bernhard Steffen, and Tiziana Margaria. Dynamic Testing Via Automata Learning. In *Haifa Verification Conference*, pages 136–152, 2007.

[RSMM08] Harald Raffelt, Bernhard Steffen, Tiziana Margaria, and Maik Merten. Hybrid test of web applications with webtest. In *TAV-WEB '08: Proceedings of the 2008 workshop on Testing, analysis, and verification of web services and applications*, pages 1–7, 2008.

[Sch07] Björn Schulte. Modellgetriebene Steuerung eingebetteter Systeme und ihrer Anwendung für Lego NXT. Diplomarbeit, Lehrstuhl für Programmiersysteme, Universität Dortmund, Juli 2007.

[SFC+94] B. Steffen, B. Freitag, A. Claßen, T. Margaria, and U. Zukowski. Intelligent software synthesis in the DaCapo environment. Technical report, University of Aarhus, 1994.

[SM99] Bernhard Steffen and Tiziana Margaria. METAFrame in Practice: Design of Intelligent Network Services. In *Correct System Design*, pages 390–415, 1999.

[SMB96] Bernhard Steffen, Tiziana Margaria, and Volker Braun. The MetaFrame'95 Environment. In *Proc. CAV'96, LNCS*, pages 450–453. Springer, 1996.

[SMC95] Bernhard Steffen, Tiziana Margaria, and Andreas Claßen. The META-Frame: An Environment for Flexible Tool Management. In *TAPSOFT '95: Proceedings of the 6th International Joint Conference*

CAAP/FASE on Theory and Practice of Software Development, pages 791–792, London, UK, 1995. Springer-Verlag.

[SMN05] Bernhard Steffen, Tiziana Margaria, and Ralf Nagel. jETI: A Tool for Remote Tool Integration. In Nicolas Halbwachs and Lenore D. Zuck, editors, *Proc. of 11th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*, volume 3440 of *LNCS*, Edinburgh, UK, April 2005. Springer Verlag.

[SN07] Bernhard Steffen and Prakash Narayan. Full Life-Cycle Support for End-to-End Processes. *Computer*, 40(11):64–73, 2007.

[SUN07a] SUN Microsystems. Java Enterprise Edition (Java EE). Internet, `http://java.sun.com/javaee/`, 2007.

[SUN07b] SUN Microsystems. NetBeans IDE. Internet, `http://www.netbeans.org/`, 2007.

[SUN09a] SUN Microsystems. JavaMail API 1.4.2. Internet, `http://java.sun.com/products/javamail/`, 2009.

[SUN09b] SUN Microsystems. Netbeans Enterprise Pack. Internet, `http://www.netbeans.org/products/enterprise/`, 2009.

[SUN09c] SUN Microsystems. Solaris ZFS. Internet, `http://www.sun.com/software/solaris/zfs_learning_center.jsp`, 2009.

[Sve08] Sven Jörges and Marco Bakera. Webseite des Projekts ConnectIT. Internet, `http://connectit.cs.tu-dortmund.de/`, 2008.

[TM96] B. Steffen T. Margaria. Automatic Synthesis of Design Plans in META-Frame. Technical report, Fakultät für Mathematik und Informatik, Universität Passau, 1996.

[TN86] H. Takeuchi and I. Nonaka. The new new product development game. *Harvard Business Review*, 1986.

[Uni07] Universität Dortmund. jABC Website. Internet, `http://www.jabc.de.org/`, 2007.

[W3C01] W3C. Web Services Description Language (WSDL) 1.1. Internet, `http://www.w3.org/TR/wsdl/`, 2001.

[W3C07] W3C. Simple Object Access Protocol (SOAP) 1.1. Internet, `http://www.w3.org/TR/2000/NOTESOAP-20000508/`, 2007.

[WB06] Ralf Wirdemann and Thomas Baustert. *Rapid web development mit Ruby on Rails*. Hanser, München (u.a.), 2006.

[WCL+05] Sanjiva Weerawarana, Francisco Curbera, Frank Leymann, Tony Storey, and Donald F. Ferguson. *Web Services Platform Architecture :*

SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More. Prentice Hall PTR, March 2005.

[WHKL08] Gerd Wütherich, Nils Hartmann, Bernd Kolb, and Matthias Lübken. *Die OSGi Service Platform: Eine Einführung mit Eclipse Equinox*. dpunkt, Heidelberg, 2008.

[Win06] Christian Winkler. Entwicklung eines jABC-Plugins zum Design von JDBC-kompatiblen Datenbankschemata. Diplomarbeit, Lehrstuhl für Programmiersysteme, Universität Dortmund, März 2006.

[Win08] Stephan Windmüller. Verifikation von Firewall Regel mit dem jABC. Diplomarbeit, Lehrstuhl für Programmiersysteme, Universität Dortmund, Juli 2008.

[WM08] Stephen A. White and Derek Miers. *BPMN Modeling and Reference Guide*. Future Strategies Inc., Lighthouse Pt, FL, City, 2008.

[Wol09] Wolf Paulus. SwiX$^{\text{ml}}$ small GUI generating engine for Java applications and applets. Internet, `http://www.swixml.org/`, 2009.

Abbildungsverzeichnis