Christian-Albrechts-Universität zu Kiel

Diplomarbeit

Ein Web Service für das automatische Layout von Graphen

cand. inform. Stephan Wersig

31. Oktober 2011

Institut für Informatik
Arbeitsgruppe für Echtzeitsysteme und Eingebettete Systeme

Prof. Dr. Reinhard von Hanxleden

betreut durch:
Miro Spönemann

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Kiel,

_____

Zusammenfassung

Grafische Modellierung findet in immer größerem Umfang Anwendung in der Software-Entwicklung. Die Unterstützung ihrer Anwender durch durchdachte, effiziente Bedienkonzepte befindet sich in vielen Werkzeugen noch in den Anfängen. Die Effizienz im Umgang mit grafischen Sprachen kann durch geeignete *pragmatische* Mittel wie z.B. *Focus and Context* oder *strukturbasiertes Editieren* deutlich verbessert werden. Grundlage dafür ist das *automatische Layout* der visuellen Darstellung. Das im Rahmen des KIELER Forschungsprojekts entwickelte *Meta Layout* bietet eine heterogene, an Algorithmen reiche Infrastruktur für die Berechnung des Layouts von Graphen, welche bislang nur der Java-Plattform zugänglich war. Diese Arbeit beschreibt die Veröffentlichung des KIELER Layouts auf Grundlage eines Web Service, der dem vorgestellten Dienst *KWebS* die geeignete Abstraktion bietet, das KIELER Layout auch Nutzern anderer Plattformen zur Verfügung zu stellen. KWebS bietet seinen Nutzern damit eine plattformunabhängig nutzbare, kostenfreie und öffentlich verfügbare Grundlage zur Umsetzung pragmatischer Techniken und trägt auf diese Art seinen Teil zur Verbesserung der Effizienz grafischer Modellierung bei.

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

Abkürzungsverzeichnis

AJAX	Asynchronous Java and XML
API	Application Programming Interface
CA	Certificate Authority
CI	Continuous Integration
CORBA	Common Object Request Broker Architecture
CSV	Comma Separated Values
CVS	Concurrent Version System
DCOM	Distributed Component Object Model
EMF	Eclipse Modeling Framework
ETI	Electronic Tool Integration
GMF	Graphical Modeling Framework
GUI	Graphical User Interface
HATEOAS	Hypermedia as the Engine of Application State
HTML	Hyper Text Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IDL	Interface Definition Language
JAXB	Java Architecture for XML Binding
jABC	Java Application Building Center
JAX-RPC	Java API for XML-based RPC
JAX-WS	Java API for XML - Web Services
JDT	Java Development Tools
jETI	Java Electronic Tool Integration
JSON	JavaScript Object Notation
KIEL	Kiel Integrated Environment for Layout
KIELER	Kiel Integrated Environment for Layout Eclipse Rich Client
KIML	KIELER Infrastructure for Meta Layout
KWebS	KIELER Web Service for Layout
M2M	Modell-zu-Modell
MOF	Meta Object Facility
OASIS	Organization for the Advancement of Structured Information Standards
OGDF	Open Graph Drawing Framework
OGML	Open Graph Markup Language
OMG	Object Management Group
OSGi	Open Services Gateway initiative
PDE	Plug-In Development Environment
PDF	Portable Document Format
PNG	Portable Network Graphics
RCA	Rich Client Application
RCP	Rich Client Platform
REST	Representational State Transfer
RIA	Rich Internet Application

S-RAMP	SOA Repository Artifact Model & Protocol
SCM	Source Code Management
SEI	Service Endpoint Interface
SEPP	Streaming Eti Performance Protocol
SIB	Service Independent Building Block
SOA	Serviceorientierte Architektur
SSL	Secure Socket Layer
SVG	Scalable Vector Graphics
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDDI	Universal Description, Discovery and Integration
UML	Unified Modeling Language
URL	Uniform Resource Locator
URI	Uniform Resource Identifier
W3C	World Wide Web Consortium
WADL	Web Application Description Language
WiGis	Web-based Interactive Graph Visualizations
WSDL	Web Service Description Language
WWW	World Wide Web
XML	Extensible Markup Language
XMI	XML Metadata Interchange

1. Einleitung

Seit dem Beginn der elektronischen Datenverarbeitung haben sich die Anforderungen an Softwaresysteme stark verändert und entwickelt. Während früher Anwendungen meist für einen sehr begrenzten, klar definierten Anwendungsfall entwickelt wurden, müssen heutige Softwaresysteme mit einem stark heterogenen Umfeld unter verschiedensten Gesichtspunkten wie z.B. Sicherheit, Zuverlässigkeit und Vertraulichkeit interagieren. Die Verifikation von Code gemäß den erstellten Anforderungen ist z.B. im Bereich der eingebetteten Systeme von großer Wichtigkeit. Die stetig steigende Komplexität der Anforderungen an eine Software spiegelt sich auf natürliche Weise in der Art ihrer Erstellung wieder.

Diese Arbeit stellt den *KIELER Web Service for Layout* (KWebS) vor, der seinen Nutzern auf Grundlage eines Web Service das automatische Layout von Graphen anbietet. Um zu verstehen, welche Rolle das automatische Layout in der Entwicklung von Software-Systemen spielt, betrachten wir zunächst in Abschnitt 1.1 den Stellenwert grafischer Modellierung im Bereich der Software-Entwicklung und schauen uns in Abschnitt 1.2 an, wie die Unterstützung ihrer Nutzer in heutigen Werkzeugen umgesetzt ist. In Abschnitt 1.3 lernen wir anschließend Techniken kennen, den Nutzern grafischer Modellierung den Umgang mit Modellen zu erleichtern. Die Grundlage dieser Techniken ist das automatische Layout von Graphen, welches wir in Abschnitt 1.4 betrachten. Abschnitt 1.5 geht darauf folgend auf Web Services ein, bevor Abschnitt 1.6 zum Abschluss der Einleitung das Ziel dieser Arbeit formuliert und einen Überblick über deren Struktur gibt.

1.1. Modellierung von Softwaresystemen

Ein von einem Computer verarbeitetes Programm stellt durch seine Datenstrukturen und logischen Abläufe immer eine Abstraktion der von ihm zu handhabenden Zusammenhänge dar. Es folgt in seiner Ausführung einem Modell eines definierten Ausschnittes der Wirklichkeit. Die Modellbildung ist daher ein sehr entscheidender Schritt für die Qualität des Produkts schon in der Planungsphase einer Software. In den Pionier-Zeiten der Software-Entwicklung lag ein Modell meist nur in gedanklicher Form oder in Form einer willkürlich gewählten, technisch nicht nutzbaren Notation vor. Mit steigender Komplexität von Software-Architekturen war dieser Ansatz nicht mehr tragbar und es mussten neue Methoden gefunden werden, die Strukturen einer Architektur zu formulieren.

Die Entwicklung von Modellen ist ein im Allgemeinen komplexer und iterativer, von Menschen durchgeführter Prozess. Aus diesem Grunde werden in der Planungsphase von Software-Produkten bereits seit geraumer Zeit grafische Notationen wie

z.B. die Unified Modeling Language (UML) eingesetzt. Komplexe Strukturen sind für den Menschen grafisch visualisiert deutlich besser zu verstehen als textuelle Notationen. Die Formulierung von Modellen in einer standardisierten Sprache verbessert deren Verständlichkeit, da ihnen eine definierte Semantik zugrunde liegt, und ermöglicht aus demselben Grund eine automatisierte, technische Nutzung, z.B. in Form von *Code-Generierung* oder *Validierung*.

Softwaresysteme unterliegen besonders in der Entwicklungsphase starken Veränderungen. Wenn Modelle nur zur Dokumentation eingesetzt und im Entwicklungsprozess nicht gepflegt werden, entstehen Inkonsistenzen zur Implementierung. Die Verwendbarkeit der Modelle nimmt mit fortschreitendem Entwicklungsstand immer stärker ab. Aus dieser Beobachtung heraus entwickelte sich das Konzept der *modellgetriebenen Software-Entwicklung*. Während früher Modelle nur zur Planung und Dokumentation einer Software dienten und dessen Implementierung zum Großteil manuell erfolgte, lässt sich Software nun auf einer abstrakten Ebene plattformunabhängig definieren und durch Modelltransformationen in mehr plattformspezifische Repräsentationen überführen, bis letzten Endes eine konkrete Implementierung entsteht, die interpretativ, oder auch exekutiv direkt auf der Zielplattform ausführbar ist [22]. *Round Trip Engineering* ermöglicht das synchronisierte Manipulieren von Modellen und dem zugehörigen Generat, wodurch Modelle und Implementierung in einem konsistenten Zustand gehalten werden.

1.2. Ein Beispiel: LearnLib Studio

Das Java Application Building Center (jABC)[1] wird vom Lehrstuhl für Programmiersysteme der Technischen Universität Dortmund entwickelt [23]. Es erlaubt die grafische Modellierung komplexer domänenspezifischer Anwendungen auf Grundlage von Datenflussmodellen. Deren Kernelemente, die Service Independent Building Blocks (SIBs), stellen eine in sich geschlossene, domänenspezifische Funktionalität bereit. Das Framework ist durch Plug-Ins erweiterbar, sodass die Modelle auch simuliert, ausgeführt und verifiziert werden können. Code-Generierung in Java-Klassen und Servlets wird ebenfalls durch Plug-Ins unterstützt.

LearnLib Studio[2] [14] ist eine auf dem jABC-Framework aufbauende Anwendung zur Analyse von Zustandsautomaten. Im Folgenden betrachten wir einige Aspekte der GUI von LearnLib Studio, an denen wir Möglichkeiten zur verbesserten Interaktion feststellen.

Abbildung 1.1 zeigt ein Analyse-Modell in LearnLib Studio während seiner Simulation. Das aktuell aktive SIB (SearchCounterexampleRandom) wird durch ein Icon markiert (grünes, kreisförmiges Symbol mit weißem Dreieck), und es wird gegebenenfalls in den sichtbaren Bereich bewegt. Der zur Aktivierung führende Datenfluss wird durch farblich hervorgehobene Kanten visualisiert. Interessant ist, dass kein *Scrolling* stattfindet, es wird zum aktiven SIB gesprungen. Ein Scrolling könnte an dieser Stel-

[1] http://www.jabc.de
[2] http://faelis.cs.tu-dortmund.de/learnlib.de/index.php

le den Anwender bei der Orientierung im Modell unterstützen. Des Weiteren findet auch kein *Zooming* statt. Dies hat zur Folge, dass gegebenenfalls zur Aktivierung führende Kanten außerhalb des sichtbaren Bereichs liegen. In dem gezeigten Beispiel sind nur zwei der Kanten sichtbar, obwohl jeweils vier hervorgehoben werden.

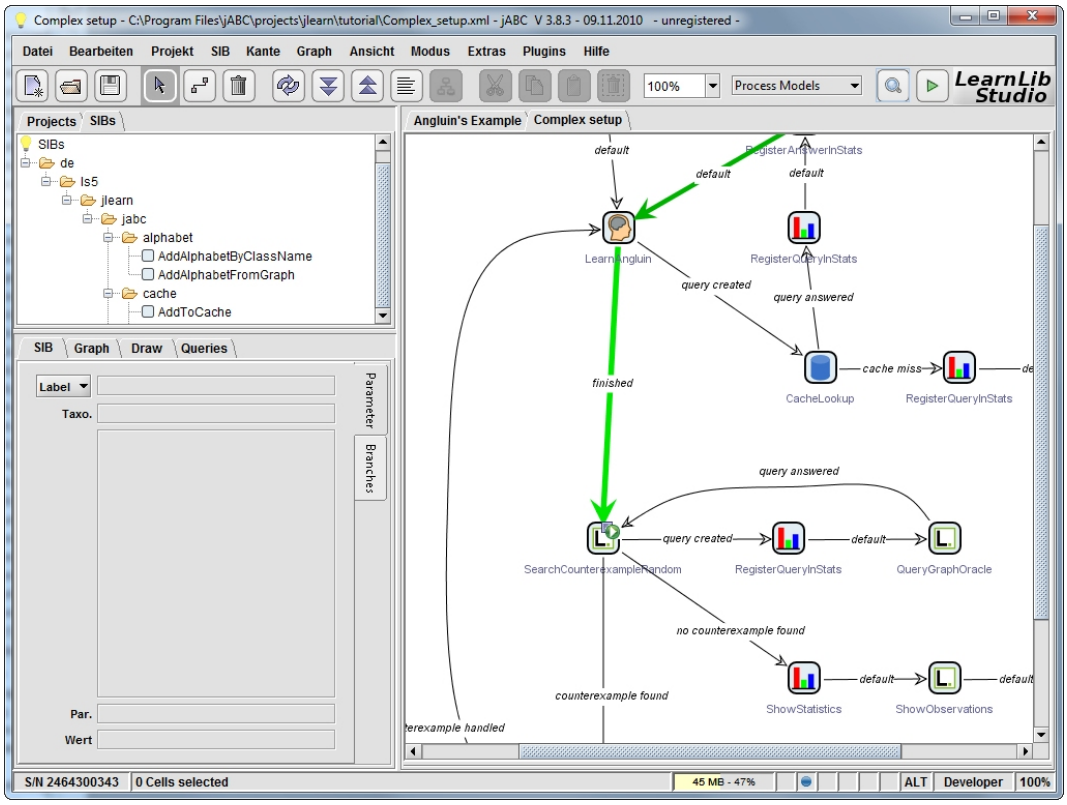


Abbildung 1.1.: Ein Modell in LearnLib Studio

Einen weiteren Ansatz zur Verbesserung bietet das Einfügen von SIBs. Der Anwender muss das einzufügende SIB aus der oberen linken Ansicht auswählen und kann es dann mit Drag and Drop oder mit einem Doppelklick in das Modell einfügen. Die Verknüpfung mit vorhandenen Modellelementen erfolgt manuell. An dieser Stelle ist eine Verbesserung möglich, indem z.B. automatisch Transitionen zu bevor selektierten SIBs erzeugt werden. Hier spielen noch andere Gesichtspunkte eine Rolle. Ein SIB verfügt im Allgemeinen über mehrere *Branches*, auf die, je nach Ausführungszustand des SIBs, der Kontrollfluss verzweigt. Daher muss eine Entscheidung getroffen werden, welchen Branches die eingefügten Kanten zugeordnet werden. Nach erfolgtem Einfügen des SIBs könnte für das Modell automatisch ein neues Layout berechnet werden, dies geschieht in LearnLib Studio manuell. In dem gezeigten Beispiel bestehen also durchaus Möglichkeiten, die Effizienz im Umgang mit Modellen zu verbessern.

1.3. Kiel Integrated Environment for Layout Eclipse Rich Client

Die Modellierung in der Ebene bietet viele neue Wege, mit den komplexen Strukturen heutiger Software effizient umzugehen. Die Herausforderung besteht in der *Pragmatik* der grafischen Werkzeuge. Die Pragmatik ist ein Forschungsgebiet der Sprachwissenschaften, das die Nutzung der Konstrukte einer Sprache untersucht. Fuhrmann und von Hanxleden [9] erweitern diesen Begriff im Kontext der grafischen Modellierung um praktische Aspekte in der Verwendung von Modellen. Wie effizient kann ein Benutzer Modellelemente einfügen, modifizieren oder löschen? Inwieweit unterstützt ein grafisches Werkzeug den Nutzer bei der Visualisierung des Modells sowohl im statischen Kontext während des Editierens als auch dynamisch während seiner Simulation? Diese Aspekte und die Qualität ihrer Umsetzung spielen eine wichtige Rolle für ein effizientes Arbeiten mit grafischen Modellen.

Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER) ist ein aktiv entwickeltes Forschungsprojekt der Arbeitsgruppe Echtzeitsysteme und Eingebettete Systeme der Christian-Albrechts-Universität zu Kiel, das sich mit der Pragmatik grafischer Modellierung befasst. Viele pragmatische Techniken finden in KIELER Anwendung. Als Beispiele dienen uns *Focus and Context* und *strukturbasiertes Editieren*, mit denen wir uns im Folgenden befassen.

Abbildung 1.2 verdeutlicht die Wichtigkeit eines guten *View Managements*. Sie stammt aus KIEL, dem Vorgänger von KIELER. Es wird Focus and Context verwendet, um während der Simulation eines SyncCharts-Modells [6] den Fokus auf den jeweils aktiven Makro-Zustand normal oder error umzuschalten. Der jeweils inaktive Zustand liegt im Kontext und wird nur minimiert dargestellt. Dabei erfolgt die Anpassung der *Sicht* auf das simulierte Modell automatisch und die Änderungen an der Sicht werden durch Animation deutlich gemacht. Dies unterstützt den Anwender dabei, seine *Mentale Karte* aufrechtzuerhalten.

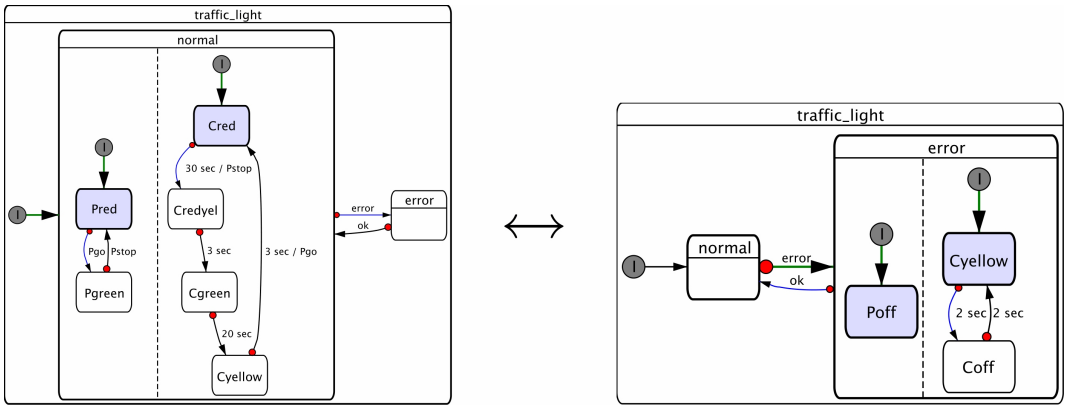


Abbildung 1.2.: View Management mit Focus and Context (Quelle: [9])

Abbildung 1.3 zeigt die Manipulation eines StateChart-Modells durch strukturbasiertes Editieren. Sie stammt ebenfalls aus KIEL. Die verwendeten Operationen "Füge Folge-Zustand hinzu" und "Füge Region hinzu" werden durch Modell-zu-Modell (M2M)-Transformationen auf dem semantischen Modell durchgeführt. Die notwendigen Sekundär-Operationen, wie z.B. das Erzeugen von Transitionen und deren Verknüpfung mit beteiligten Zuständen, werden dem Benutzer ebenfalls durch M2M-Transformationen abgenommen. Nachdem die Modifikation des semantischen Modells abgeschlossen ist, erzeugt das View-Management durch automatisches Layout eine neue Sicht auf das Modell, die die strukturellen Änderungen repräsentiert. Die grafischen Veränderungen werden ebenfalls durch Animation verdeutlicht. Dem Benutzer werden also zeitintensive, sekundäre Operationen abgenommen.

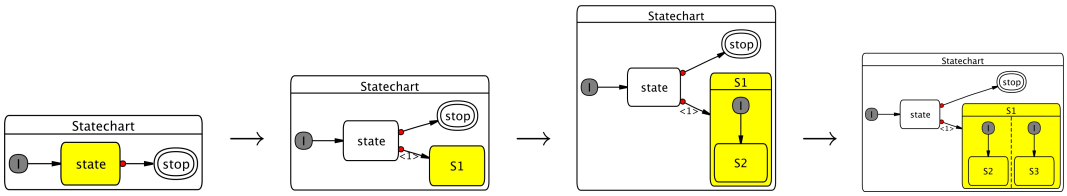


Abbildung 1.3.: Modellmanipulation mit strukturbasiertem Editieren (Quelle: [9])

Dem *Meta Layout* kommt dabei eine zentrale Bedeutung zu. Es verknüpft die verschiedenartigen Diagramme unter Verwendung einer abstrakten, strukturellen Darstellung mit den Layout-Algorithmen und ermöglicht so das automatische, parametrisierte Berechnen des Layouts der Diagramme. Auf Grundlage des automatischen Layouts wird die *Sicht* auf ein Modell nach einem Bearbeitungsschritt oder während einer Simulation angepasst. Das automatische Berechnen des Layouts nimmt nur eine geringe Zeit in Anspruch. Eine manuelle Anpassung dauert deutlich länger und lenkt den Fokus des Modellierers von seinem momentanen Aufgabenbereich ab.

Das Meta Layout von KIELER wird von dessen Teilprojekt KIELER Infrastructure for Meta Layout (KIML) realisiert und ist für diese Arbeit von besonderer Wichtigkeit, da KWebS auf dem Meta Layout basiert. In Abschnitt 3.3 gehen wir daher noch einmal im Detail auf KIML ein.

1.4. Layout von Graphen

Das Anfertigen des Layouts eines Diagramms ist kein trivialer Vorgang. Neben ästhetischen Gesichtspunkten spielt im Bereich technisch genutzter Diagramme vor allem die übersichtliche Darstellung der beteiligten Komponenten und deren Beziehungen zueinander eine zentrale Rolle. Vielen Diagrammen liegt eine Graphenstruktur zugrunde oder sie lassen sich in eine solche überführen. Daher spielen Algorithmen für das Berechnen des Layouts von Graphen eine wichtige Rolle für das Layout technisch genutzter Diagramme.

Für das Problem des Layouts von Graphen existieren vielfältige Lösungsansätze.

Kräftebasiertes Layout fasst einen Graphen als physikalisches System auf und weist den Knoten und Kanten auf geeignete Weise Kräfte zu. Eades [18] modelliert einen Graphen als mechanisches System, in dem Kanten durch mechanische Federn und Knoten durch Stahlringe repräsentiert werden. Das Layout des Graphen entsteht durch Berechnung des energetischen Minimums des Gesamtsystems. Orthogonales Layout richtet die Kanten eines Graphen parallel zu den Achsen der Zeichnungsebene aus [8]. Andere Ansätze strukturieren die Knoten eines Graphen in Ebenen (*hierarchisches Layout*), sodass deren gerichtete Kanten in einer Hauptrichtung orientiert sind, um den Daten- bzw. Kontrollfluss in einem Graphen hervorzuheben. *Zirkuläres Layout* wird häufig für netzwerkartige Strukturen verwendet.

Es lassen sich vielfältige Kriterien definieren, um die Qualität eines berechneten Layouts festzustellen, wie z.B. die Anzahl an Kantenüberschneidungen oder der Stress, der sich aus der Beziehung von struktureller Entfernung von Knoten und deren tatsächlicher euklidischer Entfernung im Layout ableitet. Viele dieser Kriterien sind gegensätzlicher Natur und das Berechnen eines Layouts bedeutet immer, eine Auswahl von beachteten Kriterien zu definieren, und eine geeignete Abwägung zwischen ihnen zu finden.

Neben der Verwendung von frei verfügbaren Bibliotheken wie Graphviz[3] oder dem Open Graph Drawing Framework (OGDF)[4] sind Layout-Algorithmen ein wesentliches Forschungsgebiet in KIELER. So wird z.B. im Bereich des Layouts portbasierter Diagramme, einem bislang wenig beachteten und beispielsweise für Datenflussmodelle relevanten Gebiet, aktiv geforscht. Diese Algorithmen sind nicht trivial und es steckt ein hoher Aufwand in deren Entwicklung. Das im Rahmen des Forschungsprojekts entwickelte automatische Layout ist zur Zeit primär nur innerhalb der von der Arbeitsgruppe entwickelten Modellierungsumgebung nutzbar. Graphical Modeling Framework (GMF)- und Graphiti-basierte Editoren in Eclipse können durch Installation entsprechender KIELER Features mit automatischem Layout ausgestattet werden[5].

1.5. Web Services

Essenziell für heutige Softwaresysteme ist die Möglichkeit, miteinander kommunizieren zu können. Aufgrund der im Allgemeinen vorherrschenden Heterogenität der beteiligten Kommunikationspartner ist es notwendig, die für die Kommunikation erforderlichen Schnittstellen und Datentypen unabhängig von einer konkreten technischen Basis zu definieren.

Bereits zu Beginn der 90er Jahre unternahm die Object Management Group (OMG) mit der Common Object Request Broker Architecture (CORBA) einen Versuch zur Standardisierung plattformunabhängiger Inter-Prozess-Kommunikation. CORBA hat sich jedoch nie in der Industrie durchsetzen können. Neben den hohen Lizensierungs-

[3] http://www.graphviz.org
[4] http://www.ogdf.net
[5] http://rtsys.informatik.uni-kiel.de/~kieler/updatesite/nightly

gebühren zu Beginn, als noch keine Open Source-Implementierungen vorhanden waren, waren es vor allem technische Mängel, wie das viel zu komplexe Application Programming Interface (API) und auch fehlende Sicherheitskonzepte, die eine breite Akzeptanz von CORBA verhinderten. Als wesentliche Ursache dafür führt Henning den Standardisierungsprozess der OMG an [15].

Auch Microsoft erkannte frühzeitig die Notwendigkeit plattformunabhängiger Inter-Prozess-Kommunikation und entwickelte das Distributed Component Object Model (DCOM). Für DCOM gab es jedoch niemals eine ausreichende Unterstützung für andere Plattformen als Windows und so konnte sich DCOM auch nicht als Standard durchsetzen.

In den letzten Jahren hat sich daher eine Technik etabliert, die im Kern auf offenen Standards wie der Extensible Markup Language (XML) und Transportschicht-Protokollen wie z.B. HTTP basiert: *Web Services*. Obwohl der Name es vermuten lässt, sind Web Services nicht an das Internet gebunden. Web Services bezeichnen allgemein Dienste, die ihre Funktionalität über eine beliebige Art von Netzwerk zur Verfügung stellen. Eine genaue Definition des Begriffs ist nicht gegeben. Das World Wide Web Consortium (W3C) spricht von einem Web Service wie folgt[6]:

> *A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.*

Die wesentlichen Komponenten eines Web Services sind nach dem W3C die *maschinell verarbeitbare* Schnittstelle, definiert in der Web Service Description Language (WSDL), und die Verwendung von SOAP[7] als Nachrichtenformat für die Kommunikation. Dadurch wird die notwendige Abstraktion von plattformspezifischen Details geschaffen, wodurch heterogene Kommunikation erst ermöglicht wird. Obwohl SOAP nicht an ein konkretes Transportschicht-Protokoll gebunden ist, wird üblicherweise HTTP verwendet, da der notwendige Port 80 in den allermeisten Fällen in Firewalls offen ist.

Aufgrund der intensiven Nutzung von XML ist diese Art Web Service recht "geschwätzig" und erzeugt einen recht großen zusätzlichen Aufwand. Daher hat sich im Umfeld der Web Services eine zweite Architektur entwickelt, die vermehrt Aufmerksamkeit erhält: Die *RESTful* Web Services, die der Representational State Transfer (REST)-Architektur von Roy Thomas Fielding folgen [21]. Diese "leichtgewichtigen" Web Services basieren im Gegensatz zu den XML-basierten Web Services nicht auf der Definition einer individuellen Schnittstelle. Die REST-Architektur setzt lediglich voraus, dass jede im Netzwerk bereitgestellte *Ressource* über eine einheitliche

[6] `http://www.w3.org/TR/ws-arch/`

[7] Seit Version 1.2 des Standards ist SOAP ein eigenständiger Begriff. Er wird nicht mehr als Akronym für "Simple Object Access Protocol" verwendet.

Schnittstelle erreichbar und manipulierbar ist. Im Kontext der RESTful Web Services wird die Schnittstelle meist aus den HTTP-Verben *POST*, *GET*, *PUT* und *DELETE* gebildet. Gelegentlich werden auch *HEAD* und *OPTIONS* unterstützt. Das Nachrichtenformat auf der Transportschicht ist beliebig wählbar und so kann auch ein leichtgewichtiges Format wie die JavaScript Object Notation (JSON) oder sogar Klartext verwendet werden, um den Aufwand zu minimieren. Abschnitt 3.4 befasst sich ausführlich mit Web Services und geht im Detail auf die beiden vorgestellten Architekturen ein.

1.6. Zielsetzung und Ausblick

Viele Desktop-Anwendungen verwenden derzeit Bibliotheken für das Layout ihrer Diagramme. Dies hat aus technischer Sicht eine starke Kopplung an die Schnittstelle einer Bibliothek bzw. an deren Modell-Notation zur Folge. Auch schränkt die Verwendung von Bibliotheken eine Anwendung auf die Algorithmen ein, die eine Bibliothek zur Verfügung stellt. Das KIELER Meta Layout ist dafür entwickelt worden, in einer heterogenen Landschaft von Graph-Notationen automatisches Layout zur Verfügung zu stellen. Es bildet einen Integrationspunkt für verschiedene Familien von Algorithmen, darunter die des KIELER Projekts und auch weit verbreitete, kostenfreie Produkte wie z.B. Graphviz.

KIELER ist ein in Java implementiertes Projekt, und als solches steht sein Layout bislang nur der Java-Plattform zur Verfügung. Das Ziel dieser Arbeit ist es, das KIELER Layout, insbesondere die von der Arbeitsgruppe entwickelten Algorithmen, auch Nutzern anderer Plattformen zur Verfügung zu stellen. Dazu verwendet KWebS einen Web Service, da er die notwendige Abstraktion von den beteiligten Plattformen bietet.

KWebS soll seitens der seinen Nutzern gebotenen Schnittstelle eine Menge an Standard-Notationen unterstützen, wie z.B. GraphML[8] oder DOT[9]. Damit wird Nutzern, die bereits über bibliothekbasiertes Layout verfügen, die Inanspruchnahme des dienstbasierten Layouts erleichtert. Des Weiteren ermöglicht KWebS seinen Nutzern damit den Zugriff auf eine heterogene Menge von Algorithmen ohne die Notwendigkeit, die verschiedenen von den beteiligten Bibliotheken erwarteten Graph-Notationen unterstützen zu müssen. Für einen Nutzer ist es ausreichend, eine der Notationen von KWebS zu verwenden. Eventuell notwendige Übersetzungen übernimmt KWebS.

Die Struktur dieser Arbeit ist wie folgt: Kapitel 2 gibt zunächst einen Überblick über verwandte Technologien und Produkte, bevor Kapitel 3 die im Rahmen dieser Arbeit verwendeten Technologien näher vorstellt. Kapitel 4 formuliert die Anforderungen an KWebS und stellt das Konzept vor, das für die Implementierung gewählt wird. Anschließend geht Kapitel 5 auf die Implementierung ein. Kapitel 6 beschreibt

[8] `http://graphml.graphdrawing.org/specification.html`
[9] `http://www.graphviz.org/doc/info/lang.html`

den Build-Prozess und die Installation von KWebS auf der für die Veröffentlichung verwendeten *virtuellen Maschine*. Darauf folgend bewertet Kapitel 7 KWebS anhand der gestellten Anforderungen, bevor Kapitel 8 diese Arbeit abschließt.

2. Verwandte Arbeiten und Produkte

Die Idee, automatisches Layout von Graphen in Form eines Dienstes anzubieten, ist nicht ganz neu. Bereits 1990 stellten Di Battista et al. mit *Diagram Server* ([3, 2]) eine Infrastruktur auf Basis des Client/Server-Modells zur Verwaltung und Visualisierung von Diagrammen vor. Das System sollte die Entwicklung diagrammorientierter Anwendungen durch die Entkopplung ihrer Anwendungslogik von der grafischen Benutzerschnittstelle erleichtern. Eine wesentliche Komponente des Diagram Server war das automatische Layout von Diagrammen. In diesem Sinne kann Diagram Server als der wahrscheinlich erste Layout-Dienst angesehen werden. Abschnitt 2.1 befasst sich im Detail mit dem Diagram Server und vergleicht ihn mit KWebS.

Bridgeman et al. veröffentlichten 1996 einen Dienst für das Zeichnen von Graphen und für die Übersetzung verschiedener Graph-Notationen ineinander [5]. In den Augen von Bridgeman et al. lagen die Vorteile des dienstbasierten Layouts in dessen Zentralisierung und der damit verbundenen Senkung des Wartungsaufwands. Standen neue oder verbesserte Implementierungen zur Verfügung, so war es ausreichend, den Dienst zu aktualisieren. Die Nutzer verfügten dadurch automatisch über die neuesten Versionen. Zu dieser Zeit spielten sicher auch Kosten eine Rolle. Das Berechnen von Layouts ist, je nach Komplexität eines Diagramms und des gewünschten Algorithmus, ein rechenintensiver Vorgang. Durch die Zentralisierung der benötigten Rechenkapazität auf einem Server konnte die Client-Hardware vergleichsweise günstig gehalten werden.

Der Großteil heutiger Anwendungen verwendet Bibliotheken wie Graphviz, OGDF, yFiles von yWorks[1] oder auch Tom Sawyer Layout[2], wenn sie auf automatisches Layout angewiesen sind. Dienstbasiertes Layout spielt heutzutage im Bereich von Rich Internet Applications (RIAs) eine Rolle. RIAs sind Anwendungen, die rein webbasiert durch Verwendung der Hyper Text Markup Language (HTML) und Javascript, oder auch in Form von Browser-Plugins, z.B. einem Adobe Flash-Plugin, in einem Web Browser ausgeführt werden. Eine RIA kann als plattformunabhängig angesehen werden. Die notwendigen Grundlagen wie Javascript oder entsprechende Erweiterungen für die Ausführung der Plugins sind praktisch in jedem aktuellen Browser vorhanden oder kostenfrei nachrüstbar. Eine RIA ist somit für einen Nutzer standortunabhängig erreichbar, ohne die Notwendigkeit der Installation einer Anwendung. Aufgrund der eingeschränkten Ressourcen in einem Browser wird die Berechnung des Layouts auf einen Dienst ausgelagert. Ein Beispiel hierfür ist das Web-based Interactive Graph Visualizations (WiGis)-Framework[3], auf welches Abschnitt 2.4 genauer eingeht.

[1] `http://www.yworks.com/de/products_yfiles_about.html`
[2] `http://www.tomsawyer.com/products/layout`
[3] `http://www.wigis.net`

Dienstbasiertes Layout wird häufig verwendet, um Diagramme in eine Web-Seite einzubetten. Zu diesem Zweck wird dem Dienst eine Beschreibung des Diagramms in textueller Form einer meist proprietären Sprache übermittelt. Er generiert daraus eine Grafik, die über ein HTML-Image-Element auf einfache Weise in eine Web-Seite eingebunden werden kann. Beispiele hierfür sind der WebDot-Dienst von Graphviz, auf den Abschnitt 2.3 genauer eingeht, yUML[4] oder auch WebSequenceDiagrams[5].

In diesem Kapitel lernen wir eine Auswahl von auf einem Layout-Dienst basierenden Infrastrukturen kennen und vergleichen sie mit KWebS.

2.1. Diagram Server

In den Augen von Di Battista et al. war die Entwicklung von grafischen Bedienoberflächen für Anwendungen, die auf Grundlage graphenbasierter, visueller Sprachen mit ihren Anwendern kommunizierten, ein aus Entwicklersicht immer wiederkehrender, kostenintensiver Prozess. Vielen der zeitgenössischen Anwendungen fehlte es an Funktionsumfang. So war z.B. die gleichzeitige Bearbeitung sowohl auf visueller Ebene als auch der textuellen Notation häufig nur eingeschränkt möglich. Änderungen in der visuellen Darstellung wurden nicht in die textuelle Notation übernommen. Die Synchronizität zwischen den Darstellungen war somit nicht gewährleistet. Des Weiteren war es im Allgemeinen nicht möglich, verschiedene Sichten desselben Modells gleichzeitig geöffnet zu haben. Ein automatisches Layout wurde von den meisten Anwendungen nicht unterstützt, wodurch die Übersichtlichkeit bei steigender Komplexität beeinträchtigt wurde. Obwohl sich konkrete Anwendungen und die von ihnen verwendeten Sprachen im Allgemeinen von einander unterschieden, ließen sich für die grafische Interaktion eine Menge von Grundoperationen definieren, die von praktisch allen Anwendungen unterstützt wurden. Diese Beobachtungen führten zur Entwicklung des Diagram Server.

Das Konzept des Diagram Server war es, eine generische, ausgereifte Benutzerschnittstelle für die Interaktion mit Diagrammen zur Verfügung zu stellen, die anwendungsbezogen angepasst werden konnte. In der Begriffswelt des Diagram Server beschrieb ein *Diagramm* die grafische Darstellung eines zugrundeliegenden *Graphen*. Ein Diagramm wurde in der Zeichnungsebene durch verschiedene *Repräsentationen* dargestellt. Die Benutzerschnittstelle war Bestandteil des *DS-User/Client-Interface* (siehe Abbildung 2.1, DS-UCI). Eine beliebige Anwendung dockte als *Client* an das DS-UCI an. Abhängig von der Art einer von einem Nutzer ausgeführten Operation führte das DS-UCI diese selbstständig aus, oder delegierte sie an den *DS-Server* (siehe Abbildung 2.1).

Ein Nutzer interagierte mit dem Client, und somit mit der eigentlichen Anwendung, über die Benutzerschnittstelle des DS-UCI. Die Operationen, die ein Nutzer ausführen konnte, wurden in vier Kategorien unterteilt:

[4] http://yuml.me
[5] http://www.websequencediagrams.com

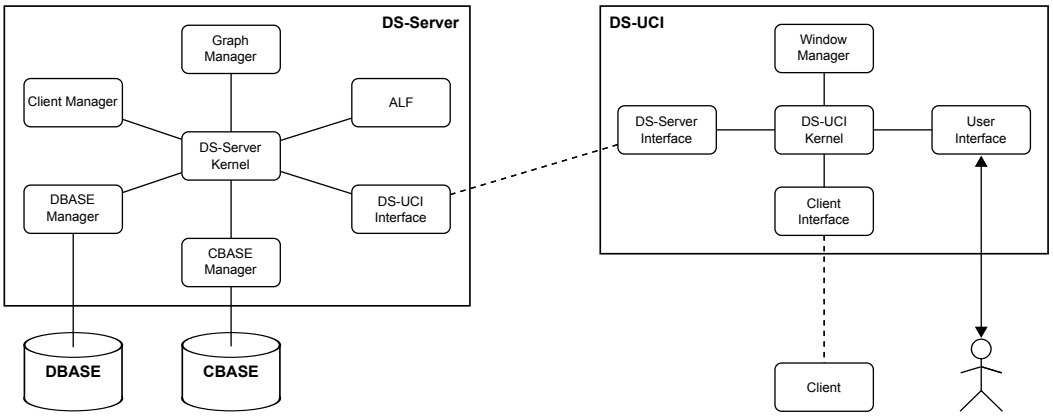


Abbildung 2.1.: Architektur des Diagram Server (nach [3])

- *Diagramm-erhaltende Operationen*, wie z.B. Zoomen, Scrollen oder das Wechseln auf ein anderes Fenster.
- *Graph-erhaltende Operationen*, wie z.B. das Bewegen von Knoten, veränderten das Diagramm, nicht aber den zugrundeliegenden Graphen.
- *Globale Operationen*, wie z.B. das Einfügen oder Löschen von Knoten, veränderten den einem Diagramm zugrundeliegenden Graphen. Aus diesem Grund mussten alle von ihm abgeleiteten Diagramme und letzten Endes auch alle zugehörigen Repräsentationen aktualisiert werden.
- *Auswahl-Operationen* waren seitens des Clients in eine der drei vorigen Operationen klassifizierbar, da die Interaktion mit einem Element des Diagramms anwendungsbezogen behandelt werden musste.

Die Benutzerschnittstelle war leichtgewichtig ausgelegt in dem Sinne, dass sie die Operationen auf Graphen und Diagrammen nicht selbstständig durchführte. Sie behandelte nur Diagramm-erhaltende Operationen. Alle struktur- oder darstellungsverändernden Operationen delegierte sie an den DS-Server. Auf ihm waren Graphen und Diagramme persistent hinterlegt. Er führte die angeforderten Operationen durch und propagierte alle daraus folgenden Änderungen durch *message passing* an das DS-UCI, und wenn erforderlich, auch an den Client.

Der DS-Server verfügte über eine *Automatic Layout Facility* (siehe Abbildung 2.1, ALF), die er für das Berechnen der Layouts der Diagramme verwendete. Sie verfügte über eine reichhaltige Menge von Algorithmen und war durch den Client auf die Bedürfnisse in Bezug auf die gewünschte Darstellung konfigurierbar. In diesem Sinne war der DS-Server ein Layout-Dienst, der verschiedenartigen Anwendungen, den Clients, automatisches Layout ihrer Diagramme bereitstellte. Das Layout wurde auf Grundlage der serverseitig gespeicherten Graphen durchgeführt. Es basierte

daher auf einer proprietären Darstellung. In diesem Punkt unterscheidet sich der Diagram Server von KWebS, da KWebS das automatische Layout auf Grundlage verschiedener, standardisierter Notationen wie z.B. GraphML zur Verfügung stellt. Des Weiteren war der Diagram Server als Plattform zur effizienten Entwicklung diagrammorientierter Anwendungen konzipiert. Zwischen den nutzenden Anwendungen und Diagram Server bestand aufgrund der proprietären Schnittstelle eine enge Kopplung. Somit stand das automatische Layout nur den Anwendungen zur Verfügung, die als Client des Diagram Server implementiert waren. Es war aber nicht über eine öffentliche Schnittstelle nutzbar. Dies ist ein weiterer konzeptioneller Unterschied, da die öffentliche Verfügbarkeit ein primäres Ziel von KWebS ist.

2.2. Graph Drawing and Translation Service

Der Graph Drawing and Translation Service von Bridgeman et al. kann als enger Verwandter von KWebS betrachtet werden. Er bot zwei Funktionen:

- Ein Nutzer konnte einen Graphen zeichnen lassen. Er konnte einen Graphen in einer von mehreren unterstützten Notationen an den Dienst übermitteln und definieren, welcher Algorithmus zum Zeichnen verwendet werden sollte. Er konnte ebenso festlegen, in welcher Notation das Ergebnis an ihn geliefert werden sollte.
- Ein Nutzer konnte den Dienst verwenden, um einen Graphen in eine andere Notation zu übersetzen.

In der Begriffswelt des Graph Drawing and Translation Service bedeutete das *Zeichnen* eines Graphen die Berechnung dessen Layouts und war nicht damit zu verwechseln, eine konkrete Grafik zu erzeugen. Die vom Nutzer gewünschte Darstellung konnte ein Grafikformat sein, aber genauso einer textuellen Notation entsprechen.

Die Interaktion mit dem Dienst fand über ein Java-Applet oder eine formularbasierte Web-Seite statt. Das Applet bot neben der textuellen Definition eines Graphen auch einen interaktiven, visuellen Editor. Es könnte tatsächlich die erste diagrammorientierte RIA der Geschichte des World Wide Web (WWW) gewesen sein. Die Web-Seite bot neben der textuellen Definition die Möglichkeit, einen Graphen in Form eines Uniform Resource Locator (URL) zu definieren, mit der die Notation seitens des Dienstes heruntergeladen wurde. In beiden Fällen musste der Nutzer festlegen, in welcher Notation der Graph vorlag, und in welcher Notation dessen Zeichnung an ihn übermittelt werden sollte.

Die Architektur des Graph Drawing and Translation Service ist in Abbildung 2.2 dargestellt. Sie zeigt die beiden Clients, die über das WWW mit dem Server kommunizierten. Die Kernkomponente des Servers war der *Manager*. Er nahm Anfragen entgegen und koordinierte, je nach Art der Anfrage, den *Parser*, den *Translator* und den *Graph Drawer*. Der Parser bereitete die für die Bearbeitung der Anfrage notwendigen Informationen auf. Anschließend verwendete der Manager, wenn erforderlich,

den Translator, um entweder die vom Nutzer gewünschte Übersetzung durchzuführen, oder den vom Nutzer übermittelten Graph in die für den gewählten Algorithmus notwendige Notation zu übersetzen. Wenn ein Nutzer eine Zeichnung anfragte, wurde sie durch den Graph Drawer erzeugt. Abschließend wurde möglicherweise noch einmal der Translator verwendet, um die Zeichnung in die vom Nutzer gewünschte Notation zu übersetzen.

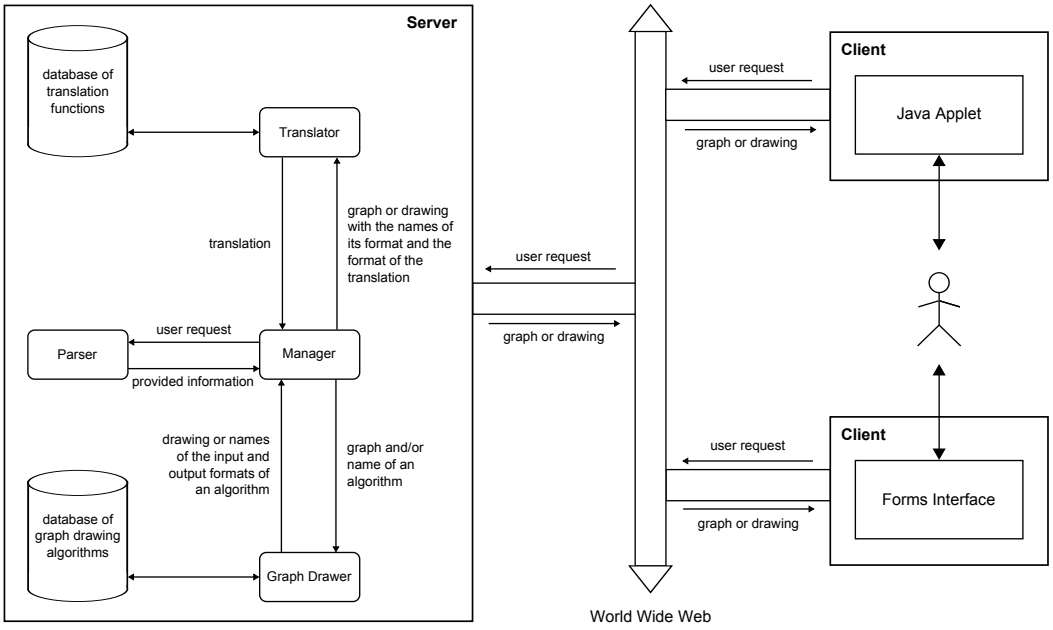


Abbildung 2.2.: Architektur des Graph Drawing and Translation Service (nach [5])

Auf architektureller Ebene gleicht der Graph Drawing and Translation Service der Struktur von KWebS. Die verwendete Web Service-Plattform (siehe Abschnitt 3.5) ist mit dem Parser vergleichbar. Beide Komponenten dienen der Übersetzung der in der Anfrage eines Nutzers enthaltenen Daten in ein anwendungsspezifisches Format. Der Manager entspricht der Implementierung des für KWebS verwendeten Web Service. KWebS unterstützt des Weiteren verschiedene Notationen wie z.B. DOT, die er in die intern verwendete Notation übersetzt und nach erfolgtem Layout in die Notation, die vom Nutzer für das Ergebnis gewünscht ist. Dies entspricht dem Translator. Der Graph Drawer entspricht dem KIELER Meta Layout, welches KWebS für das Berechnen der Layouts verwendet.

Der Unterschied liegt im gedachten Anwendungsgebiet. KWebS ist für die Nutzung von beliebigen Software-Systemen gedacht, während der Graph Drawing and Translation Service konkrete Clients für seine Nutzung zur Verfügung stellte. Die Grundidee von Bridgeman et al. bei der Entwicklung war es, eine Experimentierplattform für die sich damals rasch entwickelnde Anzahl von Algorithmen für das Layout von Graphen anzubieten. Eine Bibliothek zur programmatischen Nutzung

war von den Entwicklern des Graph Drawing and Translation Service vorgesehen. Es ist nicht klar, inwieweit diese realisiert wurde, und ob der Graph Drawing and Translation Service jemals von anderen Anwendungen als den beschriebenen Clients verwendet wurde.

2.3. Graphviz WebDot

Das Graphviz-Projekt stellt mit WebDot einen Dienst zur Verfügung[6], der das Layout von Graphen ermöglicht. Allerdings ist dieser Dienst nicht öffentlich nutzbar und nur noch zu Demonstrationszwecken online. Für die Verwendung von WebDot muss ein eigenständiger Server eingerichtet und das frei verfügbare WebDot-Paket auf ihm installiert werden[7].

Ein Nutzer übermittelt den Graphen, für den er ein Layout benötigt, in der Graphviz-eigenen Notation DOT. Nach erfolgtem Layout liefert WebDot dem Nutzer das Ergebnis in einer von mehreren möglichen, von ihm wählbaren Darstellungen. Neben DOT bietet WebDot vor allem grafikbasierte Darstellungen wie z.B. Portable Network Graphics (PNG), als JPEG kodiert und das Portable Document Format (PDF) an. WebDot ist primär darauf ausgelegt, Graphen-Darstellungen auf einfache Weise in Web-Seiten einbinden zu können. Zu diesem Zweck muss ein Nutzer den Graphen als Datei auf einem Server öffentlich verfügbar machen. Das Layout erhält er, indem er einen Aufruf an WebDot in ein HTML-*Image*-Element einbettet (siehe Listing 2.1).

```
<img src="http://service.com/webdot/http://client.com/graph.dot.dot.png"/>
```

Listing 2.1: In eine Web-Seite eingebettete Anfrage an WebDot

In dem gezeigten Beispiel läuft ein WebDot-Dienst auf dem Host service.com unter dem Kontext webdot. Die Layout-Anfrage besteht aus dem auf die Dienst-URL folgenden Bestandteil der gesamten URL: http://client.com/graph.dot.dot.png. Dieser Teil der Anfrage enthält einen Verweis auf eine Datei, die den Graphen enthält (http://client.com/graph.dot), den zu verwendenden Layout-Algorithmus (dot) und das gewünschte Ausgabeformat (png). Grafische Darstellungen von Graphen können so dynamisch erzeugt in Web-Seiten dargestellt werden. Abbildung 2.3 zeigt das durch das obige Image-Element erzeugte Bild, wobei die referenzierte Datei die in Listing 2.2 dargestellte Notation des Graphen enthält.

Eine weitere Option bei der Einbettung in Web-Seiten ist das Berechnen von *Image-Maps*. Dazu werden die Knoten und Kanten des Eingabe-Graphen mit URLs annotiert (siehe Listing 2.2).

[6] http://www.graphviz.org/webdot/index.html
[7] http://www.graphviz.org/Download_source.php

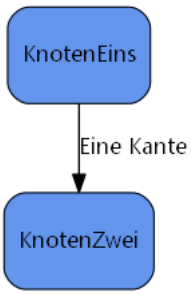


Abbildung 2.3.: Ein mit WebDot erzeugtes Layout in Form eines Bildes

```
digraph G {
  graph [ URL="Daneben.html" ]
  node [
    fontname="Segoe UI" fontsize=13 height=0.75 penwidth=0.75
    shape=box style="rounded,filled" fillcolor=cornflowerblue
  ]
  edge [
    fontname="Segoe UI" fontsize=13
  ]
  KnotenEins [ URL="KnotenEinsGeklickt.html" ]
  KnotenZwei [ URL="KnotenZweiGeklickt.html" ]
  KnotenEins -> KnotenZwei [
    URL="KanteGeklickt.html"
    label="Eine Kante"
  ]
}
```

Listing 2.2: Ein DOT-Graph mit Referenzen

Zum Erzeugen der Grafik und der Image-Map ruft ein Nutzer WebDot zweimal auf (siehe Listing 2.3). Das umschließende HTML-*Anchor*-Element verwendet die Image-Map, um den im erzeugten Bild dargestellten Knoten und Kanten Referenzen zuzuweisen.

```
<a href="http://service.com/webdot/http://client.com/graph.dot.dot.map">
  <img src="http://service.com/webdot/http://client.com/graph.dot.dot.png" ismap/>
</a>
```

Listing 2.3: WebDot-Anfrage mit Image-Map

Die im Anchor-Element verwendete Referenz definiert als Ausgabeformat `map`. Der WebDot-Dienst erzeugt somit eine HTML-konforme Image-Map. Das generierte Bild kann auf den Knoten und Kanten angeklickt werden. Der Browser erzeugt eine

Anfrage an den Web-Server, die die angeklickten Koordinaten enthält. Der Server wertet diese Koordinaten aus und leitet die Anfrage auf die entsprechende Web-Seite weiter. Auf diese Weise kann eine Navigation in einem Graphen realisiert werden.

Das primäre Anwendungsfeld von WebDot ist das Erzeugen grafischer Darstellungen von Graphen. Eine technische Nutzung innerhalb eines Software-Systems ist möglich, aber auf die Graphviz-eigene Notation DOT und die Algorithmen von Graphviz beschränkt. Des Weiteren kann ein Graph nicht einfach übermittelt werden, sondern muss als Datei auf einem Server verfügbar gemacht werden. Dies erhöht den Aufwand für eine technische Nutzung. Der Graphviz-eigene WebDot-Server ist nur noch zu Demonstrationszwecken online und kann nicht mehr mit externen Ressourcen verwendet werden. Somit muss zur Nutzung von WebDot ein eigenständiger Server aufgesetzt werden.

2.4. Web-based Interactive Graph Visualizations

WiGis ist ein von Gretarsson et al. entwickeltes Framework für die webbasierte Visualisierung von und Interaktion mit Graphenstrukturen [10]. Eines der Hauptprobleme nach Grettarson et al. von heutigen webbasierten Frameworks ist deren Skalierung bei Graphen mit einer großen Anzahl an Elementen. Hauptziel von WiGis ist es daher, auch bei komplexen Graphen gut zu skalieren.

Nach Gretarsson et al. verwenden viele der heute gebräuchlichen Frameworks generierte Bilder für die clientseitige Darstellung der Sicht und für die Interaktion mit dem Graphen. Zentraler Punkt ist daher die Bildrate, mit der die Sicht des Graphen clientseitig aktualisiert wird. Sie ist direkt abhängig von Faktoren wie der Zeit, die die Berechnung einer Sicht (inklusive automatischem Layout) in Anspruch nimmt und der Zeit, die eine Anfrage für den Transport über das Netzwerk benötigt. Von der Bildrate hängt nicht nur ab, wie schnell die Sicht nach einer erfolgten Modifikation aktualisiert wird. Auch die Pragmatik ist von der Bildrate direkt abhängig. Wird z.B. ein Graphelement dadurch verschoben, dass der Anwender es auswählt und bei gedrückter Maustaste verschiebt, so sollte das Element der Mausbewegung direkt folgen. Eine niedrige Bildrate hat zur Folge, dass das Element in der Sicht springt, wodurch für einen Anwender die Nachvollziehbarkeit seines Handelns verringert wird.

Die in WiGis verwendete Architektur (siehe Abbildung 2.4) basiert auf dem Client/Server-Modell. Es werden zwei Betriebsarten unterstützt: der *Server-Modus* und der *Client-Modus*. Die Umschaltung zwischen ihnen ist abhängig von der Größe des in der Sicht dargestellten Teilgraphen und für den Anwender transparent. Liegt die Größe des Graphen unter einer definierten Grenze, so wird der Client-Modus verwendet, andernfalls der Server-Modus. In beiden Betriebsarten müssen sowohl die Interaktion des Anwenders mit der Sicht als auch deren Generierung programmatisch gehandhabt werden. Daraus folgt, dass die dafür notwendigen Algorithmen zweifach implementiert werden müssen. Das bedeutet einen erheblichen Aufwand, da dies auch die für die Generierung der Sicht notwendigen Layout-Algorithmen

betrifft, und sowohl deren server- als auch clientseitigen Versionen stets identische Ergebnisse liefern müssen.

Auf Serverseite liegt immer das vollständige Graphenmodell vor, während auf Clientseite meist nur der für die Erzeugung der aktuellen Sicht notwendige Teilgraph vorhanden ist. Die Synchronisation zwischen beiden Modellen bei Interaktion mit der Sicht wird über Asynchronous Java and XML (AJAX)-Anfragen an den Server realisisert.

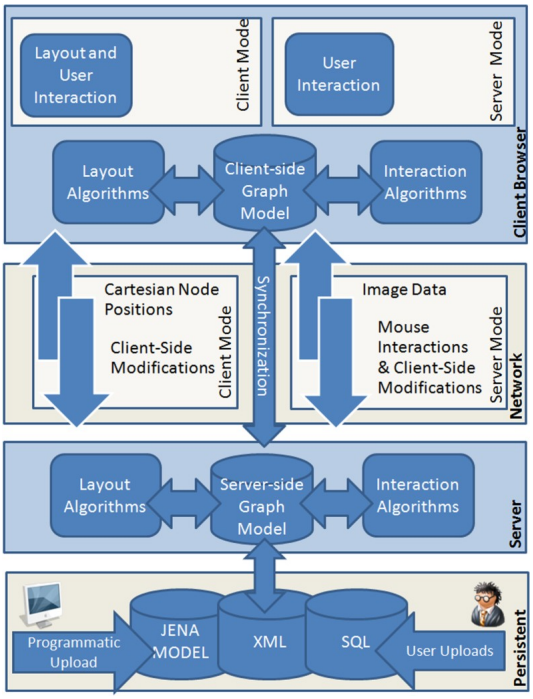


Abbildung 2.4.: Architektur von WiGis (Quelle: [10])

Für die Interaktion mit dem Graphen werden primär Mausgesten verwendet, die auf dessen Sicht angewendet werden. Die Sicht des Graphen wird dabei durch eine Grafik realisiert, die im Server-Modus regelmäßig über AJAX-Anfragen des Clients an den Server aktualisiert wird. Im Client-Modus wird die Sicht aus dem auf dem Client zwischengespeicherten Teilgraphen als eingebettete Scalable Vector Graphics (SVG)-Grafik neu generiert.

Aus Sichtweise der Entwickler von WiGis bietet die gewählte Architektur den Vorteil, dass Serverkapazitäten nur dann verwendet werden, wenn der im Client bearbeitete Graph von so hoher Komplexität ist, dass die Kapazitäten des Clients nicht mehr ausreichend sind für eine pragmatisch adäquate Interaktivität. Für Graphen bis zu einer bestimmten Komplexität finden sämtliche notwendigen Berechnungen clientseitig statt. Dies schont die Ressourcen des Servers und bietet dem Anwender ein in hohem Maße pragmatischen Umgang mit dem Graphen. Der Server-Modus

wird nur für die Berechnung komplexer Graphen verwendet.

WiGis ist ein webbasiertes Framework für die interaktive Visualisierung von netzwerkartigen Strukturen und als solches nur bedingt mit KWebS vergleichbar. Die Nutzung des serverseitigen Layouts ist eng gekoppelt mit der RIA. Die von der RIA dargestellten Graphen sind serverseitig gespeichert oder können von einem Nutzer auf den Server hochgeladen werden. Die Interaktion mit dem Server basiert auf dem Austausch von Grafiken und einer proprietären Notation für die Übermittlung der vom Nutzer vorgenommenen Änderungen am Graphen. Eine technische Nutzung in einer anderen Anwendung als der RIA des Frameworks ist somit nicht möglich.

2.5. Zusammenfassung

Dieses Kapitel stellt eine Auswahl von Frameworks vor, die ein dienstbasiertes Layout von Graphen verwenden oder verwendet haben. Der Diagram Server und der Graph Drawing and Translation Service sind recht alte Produkte und als solche heutzutage nicht mehr verfügbar. Der Diagram Server ist nur bedingt mit KWebS vergleichbar, da er als Anwendungsplattform konzipiert war. Der Graph Drawing and Translation Service war praktisch ein Vorgänger von KWebS, auch wenn er als Experimentierplattform ausgelegt war und nicht direkt zur Nutzung von beliebigen Softwaresystemen. Dies war aufgrund seiner HTTP-basierten Schnittstelle jedoch prinzipiell möglich. Graphviz WebDot ist das einzige dem Autor bekannte Framework, welches aus technischer Sicht von beliebigen Anwendungen verwendet werden kann. Kostenpflichtige Produkte wie yFiles oder Tom Sawyer Layout bieten ebenfalls die Möglichkeit, ihr Layout mithilfe eines Dienstes anzubieten. Auf diese oder ähnliche Produkte geht diese Arbeit nicht ein, da sie aufgrund der Lizensierung nicht öffentlich verfügbar gemacht werden können. Die Einschränkungen von WebDot liegen in der einzig unterstützten Notation (DOT), den für das Layout wählbaren Algorithmen (Graphviz) und der Tatsache, dass ein eigener Dienst aufgesetzt werden muss, da der einzig öffentlich verfügbare Dienst das Layout externer Ressourcen nicht unterstützt. WiGis ist ein interessantes Beispiel für eine eng an den verwendeten Dienst gekoppelte RIA. Auch hier ist der Dienst nicht von beliebigen Softwaresystemen technisch nutzbar, da er das berechnete Layout in Form von Grafiken liefert.

Die von diesem Kapitel vorgestellten Beispiele für dienstbasiertes Layout dienen dem Überblick, wie es heutzutage eingesetzt wird. Dem Autor dieser Arbeit ist kein öffentlich verfügbarer, kostenfreier Dienst bekannt, der dem in dieser Arbeit vorgestellten Layout-Dienst KWebS gleicht.

3. Verwendete Technologien

In diesem Kapitel werfen wir einen Blick auf die im Rahmen dieser Arbeit verwendeten Technologien. Wir beginnen in Abschnitt 3.1 mit der Eclipse-Plattform, da diese zentraler Bestandteil des KIELER Forschungsprojekts und auch von KWebS ist. Danach betrachten wir in Abschnitt 3.2 das Eclipse Modeling Framework (EMF). Von zentraler Bedeutung für das automatische Layout in KIELER ist dessen Meta Layout, realisiert durch KIML. KWebS verwendet es ebenfalls für die Berechnung des Layouts, daher lernen wir das Meta Layout in Abschnitt 3.3 genauer kennen. Darauf folgend stellt uns Abschnitt 3.4 die beiden Web Service-Architekturen vor, welche für die Umsetzung von KWebS in Betracht gezogen werden. Wie wir im Verlauf dieser Arbeit sehen werden, bietet KWebS das dienstbasierte Layout auf Grundlage eines SOAP Web Service an. Das dazu verwendete Framework, die Referenzimplementierung der Java API for XML - Web Services (JAX-WS), betrachten wir in Abschnitt 3.5. Abschließend lernen wir in Abschnitt 3.6 die Java Electronic Tool Integration (jETI)-Plattform kennen, welche KWebS für eine alternative Umsetzung des Layout-Dienstes verwendet.

3.1. Die Eclipse-Plattform

Es gibt viele Möglichkeiten, die Eclipse-Plattform zu beschreiben. Für viele ist Eclipse eine integrierte Entwicklungsumgebung für verschiedene Sprachen wie z.B. Java, HTML, XML oder auch C++. Vom technischen Standpunkt betrachtet ist Eclipse eine Plattform, die die Entwicklung beliebiger Anwendungen auf Basis eines modularen und erweiterbaren Komponentenmodells ermöglicht. Jede funktionale Erweiterung der Plattform erfolgt auf Basis von Komponenten, den *Plug-Ins*. Die Anbindung neuer Funktionalität geschieht dabei mithilfe des *Extension-Point-Mechanismus*. Die dafür notwendige Laufzeitumgebung bestand ursprünglich aus einer proprietären Implementierung. Seit Version 3 verwendet die Eclipse-Plattform den Open Services Gateway initiative (OSGi)-Standard. Die Eclipse-Implementierung *Equinox* gilt als dessen Referenzimplementierung.

Die Entwicklung von Anwendungen auf Grundlage von Komponenten bietet viele Vorteile, von denen dieser Abschnitt nur auf die im Kontext der Eclipse-Plattform relevanten eingeht. Ein guter Überblick findet sich im technischen Bericht der OSGi-Alliance [17].

3.1.1. Plug-Ins

Ein Plug-In ist ein Container, der beliebige Ressourcen wie z.B. HTML-Quelltext, Bitmaps, Icons und insbesondere auch ausführbaren Code enthalten kann. Es kann seine Abhängigkeiten zu anderen Plug-Ins, seine Beiträge zur Plattform und auch seine Schnittstellen zum "andocken" für andere Plug-Ins *explizit* definieren. Diese Metadaten sind in den Dateien MANIFEST.MF und plugin.xml eines Plug-Ins hinterlegt. Diese Dateien sind von der Laufzeitumgebung auswertbar, ohne Code des Plug-Ins laden und ausführen zu müssen. Das ermöglicht die dynamische Bindung von Plug-Ins. Sie können somit zur Laufzeit einer Anwendung entfernt, ausgetauscht oder hinzugefügt werden. Daher können Anwendungen zur Laufzeit aktualisiert werden. Im Vergleich zu monolithischen Anwendungen verringert sich die Größe des Updates, da Plug-Ins versioniert sind, und somit nur aktualisierte Plug-Ins ausgetauscht werden müssen. Ein Beispiel ist der Update-Mechanismus von Eclipse. Die Versionierung von Plug-Ins ermöglicht des Weiteren, dass verschiedene Versionen eines Plug-Ins zur Laufzeit nebeneinander existieren können. Ein Plug-In kann festlegen, welche konkrete Version eines anderen Plug-Ins es für seine Ausführung benötigt. Auf diese Weise kann die Laufzeitumgebung Inkompatibilitäten zwischen Plug-Ins auflösen.

Ein zentrales Konzept von Eclipse ist das *lazy loading*. Lazy loading bedeutet die Bereitstellung beliebiger Ressourcen, seien es Instanzen von Java-Klassen oder Icons für die UI, auf den spätest möglichen Zeitpunkt zu verlegen. Es vermeidet im Vergleich zu monolithischen Anwendungen den Aufwand für die Initialisierung potenziell nicht benötigter Ressourcen. Dies kommt dem Laufzeitverhalten und Ressourcenbedarf einer Anwendung zugute. Die Laufzeitumgebung folgt diesem Konzept, denn sie kann das Laden von Ressourcen eines Plug-Ins anhand seiner Metadaten auf den Zeitpunkt verlegen, zu dem sie tatsächlich benötigt werden.

3.1.2. Extension-Point-Mechanismus

Ein wichtiges über die OSGi-Spezifikation hinausgehendes Konzept der Eclipse-Plattform ist der Extension-Point-Mechanismus. Ein Plug-In kann *Extension-Points* definieren, die andere Plug-Ins durch Definition einer entsprechenden *Extension* erweitern können. Der Extension-Point stellt einen Vertrag dar zwischen dem definierenden und den erweiternden Plug-Ins, indem er festlegt, welche Informationen ein Plug-In für die Erweiterung bereitstellen muss. Auf diese Weise kann ein Plug-In Daten und Code einem anderen Plug-In zur Verfügung stellen. Dies ermöglicht die Erweiterung von Anwendungen um Funktionalität, ohne dass eine erneute Übersetzung der gesamten Anwendung erforderlich wird. Es ist ausreichend, das die Extension definierende Plug-In zu übersetzen. Ein Plug-In kann beliebig viele Extension-Points definieren und auch beliebig viele Extensions zu unterschiedlichen Plug-Ins (siehe Abbildung 3.1).

Ein Beispiel ist das Plug-In org.eclipse.ui. Es ist in Eclipse unter anderem für die Handhabung der *Preference Pages* zuständig. Es definiert den Extension-Point org.eclipse.ui.preferencePages. Er kann von anderen Plug-Ins dazu verwendet wer-

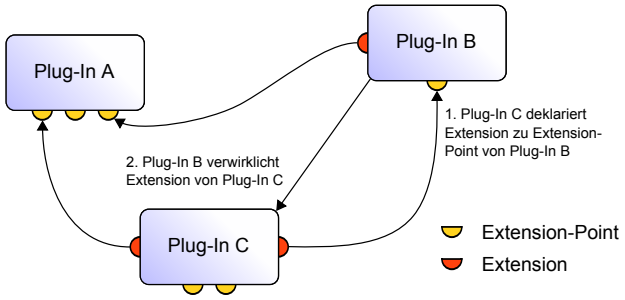


Abbildung 3.1.: Der Extension-Point-Mechanismus von Eclipse

den, dem Nutzer anwendungsbezogene Optionen zur Konfiguration zur Verfügung zu stellen. org.eclipse.ui koordiniert die Anzeige und Handhabung aller definierten Preference Pages. Ein den Extension-Point erweiterndes Plug-In ist lediglich für die Anzeige und Handhabung der Optionen zuständig, die seine Preference Pages bereitstellen.

Die Informationen über die Bereitstellung und Nutzung von Extension-Points sind in der Datei plugin.xml eines Plug-Ins hinterlegt. Die Eclipse-Plattform wertet diese aus und stellt die enthaltenen Informationen der laufenden Anwendung über die *Extension Registry* zur Verfügung. Wie Abbildung 3.2 zeigt, ist das definierende Plug-In für die Auswertung der Extensions anhand der Extension Registry verantwortlich.

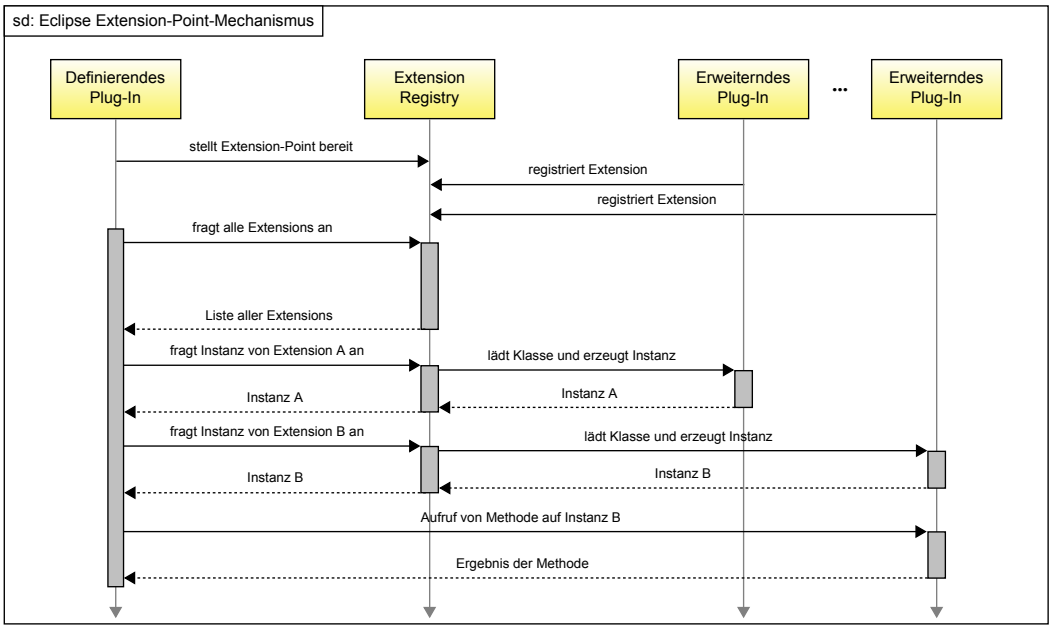


Abbildung 3.2.: Interaktion von Plug-Ins durch die Extension-Registry

3.1.3. Eclipse Rich Client Platform

Eclipse war schon immer auf einem Komponentenmodell aufgebaut. Bis einschließlich Version 2 war Eclipse als IDE konzipiert und verwendete ein proprietäres Komponentenmodell. Viele Entwickler stellten jedoch fest, dass sich Eclipse auch als Basis für die Implementierung beliebiger Anwendungen verwenden ließ. Im Allgemeinen mussten jedoch manuelle Anpassungen an der Implementierung von Eclipse vorgenommen werden, um z.B. nicht benötigte, auf die Entwicklung von Software bezogene Elemente aus der grafischen Benutzerschnittstelle zu entfernen. Dies führte zu der Idee, Eclipse in eine Rich Client Platform (RCP) zu überführen[1].

Seit Version 3 verwendet Eclipse den OSGi-Standard und ist modular aufgebaut. Abbildung 3.3 gibt einen Überblick über den Aufbau der Eclipse-Plattform. Im Kern besteht sie aus der Eclipse-RCP, alle weiteren Komponenten werden anwendungsbezogen hinzugefügt wie z.B. die Unterstützung für teambasiertes Arbeiten, die Java Development Tools (JDT) oder die Plug-In Development Environment (PDE).

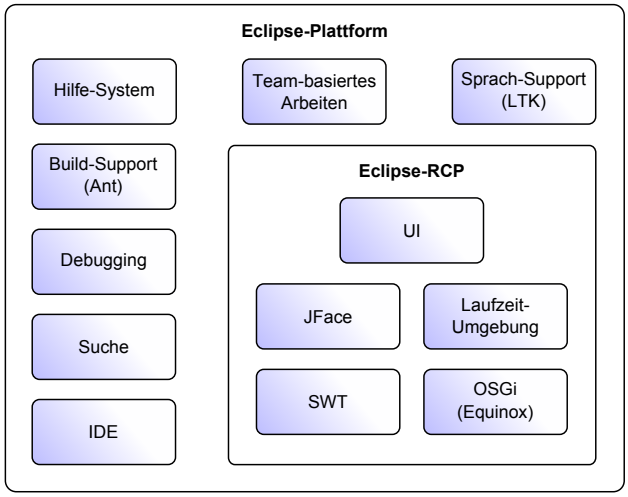


Abbildung 3.3.: Die Eclipse-Plattform (nach [11])

Die Eclipse-RCP bildet einen minimalen Kern von Komponenten, die im Allgemeinen für die Entwicklung einer beliebigen Anwendung benötigt werden. Für Anwendungen ohne GUI sind z.B. deren Komponenten JFace, SWT und UI nicht erforderlich.

3.2. Eclipse Modeling Framework

Wie die Einleitung in Abschnitt 1.1 beschreibt, ist die modellgetriebene Softwareentwicklung der aktuelle Stand in der Entwicklung von Softwaresystemen. Das EMF

[1] https://bugs.eclipse.org/bugs/show_bug.cgi?id=36967

ist in der Eclipse-Welt der akzeptierte Standard für die Entwicklung von Anwendungen basierend auf *strukturierten* Modellen. Einem strukturierten Modell liegt ein weiteres Modell zugrunde, welches eine Menge von Objekten definiert und die Art, wie diese zueinander in Beziehung stehen können. Dieses *Metamodell* definiert somit die Syntax der von ihm abgeleiteten Modelle, und jedes Modell das dieser Syntax entspricht ist eine gültige Instanz des Metamodells.

Das grundlegende Metamodell in EMF ist *Ecore* (siehe Abbildung 3.4). Jedes in EMF definierte Modell ist von Ecore direkt oder indirekt abgeleitet. Ecore ist in sich selbst definiert, um einen Abschluss in der Hierarchie zu bilden. Dies ist analog zur Meta Object Facility (MOF), in der die M3-Ebene den Abschluss bildet. Tatsächlich begann EMF als Implementierung der MOF-Spezifikation, hat sich jedoch in ein eigenständiges Projekt entwickelt[2].

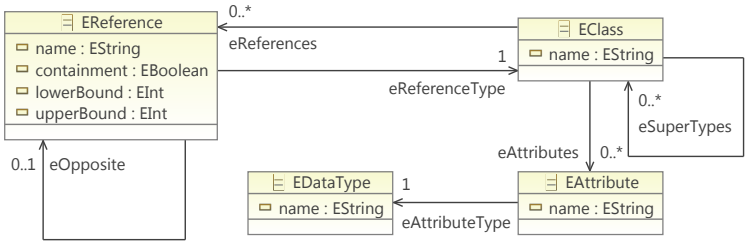


Abbildung 3.4.: Das Ecore-Metamodell (vereinfachte Darstellung)

EMF erlaubt die Spezifikation von Modellen in mehreren Darstellungen. Es kann als XML Metadata Interchange (XMI)-Dokument, als XML Schema-Dokument oder auch als annotierte Java-Klasse definiert werden. Jede dieser Darstellungen ist in die jeweils anderen überführbar. Die umgänglichste Art ist es, ein Modell im grafischen Editor zu erstellen.

Ein Kernfeature von EMF ist das automatische Erstellen von Code auf Grundlage eines Modells. Der generierte Code kann manuell erweitert werden und diese Erweiterungen fliessen bei erneuter Generierung, nachdem z.B. Erweiterungen am Modell vorgenommen wurden, in den Generierungsprozess mit ein. Dies ist ein entscheidendes Feature, da so das Modell zentraler Bestandteil des Entwicklungsprozesses ist und mit der Implementierung konsistent gehalten wird.

3.3. KIELER Infrastructure for Meta Layout

Das Ziel des KIELER Forschungsprojekts ist die Verbesserung der Pragmatik grafischer Modellierung. In diesem Zusammenhang ist das automatische Layout von Diagrammen von zentraler Bedeutung.

[2] `http://help.eclipse.org/indigo/index.jsp?topic=/org.eclipse.emf.doc/references/overview/EMF.html`

Auf der einen Seite verwendet KIELER verschiedene Familien von Algorithmen, welche ihrerseits jeweils eine eigene Notation für Modelle definieren (KIELER, Graphviz, OGDF, ...). Auf der anderen Seite verwendet KIELER unterschiedliche Frameworks zur Bearbeitung der Diagramme (derzeit GMF[3], Graphiti[4] und Piccolo2D[5]). Die Diagramme liegen im Allgemeinen in einer Vielzahl domänenspezifischer Notationen vor. Das Meta Layout bildet eine Brücke zwischen dem Layout und der Kombination aus Framework und Notation auf Grundlage einer abstrakten, strukturellen Repräsentation. Zu diesem Zweck nutzt es die Eigenschaft vieler Modellierungssprachen, dass sich die Struktur ihrer Modelle als Graph darstellen lässt, und definiert mit dem KGraph (siehe Abbildung 3.5) ein auf Ecore basierendes Metamodell für die strukturelle Repräsentation von Diagrammen.

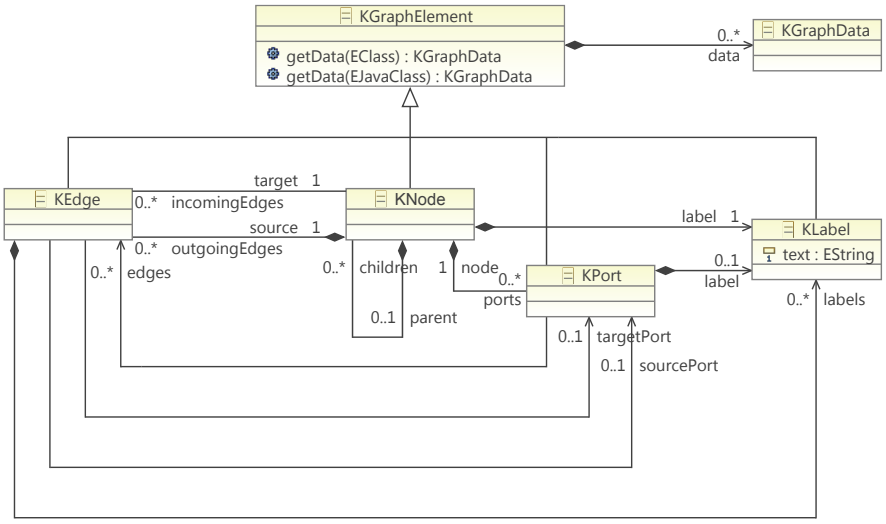


Abbildung 3.5.: Das KGraph-Metamodell

3.3.1. Das KGraph-Metamodell

Wie Abbildung 3.5 zeigt, bildet das KGraph-Metamodell die Struktur von Graphen ab. Es können Knoten definiert und direkt durch Kanten miteinander verbunden werden. Alternativ erlaubt es die Definition von Ports, welche mit genau einem Knoten assoziiert sind. Die Ports modellieren das Konzept genau definierter Verbindungen zwischen den Eingängen und Ausgängen der Komponenten eines Datenflussmodells. Jedes der zuvor genannten Elemente eines KGraph-Modells kann durch ein oder mehrere Label mit textuellen Informationen versehen werden.

[3] http://www.eclipse.org/modeling/gmp
[4] http://www.eclipse.org/graphiti
[5] http://www.piccolo2d.org

Der KGraph ist ein Strukturmodell und dessen Instanzen tragen daher keine layoutbezogenen Informationen. Die Assoziation zu KGraphData-Elementen bietet einen Mechanismus für das Anreichern eines Modells mit beliebigen Informationen. Dies wird für die layoutbezogenen Daten genutzt, das Meta Layout verwendet dafür das Ecore-Metamodell KLayoutData (siehe Abbildung 3.6).

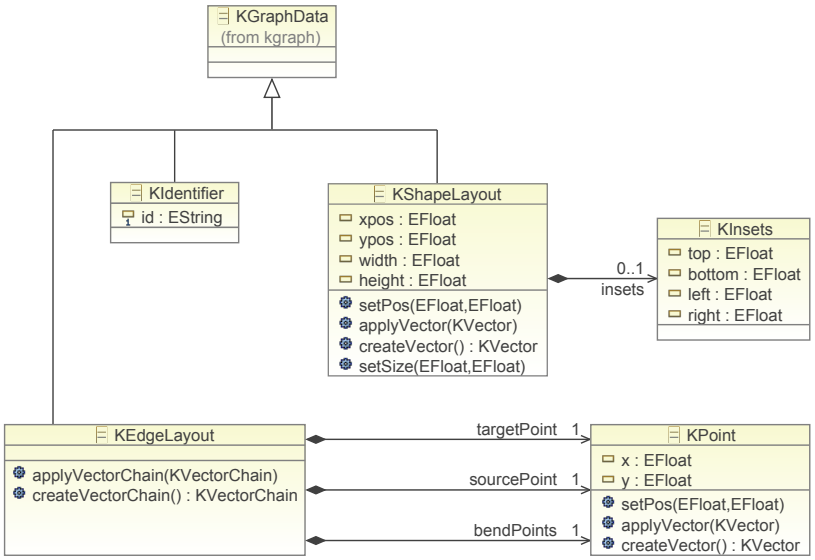


Abbildung 3.6.: Das KLayoutData-Metamodell

Jedes KGraph-Modellelement kann, abhängig von seinem Typ, mit Daten bezüglich dessen visueller Darstellung angereichert werden. Das Meta Layout verwendet dazu die Elemente KShapeLayout und KEdgeLayout. Das Element KIdentifier wird im Kontext der Integration des dienstbasierten Layouts in KIELER für die Identifikation von KGraph-Modellelementen verwendet, um layoutbezogene Informationen des von KWebS berechneten Modells in das Ursprungsmodell zu übernehmen.

Layoutbezogene Metadaten

Die Singleton-Klasse LayoutDataService hält zur Laufzeit von KIELER die layoutbezogenen Metadaten zentralisiert vor, wie z.B. die verfügbaren Layout-Algorithmen und Layout-Optionen. Layoutbezogene Plug-Ins stellen ihr diese Daten über Extensions der vom Meta Layout dafür bereitgestellten Extension-Points layoutProviders und layoutInfo zur Verfügung. Im Kontext dieser Arbeit ist der Extension-Point layoutProviders von besonderem Interesse, da KWebS seinen Nutzern mitteilen können muss, über welche Fähigkeiten sein dienstbasiertes Layout verfügt. An dieser Stelle ist anzumerken, dass diese Beschreibung den Stand von KIML zu Beginn der Arbeit widerspiegelt. Die im Rahmen dieser Arbeit vorgenommenen Erweiterungen stellt Abschnitt 5.2.2 vor.

3.3.2. Layout in KIELER

Wenn für ein Diagramm ein Layout benötigt wird, verwendet der zugehörige Editor eine Instanz der Klasse DiagramLayoutEngine. Sie ist der zentrale Zugang zum automatischen Layout innerhalb von KIELER und bildet die notwendige Brücke zwischen dem vom Editor genutzten Framework und dem automatischen Layout. Die Berechnung des Layouts findet auf Grundlage eines dem Diagramm strukturell entsprechenden KGraph-Modells statt. KIELER bietet Editoren für verschiedene Modellierungssprachen, für deren Implementierung es verschiedene Frameworks einsetzt. Das Ableiten des KGraph-Modells und auch das Anwenden des berechneten Layouts auf ein Diagramm muss daher auf eine Art geschehen, die die interne Repräsentation des Diagramms in einem spezifischen Framework berücksichtigt. Zu diesem Zweck verwendet KIELER *Layout Manager*. Gegenwärtig setzt KIELER drei Frameworks für die Implementierung seiner Editoren ein, dementsprechend gibt es drei Layout Manager:

- den GmfDiagramLayoutManager für auf GMF basierende Editoren,
- den GraphitiDiagramLayoutManager für Graphiti und
- den PiccoloDiagramLayoutManager für Piccolo2D.

Jeder Manager erbt von der abstrakten Basisklasse DiagramLayoutManager. Abbildung 3.7 gibt einen Überblick über den Layout-Prozess in KIELER. Wenn ein Editor das Layout des in ihm dargestellten Diagramms anfragt, stellt die DiagramLayoutEngine den zum gegebenen Editor kompatiblen Layout Manager fest. Mit ihm leitet sie das Strukturmodell des Diagramms ab. Daraufhin verwendet sie die RecursiveGraphLayoutEngine für die konkrete Berechnung dessen Layouts. Diese berücksichtigt mögliche Hierarchien des Strukturmodells und verwendet einen oder mehrere Algorithmen für die Berechnung. Nachdem die Berechnung des Layouts abgeschlossen ist, verwendet die DiagramLayoutEngine erneut den zuvor ermittelten Layout Manager für die Anwendung des Layouts auf das Diagramm und die Visualisierung dessen veränderter Sicht durch Animation.

3.4. Web Services

Web Services haben sich aus dem Bedarf nach einer auf offenen Standards basierenden, plattformunabhängigen Infrastruktur zur Inter-Prozess-Kommunikation in heterogenen Umgebungen entwickelt. Die klassischen *SOAP Web Services* basieren im Kern auf XML-Derivaten und sind mächtig in Bezug auf Dienstgüte. Aufgrund ihrer XML-Natur sind sie verhältnismäßig komplex in der Anwendung und vor allem „geschwätzig“ auf der Transportschicht. Aus diesem Grund hat sich eine alternative, leichtgewichtige Technologie entwickelt, die auf der REST-Architektur von Roy Thomas Fielding basiert [21]. Dieser Architektur folgende Web Services werden als *RESTful* bezeichnet.

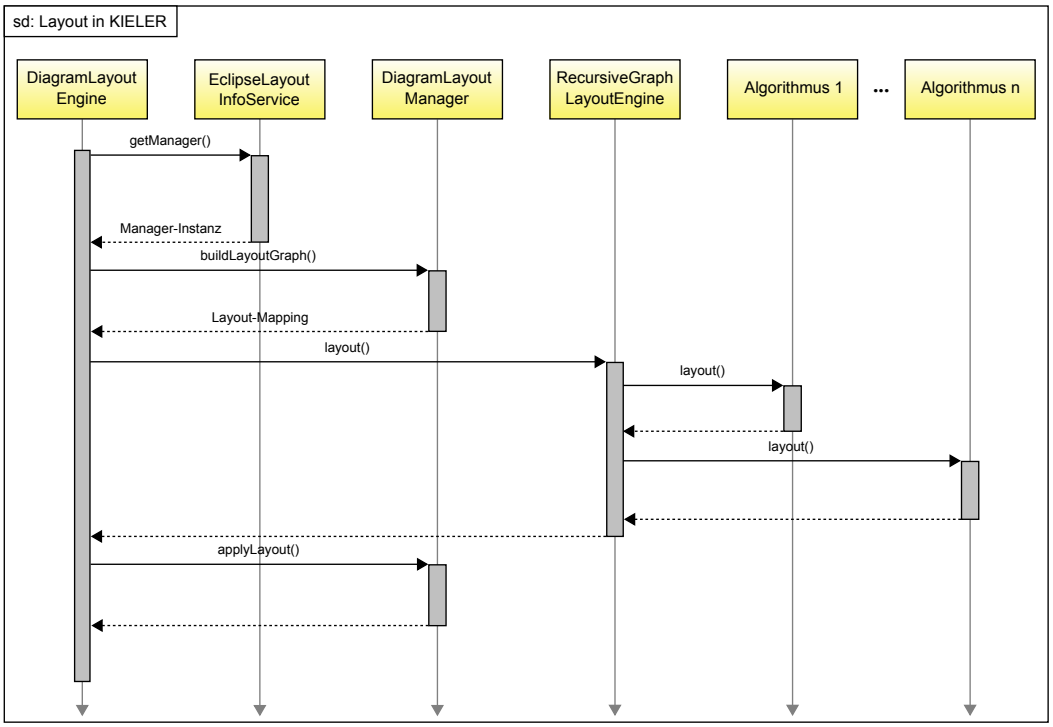


Abbildung 3.7.: Verlauf eines lokalen Layouts

Im Folgenden betrachten wir zunächst die SOAP Web Services. Wie wir im Verlauf dieser Arbeit sehen werden, verwendet KWebS zur Veröffentlichung seines dienstbasierten Layouts einen SOAP Web Service. Im Anschluss gehen wir auf die RESTful Web Services ein, um später beide Architekturen miteinander vergleichen und die Entscheidung für einen SOAP Web Service begründen zu können.

3.4.1. SOAP Web Services

In ihrem Konzept folgen die SOAP Web Services der serviceorientierten Architektur (SOA). Ihre Namensgebung ist in gewisser Hinsicht unscharf: SOAP ist ein unabhängiges Protokoll, es findet auch in nicht Web Service-bezogenen Umgebungen Anwendung. Des Weiteren ist die Verwendung von SOAP technisch gesehen auch nicht zwingend, da die verwendete Schnittstellenbeschreibungssprache (Interface Definition Language, IDL) offen ist für andere Standards. In der Praxis wird jedoch meist SOAP eingesetzt. Daher hat sich dieser Begriff etabliert.

Die SOA ist ein gedankliches Konzept, dessen grundlegende Idee es ist, komplexe Vorgänge durch Modularisierung zu entkoppeln. Dazu werden grundlegende, funktional in sich geschlossene Abläufe eines Vorgangs erarbeitet und in Diensten gekapselt. Es ist wichtig die Funktionalität der Dienste von den spezifischen Details eines konkreten Anwendungsfalls zu abstrahieren. Auf diese Weise erhöht sich ihre

Wiederverwendbarkeit und damit ihr Nutzwert. Dies ermöglicht des Weiteren ihre Kombinierbarkeit zu einer höherwertigen Funktionalität: sie sind *orchestrierbar*.

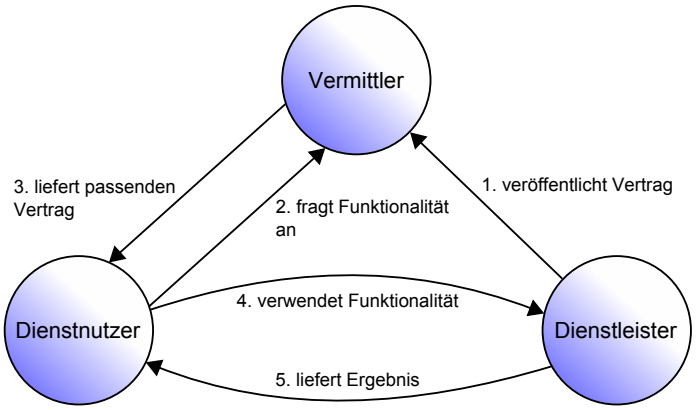


Abbildung 3.8.: Serviceorientierte Architektur

Abbildung 3.8 zeigt die Struktur einer SOA. Sie besteht aus den *Dienstleistern*, den *Dienstnutzern* und den *Vermittlern*. Ein Dienstleister bietet eine definierte Funktionalität in Form von Diensten innerhalb eines Netzwerkes öffentlich an. Er legt die gebotene Funktionalität und deren Nutzungsbedingungen in einem *Vertrag* fest. Der Vertrag bietet eine Abstraktion von den Details der Plattform des Dienstleisters und beschreibt dessen Funktionalität in einer maschinenlesbaren Form. Er ermöglicht dadurch die automatisierte Nutzung der von ihn beschriebenen Funktionalität in heterogenen Umgebungen. Der Dienstleister registriert den Vertrag bei einem Vermittler. Die Vermittler bilden die Brücke zwischen Dienstnutzer und Dienstleister und ermöglichen deren lose Kopplung. Ein Dienstnutzer fragt benötigte Funktionalität bei einem Vermittler an. Ist ein passender Dienst registriert, so erhält der Dienstnutzer den entsprechenden Vertrag, anhand dessen er den zugehörigen Dienst im Netzwerk auffinden und in Anspruch nehmen kann.

Abbildung 3.9 zeigt die Infrastruktur einer auf SOAP Web Services aufgebauten SOA. Die Formulierung eines Vertrages erfolgt in der Schnittstellenbeschreibungssprache WSDL. Die Interaktion mit einem Web Service erfolgt anhand von *Operationen*. Das Konzept der WSDL ist es, die von einem Web Service bereitgestellten Operationen und die zur Nutzung der Operationen notwendigen Datentypen abstrakt zu definieren. Details zur konkreten Nutzung, also das für den Transport verwendete Protokoll und Nachrichtenformat, werden darauf aufbauend durch *Bindungen* festgelegt. Die abstrakt definierten Schnittstellen eines Web Service können dadurch erneut verwendet und mit unterschiedlichen Bindungen angeboten werden. Die am häufigsten anzutreffende Bindung ist SOAP über HTTP. Der Grund dafür ist zum einen, dass SOAP für die Verwendung dieser Bindung einen Standard definiert und zum anderen, dass der von HTTP verwendete Port 80 von Firewalls weitestgehend nicht blockiert wird.

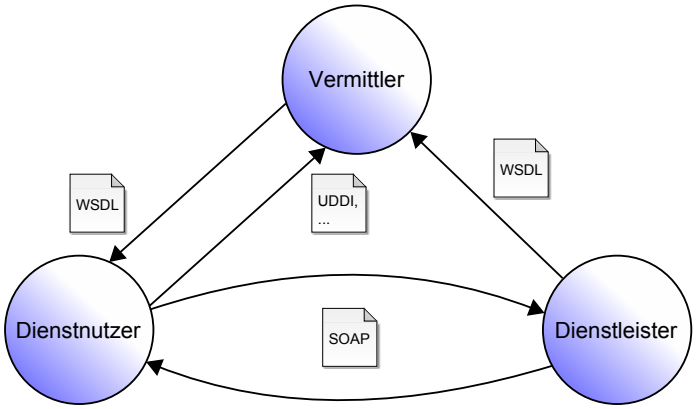


Abbildung 3.9.: SOA mit Web Services

Gemäß dem Konzept der SOA wird für das Veröffentlichen und Auffinden eines SOAP Web Service ein Repository und ein Brokering-Protokoll benötigt. Die Literatur nennt in Verbindung mit Web Services meist Universal Description, Discovery and Integration (UDDI). Dies gilt jedoch aufgrund technischer Mängel zumindest im allgemeinen Umfeld der SOA inzwischen als überholt und ein Nachfolger steht derzeit noch nicht fest[6]. Mit dem SOA Repository Artifact Model & Protocol (S-RAMP) befindet sich derzeit ein Standard für SOA-Repositories bei der Organization for the Advancement of Structured Information Standards (OASIS) in der Entwicklung, dessen Veröffentlichung für Ende 2011 geplant ist[7]. Er steht somit erst nach Abschluss dieser Arbeit zur Verfügung. Es ist des Weiteren nicht klar, ob sich S-RAMP durchsetzen wird. Im Kontext dieser Arbeit wird daher auf den Aspekt des Vermittelns nicht weiter eingegangen.

Web Service Description Language

SOAP Web Services verwenden WSDL [7] für die Formulierung des Vertrages. WSDL basiert auf XML, da dies ein allgemein akzeptierter Standard für die Notation strukturierter Daten ist und auf allen gängigen Plattformen Parser für die Verarbeitung von XML existieren. Dadurch erreicht WSDL die Abstraktion von Plattformen, und die maschinelle Verarbeitbarkeit ist ebenso sichergestellt.

Gegenwärtig existieren zwei Versionen der WSDL. Der Aufbau eines WSDL-Dokuments hängt von der verwendeten Version ab. Abbildung 3.10 zeigt die Struktur eines WSDL-Dokuments in Abhängigkeit der verwendeten Version. Von struktureller Seite her sind einige der Elemente umbenannt worden und die Signaturen der Operationen werden nicht mehr durch Nachrichten, sondern direkt durch die Datentypen definiert. WSDL in der Version 2.0 unterstützt Vererbung von Schnittstellen und bietet besseren Support für RESTful Web Services (siehe 3.4.2) durch die Un-

[6] IBM, Microsoft und SAP haben 2007 ihre öffentlichen UDDI-Registrierungen abgeschaltet.
[7] `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=s-ramp`

terstützung der vier HTTP-Verben *GET*, *POST*, *PUT* und *DELETE*. Die folgende Beschreibung beschränkt sich auf Version 1.1. Sie ist lange Zeit Standard gewesen (nicht beim W3C, aber in der Praxis) und wird auch im Rahmen dieser Arbeit verwendet, um Abwärtskompatibilität zu gewährleisten.

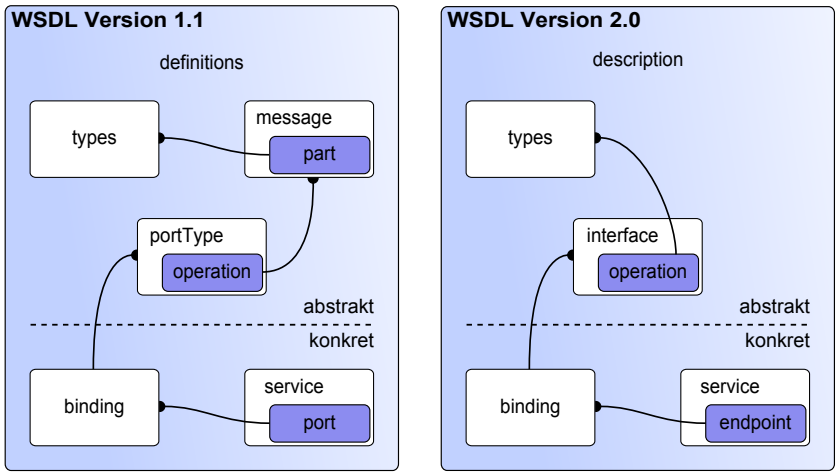


Abbildung 3.10.: Struktur der WSDL

Ein WSDL-Dokument besteht aus einem *abstrakten* und einem *konkreten* Teil. Der abstrakte Teil dient der plattformunabhängigen Definition der Schnittstellen eines Web Service. Ein Dokument beginnt mit dem Element definitions als Wurzel, innerhalb dessen die folgenden Elemente zur abstrakten Beschreibung der Schnittstellen verwendet werden:

- types: Ein Web Service deklariert im Allgemeinen eine Menge von strukturierten Datentypen, die die von ihm angebotenen Operationen in Form von Parametern oder Rückgabewerten verwenden. Dies können z.B. bei einem Java-basierten Web Service eigens implementierte Java-Klassen sein. Diese Datentypen müssen einem Nutzer bekannt sein, um den Web Service in Anspruch nehmen zu können. Das types-Element bietet die Möglichkeit, diese Datentypen plattformunabhängig zu definieren. Die zu verwendende Notation ist durch den Standard von WSDL nicht zwingend festgelegt, er empfiehlt XML Schema.
- message: Eine Schnittstelle definiert eine Menge von Operationen, die von einem Nutzer in Anspruch genommen werden können. Eine Operation verfügt im Allgemeinen über Parameter und einen Rückgabewert. Sie werden auf der Transportschicht durch Nachrichten modelliert. Die WSDL folgt diesem Konzept, indem sie die Signaturen von Operationen ebenfalls auf Nachrichten abbildet. Das message-Element dient der Definition einer Nachricht, welche als Eingabe oder Ausgabe einer Operation, oder zur Benachrichtigung des Nutzers über eine aufgetretene Fehlersituation verwendet werden kann.

Ein message-Element gruppiert eine Menge von part-Elementen. Während das message-Element die Signatur einer Operation modelliert, dient das part-Element der Modellierung ihrer einzelnen Parameter. Jedes part-Element verfügt über einen innerhalb der Nachricht eindeutigen Bezeichner. Seine Typisierung erfolgt durch Referenzierung eines innerhalb des types-Elements deklarierten Datentypen.

- operation: Die Operationen eines Web Service werden mithilfe des operation-Elements definiert. Je nach Art der Operation enthält es ein input- und ein output-Element, welche durch Referenzierung der definierten Nachrichten die Signatur der Operation bilden. Ein drittes mögliches Element ist fault zum Signalisieren einer Fehlersituation.

- portType: Das portType-Element gruppiert eine Menge von Operationen und definiert damit eine abstrakte Schnittstelle eines Web Service. Ein WSDL-Dokument kann mehrere portType-Elemente und damit mehrere abstrakte Schnittstellen enthalten.

Der konkrete Teil eines WSDL-Dokuments nimmt die Bindung der abstrakt definierten Schnittstellen an konkrete Nachrichtenformate und Transportprotokolle vor:

- binding: Das binding-Element bindet eine abstrakte Schnittstelle eines Web Service an ein Transportprotokoll und ein Nachrichtenformat. Eine abstrakte Schnittstelle kann über mehrere Bindungen verfügen.

- port: Während das binding-Element die Art der Bindung einer Schnittstelle festlegt, definiert das port-Element die konkrete Adresse, über die Nutzer ihre Operationen in Anspruch nehmen können.

- service: Ein Web Service kann zur gleichen Zeit mehrere Schnittstellen veröffentlichen. Das service-Element fasst ein oder mehrere port-Elemente zu einem Web Service zusammen.

Listing 3.1 zeigt den Vertrag eines virtuellen Uhrzeit-Web Service. Die von ihm veröffentlichte Operation getTime liefert die gegenwärtige Uhrzeit in Nanosekunden.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="TimeService"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://www.timeservice.com/2011/09/TimeService"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.timeservice.com/2011/09/TimeService">
  <types>
    <xs:schema version="1.0"
      xmlns:tns="http://www.timeservice.com/2011/09/TimeService"
      xmlns:xs="http://www.w3.org/2001/XMLSchema"
```

```
      targetNamespace="http://www.timeservice.com/2011/09/TimeService">
      <xs:element name="getTime" type="tns:getTime"/>
      <xs:element name="getTimeResponse" type="tns:getTimeResponse"/>
      <xs:complexType name="getTime">
        <xs:sequence/>
      </xs:complexType>
      <xs:complexType name="getTimeResponse">
        <xs:sequence>
          <xs:element name="return" type="xs:double"/>
        </xs:sequence>
      </xs:complexType>
    </xs:schema>
  </types>
  <message name="getTime">
    <part name="parameters" element="tns:getTime"/>
  </message>
  <message name="getTimeResponse">
    <part name="parameters" element="tns:getTimeResponse"/>
  </message>
  <portType name="TimeServicePort">
    <operation name="getTime">
      <input message="tns:getTime"/>
      <output message="tns:getTimeResponse"/>
    </operation>
  </portType>
  <binding name="TimeServicePortBinding" type="tns:TimeServicePort">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
                  style="document"/>
    <operation name="getTime">
      <soap:operation soapAction=""/>
      <input><soap:body use="literal"/></input>
      <output><soap:body use="literal"/></output>
    </operation>
  </binding>
  <service name="TimeService">
    <port name="TimeServicePort" binding="tns:TimeServicePortBinding">
      <soap:address location="http://www.timeservice.com/timeservice"/>
    </port>
  </service>
</definitions>
```

Listing 3.1: Vertrag eines virtuellen Uhrzeit-Dienstes

Das gezeigte Listing soll an dieser Stelle nicht im Detail beschrieben werden, es soll lediglich einen Eindruck der Komplexität von auf XML basierenden Web Services vermitteln. Es benötigt 51 Zeilen und 27 Elemente für die Definition eines Web Service, der lediglich eine Operation veröffentlicht, die keine Parameter benötigt und eine Gleitkommazahl als Ergebnis liefert. Die Anzahl an verwendeten Attributen und Namensräumen trägt ihren eigenen Teil zur Komplexität bei.

SOAP

Das Anliegen von SOAP [26] ist es, einen Standard für die Nachrichtenübermittlung in verteilten, heterogenen Systemen zu definieren. Ein zentraler Aspekt bei der Entwicklung von SOAP war die Erweiterbarkeit. Daher legt sich SOAP nicht auf ein bestimmtes Transportprotokoll fest und enthält auch keine Mechanismen für die Sicherstellung von Dienstgüte-Kriterien. Im Kern beschreibt der Standard die Übermittlung einer SOAP-Nachricht von einem *initialen Sender* über *intermediäre Empfänger* an einen *ultimativen Empfänger* und definiert, wie jeder der beteiligten *SOAP-Knoten* in Abhängigkeit seiner *Rolle* mit einer Nachricht umzugehen hat.

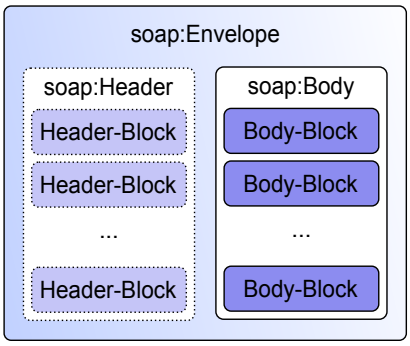


Abbildung 3.11.: Aufbau einer SOAP-Nachricht (nach [26])

Das Nachrichtenformat von SOAP basiert auf XML und ist sehr einfach gehalten. Neben dem Wurzelelement Envelope besteht eine SOAP-Nachricht aus einem optionalen Header-Element und einem erforderlichen Body-Element, welche jeweils beliebig geformtes XML enthalten können (siehe Abbildung 3.11). Die jeweils direkten Kinder des Header- und des Body-Elementes bilden die *Header-Blöcke* und die *Body-Blöcke*, deren Semantik anwendungsspezifisch ist.

Ein Sender legt seine Nutzlast im Body ab und für die Verarbeitung der Nachricht eventuell benötigte Metadaten im Header. Die Nutzlast ist ausschließlich an den ultimativen Empfänger gerichtet, während die Metadaten von jedem am Transport der Nachricht beteiligten SOAP-Knoten verarbeitet werden müssen, sofern sie an die Rolle gerichtet sind, die ein SOAP-Knoten auf dem Transportweg einnimmt. Sollte bei der Verarbeitung einer Nachricht ein Fehler auftreten, kann ein SOAP-Knoten dies durch ein Fault-Element im Body der Antwort signalisieren.

SOAP definiert eine Reihe von Konventionen, wie Remote Procedure Calls (RPCs) in einer Nachricht abzubilden sind. Betrachten wir den virtuellen Uhrzeit-Dienst aus Listing 3.1, dann hat eine SOAP-Anfrage nach der aktuellen Uhrzeit die in Listing 3.2 gezeigte Form.

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ts:getTime xmlns:ts="http://www.timeservice.com/2011/09/TimeService"/>
  </soap:Body>
</soap:Envelope>
```

Listing 3.2: Eine SOAP-Anfrage an den Uhrzeit-Dienst

Der Aufruf der Operation getTime wird auf ein Element gleichen Namens im Body der SOAP-Nachricht abgebildet. Da diese Operation keine Parameter erwartet, ist das getTime-Element ein leeres Element. Allgemein werden Parameter einer Operation als Kind-Elemente unterhalb des die Operation definierenden Elementes abgebildet. Listing 3.3 zeigt eine mögliche SOAP-Antwort des virtuellen Uhrzeit-Dienstes.

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ts:getTimeResponse xmlns:ts="http://www.timeservice.com/2011/09/TimeService">
      <return>
        7.5260286666505E13
      </return>
    </ts:getTimeResponse>
  </soap:Body>
</soap:Envelope>
```

Listing 3.3: Eine SOAP-Antwort des Uhrzeit-Dienstes

Das Ergebnis der Operation ist in ein return-Element eingebettet. Dieses ist wiederum Kind des getTimeResponse-Elementes. Der Name dieses Elementes entspricht der Konvention, dass in der Antwort-Nachricht für das Wurzelelement der Name der aufgerufenen Operation mit angehängtem „Response“ verwendet wird.

Ein interessanter Aspekt von SOAP ist die Anreicherung mit Dienstgüte-Kriterien. Dafür bestehen zwei Möglichkeiten. Zunächst können Eigenschaften des Übertragungsweges ausgenutzt werden. Die Übertragung einer SOAP-Nachricht über HTTP sorgt z.B. dafür, dass die Nachricht auch ankommt (solange der Empfänger tatsächlich existiert und auch erreichbar ist). HTTP erweitert SOAP damit um Zuverlässigkeit. Durch die Natur von HTTP als *Request/Response*-Protokoll wird des Weiteren eine Kopplung von Anfrage-Nachricht und Antwort-Nachricht erreicht. Durch HTTPS kann zusätzlich Vertraulichkeit gewährleistet werden. Die zweite Möglichkeit liegt in der Struktur einer SOAP-Nachricht. Der Header einer SOAP-Nachricht ist konzeptionell für die Anreicherung einer Nachricht mit beliebigen, für die Vearbeitung erforderlichen Metadaten vorgesehen. Dieser Mechanismus kann von Standards für die Definition dienstgütebezogener Header-Blöcke und entsprechender Verarbeitungsregeln für SOAP-Knoten genutzt werden.

3.4.2. RESTful Web Services

SOAP Web Services bieten nicht zuletzt durch die Verwendung von SOAP ein mächtiges Werkzeug für plattformunabhängige Inter-Prozess-Kommunikation. Ihre XML-Natur bedeutet auf der Transportebene in vielen Fällen einen spürbaren Zusatzaufwand zu den eigentlichen Nutzdaten, und auch die Verarbeitung von XML geht mit erhöhter Rechenleistung einher. Viele Anwendungsfälle erfordern die technischen Möglichkeiten eines SOAP Web Service nicht und deren Komplexität ist eher hinderlich.

Eine weitere Kritik an SOAP Web Services ist, dass sie HTTP ausschließlich als Transportprotokoll verwenden, und das weitestgehend, um transparent zu sein in Bezug auf Firewalls. Die Stärken von HTTP als Anwendungsprotokoll werden dabei außer Acht gelassen. Die HTTP-Verben *GET*, *POST*, *PUT* und *DELETE* bilden, im Gegensatz zu den proprietären Schnittstellen der SOAP Web Services, eine standardisierte Schnittstelle und Semantik.

Aus diesen Gründen entwickelte sich eine alternative Technologie. Sie basiert auf der Doktorarbeit „Architectural Styles and the Design of Network-based Software Architectures“ von Roy Thomas Fielding [21]. Sie untersucht, warum ein so großes und heterogenes Netzwerk wie das WWW sich so erfolgreich etablieren konnte, und leitet daraus einen Architekturstil für netzwerkbasierte Systeme ab.

Die REST-Architektur fördert ein Schichtenmodell auf Netzwerkebene, in dem verschiedene *Komponenten* wie z.B. Proxies, Gateways oder Caches, und natürlich auch Clients und Server miteinander interagieren. Das der Architektur zugrundeliegende atomare Element der Interaktion ist die *Ressource*. Eine Ressoure stellt ein Konzept eines realen Objektes dar (z.B. eine Person, eine Datenbank oder das aktuelle Wetter). Sie wird durch verschiedene *Repräsentationen* dargestellt, wodurch eine Abstraktion vom zugrundeliegenden Konzept erreicht wird. So kann z.B. das Ergebnis einer Datenbankanfrage in Comma Separated Values (CSV)-Notation oder XML dargestellt sein. REST definiert die folgenden Konzepte, um ein gut skalierendes Gesamtsystem zu erhalten:

- *Eindeutige Adressierbarkeit*: Jede Ressource ist über ein definiertes Schema eindeutig identifizierbar.
- *Austausch von Repräsentationen*: Der Austausch zwischen Client und Server basiert auf Repräsentationen. Jede Ressource kann auf verschiedene Weise dargestellt werden. Ein konkretes Format, der *Media Type*, wird zwischen Client und Server ausgehandelt. Dies gilt nicht nur für das Lesen einer Ressource. Auch das Aktualisieren oder Anlegen einer Ressource erfolgt anhand von Repräsentationen. Die Media Types sind eine eingeschränkte Menge von standardisierten Darstellungsformen. Ein konkreter Media Type ist der Repräsentation einer Ressource in Form von Metadaten beigefügt, sodass eine Komponente des Netzwerkes die Repräsentation korrekt interpretieren kann.

  Die Verwendung von Repräsentationen für die Interaktion mit einer Ressource dient dem Schutz des Servers, denn dieser muss seine internen Strukturen nicht

offenlegen (z.B. Datenbank-Schema). Des Weiteren kann die Darstellung einer Ressource auf die Bedürfnisse des Clients angepasst werden. Ein Browser benötigt z.B. eine Repräsentation in HTML. Für die automatisierte Auswertung einer Ressource ist HTML wenig geeignet, denn es enthält neben der eigentlichen Information vor allem Anweisungen bezüglich deren visueller Darstellung für einen Menschen. Um eine automatisierte Auswertung zu ermöglichen, kann eine Ressource neben HTML andere Repräsentationen anbieten wie z.B. XML, JSON, oder auch CSV.

- *Einheitliche Schnittstelle*: Die Interaktion mit den Ressourcen erfolgt anhand einer festgelegten Menge von Operationen. Die Semantik dieser Operationen ist klar definiert. Dadurch wird eine einheitliche Schnittstelle geschaffen, aufgrund derer jede Komponente im Netzwerk mit jeder Ressource interagieren kann.

  Eine einheitliche Schnittstelle verbessert die Skalierung des Netzwerkes. Eine Anfrage kann auf ihrem Weg durch das Netzwerk von intermediären Komponenten verarbeitet werden. Aufgrund der einheitlichen Schnittstelle, die auch sie unterstützen müssen, sind sie in der Lage, die Anfrage zu interpretieren. Abhängig von der Semantik der Operation können Anfragen z.B. zwischengespeichert werden. Dies verringert die Latenz einer erneuten, identischen Anfrage und die Menge von im Netzwerk ausgetauschten Daten. Ebenso findet eine Entlastung des Servers statt, an den die Anfrage ursprünglich gerichtet war.

- *Zustandslose Kommunikation*: Die Kommunikation zwischen Client und Server ist zustandslos. Alle für die Bearbeitung einer Anfrage notwendigen Informationen sind in der Anfrage enthalten. Dadurch wird eine bessere Skalierung eines Servers erreicht, denn er ist nicht darauf angewiesen, einen Kontext für die Kommunikation mit einem Client vorzuhalten. Der für die Anfrage notwendige Kontext wird auf Seiten des Clients vorgehalten und entsprechend in die Anfrage eingebettet.

- *HATEOAS*: Das Prinzip hinter Hypermedia as the Engine of Application State (HATEOAS) ist, dass Repräsentationen Verweise auf andere Ressourcen enthalten können. Diese Verweise folgen dem Adressierungsschema der Architektur. Durch die Einbettung von Verweisen in eine Repräsentation entsteht die Möglichkeit der Navigation. Da die Kommunikation zwischen Client und Server nicht zustandsbehaftet ist, modelliert eine Repräsentation (das „hypermedia“ in HATEOAS) zwei Aspekte:

  – den gegenwärtigen Zustand der Anwendung und
  – die Änderung des Anwendungszustands durch Folgen eines Verweises.

  Ein Beispiel sind die Links in Web-Seiten.

REST ist ein Architekturstil und kein konkretes Protokoll. REST trifft daher keine Aussage darüber, welche Operationen in der Schnittstelle enthalten sind oder wie das

Adressierungsschema definiert ist. RESTful Web Services basieren typischerweise auf HTTP. Tatsächlich beschreibt REST die hinter HTTP stehende Architektur, an deren Entwicklung Roy Thomas Fielding maßgeblich beteiligt war. Ressourcen werden über Uniform Resource Identifier (URI) adressiert und die Schnittstelle besteht aus den oben genannten HTTP-Verben *GET*, *POST*, *PUT* und *DELETE*. Die Semantik der Verben ist dabei wie folgt:

- *GET* dient zum Lesen einer Ressource. Der Körper der Antwort enthält die Repräsentation.
- *POST* dient zum Erzeugen einer neuen Ressource. Der Körper der Anfrage enthält deren Repräsentation. Der Server legt die URI fest, unter der die Ressource zukünftig adressierbar sein wird, und übermittelt sie im *Location*-Header der Antwort an den Client.
- *PUT* erzeugt oder aktualisiert eine Ressource. Der Körper der Anfrage enthält die aktualisierte Repräsentation. Die Adresse der Ressource wird vom Client durch die verwendete URI festgelegt.
- *DELETE* löscht eine Ressource.

Sowohl *GET*, *PUT* und *DELETE* sind *idempotent*: sie können mehrmals nacheinander auf derselben Ressource aufgerufen werden und das Ergebnis ist dasselbe, wie bei einem einmaligen Aufruf (sofern bei *PUT* auch jeweils dieselbe Repräsentation verwendet wird). *GET* ist dazu *sicher*: das Lesen einer Ressource verändert den Zustand des Servers nicht. Das einzig nicht idempotente und auch unsichere Verb ist *POST*. Neben dem Erzeugen von Ressourcen wird es daher auch für die Modellierung von Operationen verwendet, die mit den übrigen Verben nicht abzubilden sind.

3.5. Java API for XML - Web Services

Wie wir im Verlauf dieser Arbeit sehen werden, wird KWebS als SOAP Web Service angeboten und implementiert. Aus diesem Grund geht dieser Abschnitt auf die Aspekte der Implementierung eines SOAP Web Service ein und stellt das verwendete Web Service-Framework JAX-WS vor.

3.5.1. Implementierung eines SOAP Web Service

Aufgrund der Komplexität der SOAP Web Services existieren verschiedene Frameworks wie z.B. Apache Axis2[8], die die Bereitstellung eines Web Service und dessen Nutzung vereinfachen. Neben der Abbildung der in einer SOAP-Nachricht enthaltenen Anfrage auf die Plattform ihres Empfängers (dem Marshalling und Unmarshalling) liegt der Fokus auf der Automation, d.h. der automatischen Code-Generierung.

[8] `http://axis.apache.org/axis2/java/core/index.html`

Im Allgemeinen unterstützt ein Framework zwei Herangehensweisen für die Implementierung eines Web Service:

- *Code-First*: Ein Web Service wird in der für dessen Implementierung gewählten Sprache entwickelt und mit Metadaten bezüglich seiner Veröffentlichung angereichert. In Java geschieht dies im Allgemeinen durch Annotation einer Klasse oder einer Schnittstelle. Die Annotationen weisen die Klasse als Web Service bzw. die Schnittstelle als dessen Service Endpoint Interface (SEI) aus und definieren, welche ihrer Methoden der Vertrag als Operation bereitstellt. Der zugehörige Vertrag entsteht durch Code-Generierung.

- *Contract-First*: Die Entwicklung eines Web Service beginnt mit der Formulierung des Vertrages. Die Implementierung erfolgt durch Realisierung der durch Code-Generierung erstellten Schnittstelle (des SEI).

Der Hauptvorteil des Code-First-Ansatzes ist dessen Einfachheit. Ein Entwickler definiert den von ihm gewünschten Web Service in der ihm bekannten Umgebung. Das Problem besteht darin, dass der generierte Vertrag in seiner Form von dem verwendeten Framework abhängig ist. Dies bewirkt, dass der Web Service letzten Endes nicht plattformneutral ist, da die vom Web Service angebotene Schnittstelle ihre Semantik bei einem Wechsel des Frameworks verändern kann. Ein weiteres Problem des Code-First-Ansatzes ist, dass Programmiersprachen im Allgemeinen eventuelle Anforderungen an die einer Operation übergebenen Parameter, wie z.B. gültiger Wertebereich einer Ganzzahl, nicht formulieren können. Die durch den generierten Vertrag beschriebene Schnittstelle ist somit unscharf in ihrer Semantik. Der Contract-First-Ansatz ist somit zu bevorzugen, da die Semantik der Schnittstelle genauer definiert werden kann und ein Wechsel des Frameworks auf die Schnittstelle keinen Einfluss hat. Auch die Definition der Parameter einer Operation anhand von XML Schema bietet umfangreichere Möglichkeiten. So können z.B. gültige Belegungen eines String-Parameters genau definiert werden, wie Listing 3.4 zeigt. Dies ist z.B. mit einer Java-Methodensignatur nicht möglich.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="RichtigOderFalsch">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:enumeration value="Richtig"/>
        <xs:enumeration value="Falsch"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:element>
</xs:schema>
```

Listing 3.4: Gültige Belegungen eines String-Parameters in XML Schema

Die mit dem Contract-First-Ansatz verbundenen Kosten liegen im Wesentlichen darin, dass insbesondere sehr komplexe Web Services durch die XML-Natur von WSDL aufwendig in der Entwicklung sind.

3.5.2. Implementierung mit JAX-WS

JAX-WS[9] ist eine Spezifikation für die Implementierung von auf XML basierenden Web Services in Java. Historisch gesehen ist sie der Nachfolger der Java API for XML-based RPC (JAX-RPC). Das Metro-Projekt[10] stellt die Referenzimplementierung von JAX-WS[11] bereit. Sie gilt als robust und produktionsreif und findet z.B. im GlassFish-Projekt[12] Verwendung, Oracles Referenzimplementierung eines Java EE-konformen Anwendungsservers. Auch die Standard-Edition von Java unterstützt seit Version 6 JAX-WS durch ihre Integration.

Im Folgenden betrachten wir zwei Aspekte von JAX-WS. Wir beginnen mit der Implementierung eines Web Service mit JAX-WS. Im Anschluss betrachten wir das *Binding* der im Vertrag eines Web Service definierten Datentypen, d.h. deren Abbildung in Java-Klassen. Des Weiteren betrachten wir das *Marshaling* und das *Unmarshaling* der zwischen einem Nutzer und einem Web Service zur Laufzeit ausgetauschten Nachrichten.

Ein Web Service mit JAX-WS

Wie auch andere Frameworks unterstützt die Referenzimplementierung sowohl den Code-First- als auch den Contract-First-Ansatz. Die Entwicklung eines Code-First-Web Service gestaltet sich dabei durch die Verwendung von Annotationen[13] sehr einfach. Listing 3.5 zeigt die Implementierung eines Web Service, dessen einzige Operation als Ergebnis den String Hello World! liefert.

```
package de.cau.cs.kieler.service;
import javax.jws.WebService;
import javax.jws.WebMethod;
/**
 * The endpoint implementation of the "HelloWorld" web service
 */
@WebService
public class HelloWorld {
    // Implementation of the "echo" operation
    @WebMethod
    public String echo() {
        return "Hello World!";
    }
```

[9] http://jcp.org/en/jsr/detail?id=224
[10] http://metro.java.net
[11] http://jax-ws.java.net
[12] http://glassfish.java.net
[13] http://jcp.org/en/jsr/detail?id=181

```
}
```

Listing 3.5: Ein Code-First JAX-WS Web Service

@WebService weist die Klasse HelloWorld als Web Service aus und @WebMethod kennzeichnet deren Methode echo als Operation. Diese Auszeichnung ist nicht zwingend notwendig, da öffentliche Methoden standardmäßig publiziert werden. Listing 3.6 zeigt die Veröffentlichung des Web Service durch die Klasse Endpoint.

```
package de.cau.cs.kieler.server;
import javax.xml.ws.Endpoint;
import de.cau.cs.kieler.service.HelloWorld;
/**
 * Server implementation for publishing the "HelloWorld" web service
 */
public class HelloWorldServer {
    // The address on which the endpoint shall publish the web service
    private static final String SERVICE_URL = "http://localhost:8080/helloWorld";
    /**
     * Entry point of the server application
     */
    public static void main(String args[]) {
        // The instance of the web service implementation
        HelloWorld helloWorld = new HelloWorld();
        // Create the endpoint, bind it to the given address and publish
        // the web service on it
        Endpoint endpoint = Endpoint.publish(SERVICE_URL, helloWorld);
    }
}
```

Listing 3.6: Veröffentlichung eines Web Service mit JAX-WS

Für die clientseitige, Java-basierte Nutzung des Web Service können mit dem Java-Tool wsimport die benötigten Klassen HelloWorld und HelloWorldService aus seinem Vertrag generiert werden:

wsimport -p de.cau.cs.kieler.client -keep http://localhost:8080/helloWorld?wsdl

Das Argument -p legt die erzeugten Klassen im angegebenen Java-Package de.cau.cs.kieler.client ab. Durch -keep wird erreicht, dass die generierten Klassen nach ihrer Übersetzung nicht gelöscht werden. Ein Java-Client nutzt den HelloWorld-Web Service wie in Listing 3.7 abgebildet.

```
package de.cau.cs.kieler.client;
public class HelloWorldClient {
    // The endpoint address the service is bound to
```

```
    private static final String SERVICE_URL = "http://localhost:8080/helloWorld";
    /**
     * Entry point of the client application
     */
    public static void main(String args[]) {
        // Connect to the web service
        HelloWorldService helloWorldService =
            new HelloWorldService(SERVICE_URL + "?wsdl");
        // Create a proxy to access the web service
        HelloWorld helloWorld = helloWorldService.getHelloWorldPort();
        // Call the "echo" operation of the web service and display the result
        System.out.println(helloWorld.echo());
    }
}
```

Listing 3.7: Nutzung eines Web Service mit JAX-WS

Das gezeigte Beispiel erzeugt die Ausgabe Hello World!. Die mit wsimport generierten Artefakte enthalten des Weiteren das SEI, welches für die Contract-First basierte Entwicklung des Web Service genutzt werden kann.

Binding und Marshaling

Ein Kernkonzept der Web Services ist die Abstraktion von den beteiligten Plattformen. Der Vertrag modelliert die Schnittstelle eines Web Service in neutraler Darstellung und auf Transportebene werden auf XML basierende Nachrichten ausgetauscht, meist in Form von SOAP-Nachrichten. Sie modellieren die Aufrufe von Operationen und die Übergabe von Argumenten und Rückgabewerten.

Ein Web Service-Framework stellt Mechanismen zur Verfügung, die zum einen die Bereitstellung und zum anderen die Nutzung eines Web Service auf einfache Art ermöglichen. Es verwendet dafür sprachliche Konstrukte der zugrundeliegenden Plattform. Eine wesentliche Aufgabe eines Frameworks ist daher die Übersetzung der plattformabhängigen Notationen in Nachrichten auf Transportebene und umgekehrt.

Ein Web Service kann Datentypen definieren, die zu seiner Nutzung erforderlich sind. Ein Aspekt der Übersetzung ist daher die Abbildung dieser Datentypen in Artefakte der verwendeten Plattform, z.B. in Java-Klassen. Dieser Vorgang findet meist statisch statt und wird als *Binding* bezeichnet. Das Binding nimmt des Weiteren die Generierung der Datentypen aus den beteiligten Artefakten vor, falls ein Web Service Code-First entwickelt ist, und der Vertrag zu dessen Laufzeit automatisch erzeugt wird.

Sowohl Nutzer als auch Dienst verwenden plattformspezifische Darstellungen der ausgetauschten Nachrichten. Ein Java-basierter Nutzer verwendet z.B. einen Methodenaufruf, um einen Web Service in Anspruch zu nehmen. Die Argumente sind Instanzen von Java-Klassen, welche zuvor durch das Binding erzeugt wurden, oder Basistypen der Java-Plattform entsprechen. Der zweite Aspekt der Übersetzung ist

daher die Abbildung der plattformabhängigen Darstellung von Nachrichten in Nachrichten auf Transportebene. Dieser Vorgang wird als *Marshaling* bezeichnet, das *Unmarshaling* bildet dementsprechend die Nachrichten auf die Plattform des Empfängers ab.

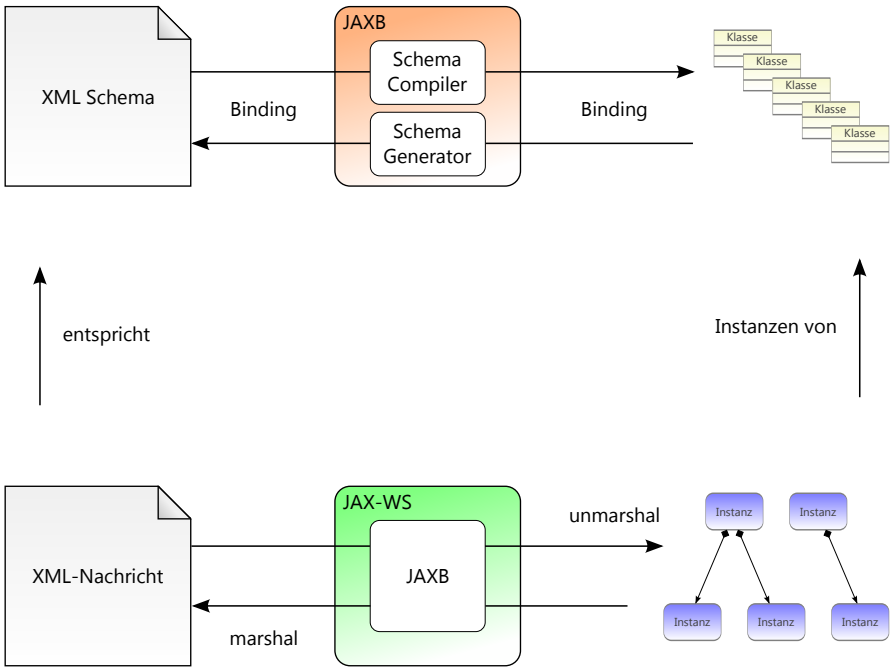


Abbildung 3.12.: Binding, Marshaling und Unmarshaling in JAX-WS mit JAXB (nach [25])

Für das Binding verwendet JAX-WS die Java Architecture for XML Binding (JAXB)[14]. Einen Überblick gibt Abbildung 3.12. Das Binding bildet die im Vertrag durch XML Schema definierten Datentypen in Java-Klassen ab. JAXB verwendet dazu den *Schema Compiler*. Dieser Vorgang findet während der Code-Generierung eines Contract-First entwickelten Web Service statt, und auch während der Generierung des zur Nutzung eines Web Service notwendigen Client-Codes. Wird ein Web Service Code-First entwickelt, so wird dessen Vertrag zur Laufzeit generiert. In diesem Fall verwendet JAXB den *Schema Generator*, um das notwendige XML Schema zur Definition der Datentypen aus den beteiligten Klassen zu generieren. Das Marshaling und Unmarshaling wird von JAX-WS ebenfalls an JAXB delegiert.

[14] `http://jcp.org/en/jsr/detail?id=222`

3.6. Java Electronic Tool Integration

jETI[15] ist eine an der Technischen Universität Dortmund entwickelte Plattform [13]. Als Java-basierte, modernisierte Neu-Implementierung der Konzepte der Electronic Tool Integration (ETI)-Plattform bietet sie die Möglichkeit, Werkzeuge, sogenannte *Tools*, über ein Netzwerk verteilt zur Verfügung zu stellen. jETI basiert auf einem *Toolserver* (siehe Abbildung 3.13), der selbst neben einer auf TCP basierenden Schnittstelle Web Services für die Bereitstellung der Tools nutzt. jETI ist speziell für auf Dateioperationen basierende Tools entworfen worden, um die einfache Integration von Legacy-Anwendungen in das jABC zu ermöglichen. Dazu wird ein Tool durch ein SIB gekapselt, welches aus der Beschreibung des Tools in XML automatisch generiert wird.

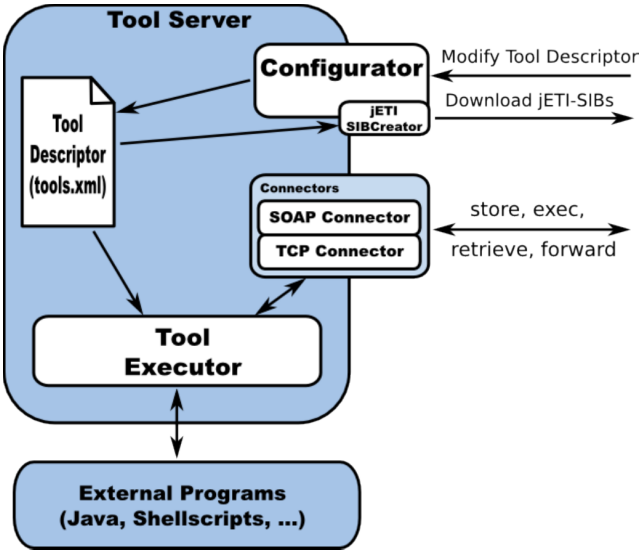


Abbildung 3.13.: Der jETI-Toolserver (Quelle: [16])

In jETI erfolgt der Zugriff auf ein Tool sitzungsorientiert und dateibasiert. Ein Nutzer startet eine Sitzung und lädt die zu verarbeitende Information in Form einer Datei auf den Server. Diese wird in einem virtuellen Dateisystem abgelegt und steht so dem Tool zur Verfügung. Das parametrisierbare Tool führt die gewünschte Operation aus und der Nutzer kann das Ergebnis, welches wiederum als Datei bereitgestellt wird, herunterladen. Mehrere Tools können nacheinander ausgeführt werden und jeweils das Ergebnis des vorigen Tools nutzen.

Für die Definition eines Tools stellt jETI den *Tool-Konfigurator* zur Verfügung. Die Konfiguration eines Tools beginnt mit der Deklaration einer Java-Klasse und der entsprechenden Methode der Klasse, die das Tool implementiert. Die Klasse wird vom Toolserver instanziiert, sobald ein Nutzer das Tool verwendet. Anschließend

[15] `http://jeti.cs.uni-dortmund.de`

ruft er die deklarierte Methode auf, um das Tool auszuführen.

Im Allgemeinen benötigt ein Tool zur Ausführung eine Menge von Parametern. Ihre typgerechte Konfiguration erfolgt ebenfalls mithilfe des Tool-Konfigurators. Der Toolserver bildet die von einem Nutzer bei Aufruf übergebenen Argumente typgerecht auf die Parameter des Tools ab und übergibt sie ihm während des Aufrufs der deklarierten Methode. Spezielle Typen von Parametern sind InputFileReference und OutputFileReference für die Ein- und Ausgabedatei des Tools.

4. Konzept

Die Kernidee des dienstbasierten Layouts ist es eine Infrastruktur zu schaffen, die das Meta Layout von KIELER öffentlich verfügbar und vor allem unabhängig von einer Plattform nutzbar macht. Das KIELER Layout wird dadurch einer größeren Menge von Nutzern zugänglich. Um zu entscheiden, auf welche Weise die verschiedenen Aspekte von KWebS umgesetzt werden, befassen wir uns in Abschnitt 4.1 mit den Anforderungen an KWebS. Darauf entwerfen wir in Abschnitt 4.2 die Schnittstelle, durch die Nutzer mit dem dienstbasierten Layout interagieren. In Abschnitt 4.3 befassen wir uns mit den technischen Aspekten von KWebS. Das dienstbasierte Layout wird auf Grundlage eines Web Service implementiert. Im Allgemeinen stehen dafür zwei Architekturen zur Verfügung: SOAP und RESTful Web Services. In Abschnitt 4.3.1 betrachten wir beide Architekturen und treffen eine Entscheidung hinsichtlich der für KWebS besser geeigneten Variante. In Abschnitt 4.3.2 gehen wir abschließend auf die Wahl der benötigten Server-Plattform ein.

4.1. Anforderungen

Um die verschiedenen Möglichkeiten zur Implementierung von KWebS gegeneinander abzuwägen, formulieren wir in diesem Abschnitt die Anforderungen, die an die Implementierung gestellt werden.

Effizienz

Eine nahezu allgegenwärtige Anforderung im Bereich der Software-Entwicklung ist die der *Effizienz*. KWebS stellt das automatische Layout über ein Netzwerk zur Verfügung. Der durch den Netzwerk-Transfer entstehende zusätzliche Aufwand ist generell wenig zu beeinflussen. Der entscheidende Faktor ist, neben der Datenrate des Netzwerkes, die Menge an Daten, die bei einem Layout-Vorgang übertragen wird. Im Rahmen dieser Arbeit bedeutet Effizienz daher, dass zur eigentlichen Berechnung des Layouts seitens KWebS wenig zusätzlicher Aufwand durch benötigte *unterstützende Operationen* hinzukommt, welche sowohl auf Seiten des Nutzers als auch auf Seiten des Dienstes erforderlich sind.

Als unterstützende Operationen treten vor allem die für den Netzwerk-Transfer benötigte Serialisierung und Deserialisierung der übertragenen Modelle auf. Des Weiteren wird, abhängig des von einem Nutzer übergebenen Modells, auf Seiten von KWebS eine Transformation des Nutzermodells in ein KGraph-Modell benötigt, um das Layout mithilfe des Meta Layouts berechnen zu können. Ferner muss KWebS das Layout anschließend auf das Nutzermodell anwenden.

Das von KWebS für die Veröffentlichung des Dienstes verwendete Web Service-Framework hat ebenso Einfluss auf die Effizienz. Generell bietet ein Framework eine Abstraktion von der programmatischen Notation einer Anfrage und deren Bearbeitung durch einen Web Service in eine Notation auf Transportebene. Dieser Prozess erzeugt durch den benötigten Rechenaufwand ebenfalls zusätzlichen Aufwand, der zum eigentlichen Layout hinzukommt.

Wartbarkeit

Eine weitere Anforderung an die Implementierung von KWebS ist die der *Wartbarkeit*. Abschnitt 3.3 verdeutlicht die enge Kopplung von KIELER und insbesondere des Meta Layouts an die Eclipse-Plattform. Die Implementierung von KWebS soll sich ohne notwendige Spezialisierungen in das KIELER Projekt integrieren. Das bedeutet insbesondere, dass keine speziell für den Betrieb innerhalb des Dienstes angepassten Varianten von KIML oder der Layout-Algorithmen notwendig sein sollen.

Einfachheit

Die zuletzt formulierte Anforderung ist die der *Einfachheit*. Diese betrifft die Nutzung von KWebS, aber auch dessen Architektur. KWebS soll technisch auf einfache Art von Software-Systemen nutzbar sein und über eine leicht verständliche Schnittstelle verfügen. Des Weiteren soll KWebS wenig Komplexität zum KIELER Projekt hinzufügen. Diese liegt, neben der konkreten Implementierung, im Wesentlichen im gewählten Web Service-Framework und dessen Abhängigkeiten zu anderen Produkten, wie z.B. einem *Anwendungsserver*. Sie müssen nicht nur im Rahmen dieser Arbeit verstanden und möglicherweise gewartet werden, sondern auch darüber hinaus. Daher ist die Wahl eines in der Anwendung wenig komplexen Web Service-Frameworks von Bedeutung.

4.2. Entwurf der Schnittstelle

Die Nutzbarkeit und damit die Akzeptanz von KWebS hängt von der einfachen Nutzbarkeit und Integrierbarkeit der angebotenen Schnittstelle in Software-Systeme ab. Im Folgenden betrachten wir die verschiedenen Aspekte des Entwurfs der Schnittstelle.

4.2.1. Sitzungsorientierter oder sitzungloser Dienst

Ein Aspekt im Entwurf der Schnittstelle ist die Entscheidung für sitzungsbasierte oder sitzungslose Nutzung des Dienstes. Eine Designentscheidung hier hat direkten Einfluss auf die Operationen der angebotenen Schnittstelle und die Architektur von KWebS. Bei Verwendung eines sitzungsorientierten Dienstes kann das vom Nutzer während einer Sitzung bearbeitete Modell auf Seiten des Dienstes vorgehalten und gegebenenfalls auch persistent gespeichert werden. Ein dienstseitig vorgehaltenenes

Modell reduziert die für die Kommunikation notwendige Bandbreite und damit die Latenz der Operationen. Bei persistenter Speicherung ist das Modell vom Nutzer von verschiedenen Arbeitsplätzen aus erreichbar.

Ein denkbares Anwendungsszenario sieht wie folgt aus: Ein Nutzer beginnt eine Sitzung und hinterlegt das während der Sitzung verwendete Modell im Dienst. Alternativ kann er ein im Dienst gespeichertes Modell verwenden, je nachdem, ob Modelle dienstseitig persistent gespeichert werden oder nicht. Von diesem Punkt an findet die Kommunikation zwischen Nutzer und Dienst auf Basis von Operationen auf dem dienstseitigen Modell statt und die daraus folgenden strukturellen und layoutbezogenen Änderungen werden vom Dienst an den Nutzer kommuniziert.

Diese Art von sitzungsbasierter Interaktion findet man in webbasierten Anwendungen. In diesem Kontext ist es notwendig, die rechenintensiven Operationen, wie z.B. das Berechnen des Layouts, auf den Dienst auszulagern, da eine webbasierte Anwendung in einem Browser ausgeführt wird und daher im Allgemeinen nicht über die Ressourcen zur Berechnung des Layouts komplexer Modelle verfügt. KWebS ist primär für den Einsatz in "vollwertigen" Desktopanwendungen gedacht, und somit spielt die Reduzierung der nutzerseitig notwendigen Ressourcen eine untergeordnete Rolle. Eine sitzungsbasierte Schnittstelle kann den auf dem Kommunikationskanal anfallenden Verkehr reduzieren, jedoch steigt die Komplexität durch die erforderliche Synchronisierung der dienst- und nutzerseitigen Modelle signifikant. Dadurch wird der für die Nutzung des Dienstes notwendige Aufwand und auch die Komplexität des Dienstes erhöht. Dies steht im Widerspruch zur Anforderung der Einfachheit. Daher wird für KWebS eine sitzungslose Schnittstelle gewählt. Der Nachteil liegt darin, dass für jeden layoutbezogenen Aufruf das vollständige Modell übertragen werden muss. In diesem Sinne stellt die Entscheidung für einen sitzungslosen Dienst eine Abwägung zwischen der Anforderung der Einfachheit und der der Effizienz dar.

4.2.2. Synchrone oder asynchrone Kommunikation

Ein weiterer Aspekt im Entwurf der Schnittstelle ist die Entscheidung für *synchrone* oder *asynchrone* Kommunikation. Synchrones Layout blockiert, solange eine Anfrage in Bearbeitung ist, während asynchrones Layout das Weiterarbeiten des Anwenders gestattet. Asynchrones Layout ist gut geeignet für hochkomplexe Modelle, deren Berechnung eine längere Zeit in Anspruch nimmt. Derartige Fälle finden sich jedoch weniger im Kontext grafischer Modellierung und eher in Bereichen, in denen technische oder naturwissenschaftliche Gegebenheiten von Wichtigkeit sind, wie z.B. dem Layout von Leiterplatinen. Asynchrones Layout birgt auch die Gefahr der Desynchronisierung zwischen dem Modell des Anwenders und dem Modell, dessen Layout gegenwärtig berechnet wird. Nimmt der Anwender strukturelle Änderungen vor, so kann das durch den Dienst berechnete Layout nicht mehr auf das Modell des Nutzers angewendet werden, woraus die Notwendigkeit einer erneuten Anfrage an den Dienst entsteht. Anwenderseitige Änderungen an der visuellen Darstellung des Modells können durch das dienstseitig erzeugte Layout überschrieben werden. Da die Komplexität der Modelle im Umfeld grafischer Modellierung in einem Bereich liegt,

in dem das Berechnen des Layouts typischerweise nur einige Sekunden oder weniger in Anspruch nimmt, wird für KWebS synchrones Layout verwendet.

4.2.3. Die Schnittstelle von KWebS

Im Folgenden betrachten wir die Operationen, die KWebS zur Verfügung stellt. Zuvor diskutierten wir die Vor- und Nachteile einer sitzungsbasierten bzw. sitzungslosen Schnittstelle und synchronem bzw. asynchronem Layout, und trafen die Entscheidung hinsichtlich der sitzungslosen, synchronen Variante. Die Schnittstelle benötigt daher keine Operationen für das *Sitzungsmanagement*, zur *Modellmanipulation* oder für das Anfragen des Fortschritts einer Berechnung.

Layout eines Graphen

Die zentrale Operation von KWebS ist die Berechnung des Layouts eines Graphen. Ihre Signatur zeigt Listing 4.1, wobei wir zur übersichtlichen Darstellung im Folgenden die Java-Notation verwenden.

```
String graphLayout(String serializedGraph, String informat,
        String outformat, List<GraphLayoutOption> options);
```

Listing 4.1: Die graphLayout-Operation von KWebS

Ein Nutzer überträgt das Modell des Graphen, für den ein Layout berechnet werden soll, in serieller Darstellung. Er wird durch den Parameter serializedGraph repräsentiert. Um das Modell ableiten und auf ihm das Layout anwenden zu können, muss KWebS das ihm zugrundeliegende Metamodell und die verwendete Form der Serialisierung kennen, das *Format*. Dieses übermittelt der Nutzer durch den Parameter informat. Es ist notwendig, dass das übermittelte Format gültig ist, d.h. von KWebS unterstützt wird. Andernfalls bricht KWebS die Bearbeitung der Anfrage mit einem Fehler ab. Die unterstützten Formate sind Bestandteil der Dienst-Metadaten. Ein Nutzer kann sie durch Aufruf der Operation getServiceData ermitteln (siehe weiter unten).

Das Berechnen des Layouts ist nicht ausschließlich von der Struktur des Graphen abhängig. Es kann von einem Nutzer parametrisiert werden. Zum Beispiel ist es einem Nutzer im Allgemeinen möglich, den minimalen Abstand zwischen Knoten oder die gewünschte Orientierung von Kanten zu definieren. Der entscheidende Parameter ist der zu verwendende Algorithmus. Die Parametrisierung des Layouts kann auf globaler Ebene geschehen oder elementspezifisch sein. Für die Festlegung elementspezifischer Optionen müssen die entsprechenden Elemente des übertragenen Graphen mit den Optionen *annotiert* werden. Für globale Optionen bietet die graphLayout-Operation eine optionale Liste, der entsprechende Parameter ist options. Jede verwendete Layout-Option wird durch den Datentyp GraphLayoutOption repräsentiert, welcher in Listing 4.2 abgebildet ist.

```
public class GraphLayoutOption {
  public String id;
  public String value;
}
```

Listing 4.2: Der GraphLayoutOption-Datentyp

Eine Layout-Option besteht aus einem String-basierten Identifikator id und einem ebenfalls String-basierten Wert value. Die verfügbaren Layout-Optionen und insbesondere deren Identifikatoren und Datentypen sind Bestandteil der Metadaten von KWebS. Ein Nutzer kann sie durch Aufruf der Operation getServiceData in Erfahrung bringen.

Nach erfolgtem Layout liefert KWebS das berechnete Ergebnis in ebenfalls serieller Darstellung unter Verwendung desselben Formats, wie es der Nutzer zum Aufruf der Operation gewählt hat. Alternativ kann der Nutzer mit dem optionalen Parameter outformat der Operation graphLayout ein anderes Format angeben.

Dienst-Metadaten

Für die Inanspruchnahme des Dienstes müssen dem Nutzer eine Menge von Informationen bekannt sein:

- In welchem Format kann ich das Modell an den Layout-Dienst übergeben?
- Welche Layout-Algorithmen stehen zur Verfügung?
- Welche Layout-Optionen gibt es?
- Welche Layout-Optionen werden von welchen Algorithmen unterstützt?

Des Weiteren sind in KIELER die Algorithmen getypt nach Anbieter (z.B. KIELER, Graphviz, OGDF) und der generellen Art des Layouts, das sie berechnen (z.B. hierarchisch, orthogonal). Diese *Metadaten* sind zur Laufzeit von KWebS statisch. Da KIELER ein aktiv entwickeltes Projekt ist, können sich die Betriebsbedingungen jedoch ändern, indem Layout-Algorithmen, Layout-Optionen und unterstützte Formate weiterentwickelt, hinzugefügt oder auch entfernt werden. Daraus folgt, dass KWebS die Metadaten zur Startzeit erzeugen muss. Des Weiteren muss KWebS seine Metadaten einem Nutzer in geeigneter, maschinell verarbeitbarer Darstellung zur Verfügung stellen, um die einfache Integration des dienstbasierten Layouts in Software-Systeme zu ermöglichen.

Aufgrund der Eclipse-Natur von KIELER verwendet das Meta Layout die Extension Registry für die Veröffentlichung layoutbezogener Metadaten innerhalb von KIELER. Layoutbezogene Plug-Ins deklarieren Extensions zu vom Meta Layout hierfür bereitgestellten Extension-Points. Von technischer Seite her betrachtet verwendet die Eclipse-Plattform eine XML-Notation zur Speicherung der Extensions, die von

einem Plug-In definiert werden. Für die Darstellung seiner Metadaten verwendet KWebS ebenfalls eine auf XML basierende Notation, welche Abschnitt 5.4 vorstellt. Sie ist zum einen aus technischer Sicht einfach aus den entsprechenden Extensions zu generieren, zum anderen ist die Verarbeitung von XML auf praktisch allen Plattformen möglich. Ein Nutzer kann die Metadaten von KWebS durch Aufruf der Operation getServiceData in Erfahrung bringen, welche in Listing 4.3 zu sehen ist.

```
String getServiceData();
```

Listing 4.3: Die getServiceData-Operation von KWebS

Vorschaubilder

Ein Anwender grafischer Modellierung ist nicht notwendigerweise vertraut mit dem Layout von Graphen und den verschiedenen Ansätzen dazu. Um ihm die Auswahl eines Algorithmus zu erleichtern, sind den layoutbezogenen Plug-Ins Informationen hinterlegt, die ihre Arbeitsweise textuell und in Form eines Vorschaubilds beschreiben. Die textuellen Informationen sind in den Metadaten hinterlegt und können auf geeignete Weise von der GUI eines Software-Systems dargestellt werden. Das Vorschaubild ist aufgrund seiner binären Natur und seiner Größe nicht direkt in die Metadaten integriert, sie enthalten lediglich einen Verweis, der von einem Software-System für das Herunterladen des Bilds verwendet werden kann. Zu diesem Zweck bietet die Schnittstelle von KWebS die Operation getPreviewImage an, welche Listing 4.4 zeigt.

```
byte[] getPreviewImage(String previewImage);
```

Listing 4.4: Die getPreviewImage-Operation von KWebS

Der Parameter previewImage repräsentiert den Verweis auf das Bild. Die Operation liefert das Bild in binärer Darstellung. Viele Frameworks unterstützen die Erzeugung von Bildern aus deren binärer Repräsentation, und so bietet diese Operation eine einfache Möglichkeit für die Integration der Vorschaubilder in die Benutzerschnittstelle eines Software-Systems. Dies hilft Anwendern dabei, ein Verständnis für die Arbeitsweise der Algorithmen aufzubauen, und erleichtert ihnen dadurch die Wahl eines geeigneten Algorithmus für ein gegebenes Modell.

4.3. Architektur des Servers

In diesem Abschnitt befassen wir uns mit den technischen Aspekten von KWebS. Zunächst betrachten wir die zur Verfügung stehenden Web Service-Architekturen. Nachdem wir eine Entscheidung für eine Architektur getroffen haben gehen wir auf den Server ein, den wir zur Veröffentlichung des Dienstes benötigen.

4.3.1. KWebS als SOAP Web Service

Die Diskussion, ob ein RESTful oder ein SOAP Web Service die bessere Wahl darstellt, ist so alt wie die Architekturen selbst. Paul Prescod diskutierte bereits im Jahre 2002 diese beiden Architekturen in seiner Veröffentlichung „Roots of the REST/SOAP Debate“ [19]. Ein gewichtiges Argument für einen RESTful Web Service ist dessen einfache Nutzbarkeit, da im Grunde genommen auf Nutzerseite nur eine HTTP-Verbindung zum Service aufgebaut werden, und daher nur ein HTTP-Client auf der Nutzerplattform vorhanden sein muss. Die Schnittstelle zur Interaktion mit den vom Service bereitgestellten Ressourcen findet anhand der durch die HTTP-Verben standardisierten Schnittstelle statt. Die Wahl eines leichtgewichtigen Formates auf der Transportschicht kann den durch die Verarbeitung einer Anfrage anfallenden Aufwand im Vergleich zu einem SOAP Web Service reduzieren.

Die Verwendung eines RESTful-Web Services bedeutet jedoch im Allgemeinen einen höheren manuellen Implementierungsaufwand seitens des Nutzers, da aufgrund der einheitlichen Schnittstelle keine servicespezifische, automatische Code-Generierung möglich ist. Des Weiteren ist eine detaillierte Kenntnis über die hierarchische Organisation der Ressourcen eines Service, deren gültiger Repräsenationen und auch der Semantik der Query-Parameter der einer URI zugeordneten Ressource notwendig. Diese servicespezifischen Informationen liegen meist in textueller Form vor und müssen manuell umgesetzt werden. Dies erhöht die Komplexität der Nutzung. Es gibt Bemühungen, für RESTful Web Services eine IDL zu definieren. So kann der aktuelle Standard der WSDL RESTful Web Services beschreiben, und mit der Web Application Description Language (WADL)[1] existiert eine weitere, auf RESTful Web Services spezialisierte IDL. Diese Standards sind jedoch umstritten, nicht zuletzt, da die Verwendung einer IDL das Architekturprinzip der einheitlichen Schnittstelle verletzt, und daher ein RESTful Web Service genau genommen nicht mehr RESTful ist.

Der Aufwand für die Nutzung eines SOAP Web Service ist im Vergleich dazu geringer. Da die Schnittstelle eines SOAP Web Service durch WSDL maschinenlesbar spezifiziert ist, kann der zu seiner Verwendung notwendige Code automatisch generiert werden. Generell wird für die Nutzung eines SOAP Web Service ein Framework benötigt. Dies ist für alle gängigen Plattformen verfügbar, sodass daraus keine Einschränkung für die breite Nutzbarkeit entsteht.

Die (nach Pautasso et al. *vermeindliche* [20]) Einfachheit der RESTful Web Services ist auch auf die Tatsache zurückzuführen, dass weniger architekturelle Entscheidungen für dessen Verwendung notwendig sind. Während z.B. SOAP Web Services konzeptionell nicht an ein konkretes Transportprotokoll gebunden sind[2] und damit architekturell gesehen mehr Freiheit bieten, wird im Kontext der RESTful Web Services praktisch ausschließlich HTTP verwendet. Da RESTful Web Services auf einer einheitlichen Schnittstelle basieren, entfällt im Vergleich zu SOAP Web Services der

[1] `http://wadl.java.net`

[2] Die Spezifikation von SOAP definiert eine Bindung an HTTP, die in der Praxis auch häufig verwendet wird.

Entwurf der Schnittstelle. Dies ist jedoch nur zum Teil wahr: Die einheitliche Schnittstelle definiert zwar die allgemeine Semantik der Interaktion mit einer Ressource, ein konkretes Schema für die Adressierung der von einem Service zur Verfügung gestellten Ressourcen muss dennoch sorgfältig entwickelt werden. Dazu gehört es auch zu definieren, welche Repräsentationen sie unterstützen und wie eventuell notwendige Query-Parameter gestaltet sind. Ein weiterer wichtiger Faktor im Vergleich von Pautasso et al. ist die Unterstützung von Dienstgüte-Kriterien. Für SOAP Web Services existieren diverse Standards wie z.B. WS-Transaction oder auch WS-Reliable Messaging, die verschiedene Kriterien der Dienstgüte umsetzen. RESTful Web Services sind im Allgemeinen auf proprietäre Implementierungen angewiesen.

Die Entscheidung, KWebS als RESTful oder SOAP Web Service anzubieten, kann von der Art der geplanten Nutzung her betrachtet werden. Pautasso et al. geben eine grobe Richtlinie:

- RESTful Web Services sind gut geeignet für *ad hoc*-Integration über das Internet. Als Beispiel führen sie *Mashups* an.
- SOAP Web Services sind besser geeignet für eine langlebige Integration in Software-Systeme mit anspruchsvollen Anforderungen bezüglich der Dienstgüte.

Ein Mashup ist eine meist Browser-basierte Anwendung die verschiedene, nicht miteinander in Beziehung stehende Ressourcen wie z.B. Web Services miteinander kombiniert, um eine höherwertige Funktionalität bereitzustellen. Ein Beispiel sind Preissuchmaschinen. Sie verwenden öffentliche APIs verschiedener Anbieter für die Ermittlung ihres Preises zu einem gegebenen Produkt und bieten ihren Nutzern den Mehrwert des Vergleichs.

KWebS lässt sich im Grunde keiner der beiden Kategorien vollständig zuordnen. Der Dienst ist für die dauerhafte Integration in Software-Systeme gedacht, und nicht für eine „Ad Hoc"-Integration im Stile eines Mashups. Aus diesem Blickwinkel scheint ein SOAP Web Service besser geeignet zu sein als ein RESTful Web Service. Allerdings hat KWebS keine Anforderungen in Bezug auf die Dienstgüte zu erfüllen, welche nur mit den technischen Möglichkeiten eines SOAP Web Service zu realisieren sind. Der grundlegende Aspekt der Sicherheit und Vertraulichkeit kann z.B. auf Transportschicht durch Verwendung von HTTPS umgesetzt werden.

Ein weiterer Gesichtspunkt für die Wahl einer Web Service-Architektur ist deren Effizienz, da Effizienz eine der Anforderungen für die Implementierung von KWebS ist. Ein Argument für die Verwendung eines RESTful Web Service ist, dass auf Transportebene, im Gegensatz zu SOAP Web Services, ein nicht auf XML basierendes Nachrichtenformat gewählt werden kann, und damit der durch das Parsen von XML erzeugte Aufwand entfällt. Dieses Argument ist für KWebS nicht von Bedeutung:

- Eine SOAP-Nachricht ist im Kern ein sehr einfaches XML-Dokument, bestehend aus Envelope, Header, Body, und der in XML abgebildeten Anfrage. Der

zusätzliche Aufwand ihres Parsens fällt daher kaum ins Gewicht. Der wesentliche Aufwand entsteht erst durch die Verwendung weiterer Standards, wie z.B. WS-Security, welche zum einen die Komplexität der Nachricht durch hinzugefügte Header-Blöcke erhöhen, und zum anderen Aufwand durch deren Verarbeitung erzeugen.

- KWebS erwartet Modelle in serieller Notation. Die Bearbeitung einer Layout-Anfrage umfasst, in Abhängigkeit der vom Nutzer gewählten Notation und des gewünschten Algorithmus, ausgehend von der seriellen Notation folgende Schritte:

  1. das Parsen des Nutzermodells aus der seriellen Notation,
  2. das Ableiten eines strukturell identischen KGraph-Modells,
  3. die Annotation des KGraph-Modells mit den vom Nutzer angegebenen Layout-Optionen,
  4. das Berechnen des Layouts des KGraph-Modells,
  5. das Anwenden des Layouts auf das Nutzermodell und
  6. das Serialisieren des Nutzermodells.

  Wie diese Auflistung zeigt, sind eine Menge von nicht trivialen Operationen für die Bearbeitung einer Anfrage notwendig, von denen die eigentliche Berechnung des Layouts nur einen Teil der Gesamtdauer in Anspruch nimmt. Der wesentliche Aufwand des dienstbasierten Layouts entsteht somit durch die Verarbeitung der Anfrage selbst.

Der durch das Marshaling und Unmarshaling der auf Transportschicht ausgetauschten Nachrichten seitens eines SOAP Web Service erzeugte zusätzliche Aufwand ist daher für die Verarbeitung einer Layout-Anfrage nicht von Bedeutung.

Warum SOAP?

Wenn wir uns die zuvor genannten Gesichtspunkte der beiden Web Service-Architekturen anschauen stellen wir fest, dass jede ihre Vorzüge hat und es vom Anwendungsfall abhängig ist, welche Architektur letzten Endes die bessere Wahl darstellt. Im Falle von KWebS spielt die effizientere Darstellung und Verarbeitung der RESTful Web Services von Nachrichten auf Transportebene eine untergeordnete Rolle, da der zusätzliche Aufwand der SOAP Web Services minimal ist und die Effizienz des Dienstes somit nicht wesentlich beeinflusst wird. Ein SOAP Web Service vereinfacht die Integration in Software-Systeme durch die Möglichkeit der automatisierten Generierung des dazu notwendigen Client-Codes und kommt somit der Anforderung der einfachen Nutzbarkeit nach. KWebS wird daher auf Grundlage eines SOAP Web Service implementiert.

Ein Wort zur Schnittstelle

Die in Abschnitt 4.2 vorgestellte Schnittstelle stellt im Grunde genommen einen Kompromiss zwischen den beiden Architekturen dar der seinen Teil dazu beiträgt, dass der Aufwand eines SOAP Web Service nicht zum tragen kommt. Sie modelliert das übertragene Modell in serieller Notation und macht nicht von der Möglichkeit Gebrauch, ein eigenständiges Graphenmodell abstrakt und damit plattformneutral zu definieren. Gleiches gilt für die unterstützten Formate, für welche die Schnittstelle keine gültigen Ausprägungen definiert. Dieser Parameter wäre bei einem eigenständig definierten Graphenmodell auch nicht erforderlich.

Der Grund dafür ist, dass KWebS offen gestaltet ist für die Unterstützung beliebiger Modelle. Die Definition eines Graphenmodells seitens der Schnittstelle würde lediglich eine weitere proprietäre Notation einführen und damit die Verwendung des dienstbasierten Layouts erschweren. Nutzer müssten für ihre Modelle grundsätzlich eine Transformation in das von KWebS verwendete Modell implementieren, und natürlich auch in entgegengesetzter Richtung. Durch die Verwendung serieller Notationen ist der Dienst grundsätzlich in der Lage, beliebige Modelle zu berechnen. Ferner bedeutet die Definition der Notationen als String seitens der Schnittstelle keine Beeinträchtigung der Plattformunabhängigkeit, da dies ein Basistyp praktisch aller Plattformen ist.

Die gültigen Formate werden ebenfalls seitens der Schnittstelle nicht eingeschränkt. Es ist denkbar, dass KWebS in Zukunft weitere Formate unterstützen wird als die, die im Rahmen dieser Arbeit implementiert wurden. Diese Formate müssten dann im Vertrag definiert werden. Zwischen dem Vertrag und den aus ihm generierten Client-Implementierungen besteht jedoch eine enge Kopplung. Änderungen am Vertrag führen zu Inkompatibilitäten, die eine erneute Generierung des zur Nutzung verwendeten Codes erforderlich machen. Dies mag während des Entwicklungsprozesses eines Software-Systems als akzeptabel erscheinen. In der Praxis hat dies zur Folge, dass eine große Anzahl an Anwendungen (die sich bereits im produktiven Einsatz befinden) aktualisiert werden müssen, ohne einen direkten Nutzen daraus zu ziehen, da das von ihnen verwendete Format ja bereits unterstützt wird. Die Verwendung eines nicht eingeschränkten Parameters zur Deklaration des Formats seitens der Schnittstelle sorgt für die gegenüber seinen Nutzern transparente Erweiterbarkeit von KWebS.

Umsetzung von KWebS mit JAX-WS

Für die Veröffentlichung des Dienstes wird ein Framework benötigt. Es gibt viele Alternativen wie z.B. Apache Axis2[3], Apache CXF[4] oder auch Banshee[5]. Im Rahmen dieser Arbeit fällt die Wahl auf die Referenzimplementierung von JAX-WS. Sie gilt als produktionsreif und bietet somit eine stabile Grundlage für KWebS.

[3] http://axis.apache.org/axis2/java/core
[4] http://cxf.apache.org
[5] http://www.bansheeframework.com

Sicherheit und Vertraulichkeit

Die von einem Nutzer übertragenen Modelle können vertraulicher Natur sein. KWebS realisiert diesen Aspekt auf Transportschicht durch Verwendung von HTTPS.

HTTPS nutzt das Transport Layer Security (TLS)-Protokoll zum Aufbau einer abhörsicheren, nicht manipulierbaren Verbindung. TLS ist eine Weiterentwicklung von Secure Socket Layer (SSL) und nicht dazu abwärtskompatibel. Normalerweise verwendet ein Client HTTPS zur Sicherstellung der Authentizität eines Servers. Die Authentifizierung des Clients seitens des Servers ist ebenso möglich.

Für die gesicherte Kommunikation müssen sich Client und Server auf einen Schlüssel einigen, den sie für die Sicherung der übertragenen Informationen verwenden. Der Aufbau einer Verbindung beginnt daher mit einem *handshake*. Im ersten Schritt einigen sich Client und Server auf die während der Sitzung verwendeten Algorithmen zur Verschlüsselung der Daten und auf eine Hash-Funktion zur Feststellung deren Echtheit. Client und Server können über unterschiedlich starke Varianten der Algorithmen verfügen, von denen sie die stärkste Kombination aushandeln, die sie beide unterstützen. Der Server übermittelt anschliessend sein *Zertifikat* an den Client. Im Allgemeinen ist dieses von einer auf globaler Ebene vertraulichen Instanz *signiert*, einer Certificate Authority (CA), sodass der Client die Echtheit des Zertifikats und damit die des Servers sicherstellen kann. Dieser verfügt über einen *asymmetrischen* Schlüssel, von dem der öffentliche Teil im Zertifikat enthalten ist. Der Client nutzt diesen für die gesicherte Übermittlung einer zufällig generierten Zahl, dem *pre-master secret*, anhand derer Client und Server das *master secret*, und daraus den von beiden Seiten für die Sitzung verwendeten *symmetrischen* Schlüssel generieren.

Ein weiterer Gesichtspunkt bezüglich der Vertraulichkeit ist die beabsichtigte oder nicht beabsichtige Weitergabe der vom Nutzer übermittelten Modelle an Dritte. KWebS hält keine Informationen über die Nutzermodelle vor, seien es die Modelle selbst, oder anders geartete Daten (wie z.B. Statistiken), welche von Dritten für den Zugriff auf vertrauliche Informationen genutzt werden können. Aufgrund der Open Source-Natur von KWebS ist dies jedoch eine Limitierung, die vom Grundsatz her durch entsprechende Anpassungen leicht umgangen werden kann.

Alternativer Ansatz mit jETI

Die Hauptlösung dieser Arbeit basiert auf JAX-WS. Sie verwendet eine sitzungslose, asynchrone Schnittstelle und kommt dadurch der Anforderung der Einfachheit nach. Diese Entscheidung im Entwurf des Dienstes stellt eine Abwägung zwischen Effizienz und Einfachheit dar, die in gewissen Anwendungsfällen eine starke Beeinträchtigung der Effizienz darstellt. Möchte ein Nutzer z.B. verschiedene Algorithmen auf demselben Modell anwenden, um deren Layouts miteinander zu vergleichen, so muss er das Modell für jeden Layout-Vorgang erneut übertragen. Dies geht, insbesondere für komplexe Modelle, mit einer deutlich erhöhten Latenz einher.

Die in Abschnitt 3.6 beschriebene jETI-Plattform dient daher im Rahmen dieser Arbeit als Grundlage für eine alternative Implementierung des Dienstes. Er wird als

Tool implementiert und erwartet das Nutzermodell in Form einer virtuellen Datei, das berechnete Layout legt er ebenfalls in einer virtuellen Datei ab. Aufgrund der sitzungsbasierten Natur von jETI werden diese Dateien, insbesondere die das Nutzermodell enthaltende, für den Zeitraum der Sitzung vorgehalten. Wenn ein Nutzer den jETI-Dienst für die Berechnung eines Layouts in Anspruch nimmt, kann er die Ein- und Ausgabedateien des Tools deklarieren. Er kann sein Modell daher einmalig an den Dienst übermitteln und darauf mehrmals das Layout anwenden.

4.3.2. Server als Eclipse Rich Client Application

Das Publizieren des automatischen Layouts von KIELER bedeutet im Kern, das Meta Layout von KIELER zu veröffentlichen. KIELER ist als Eclipse-Rich Client Application (RCA) konzipiert, und so folgt das Meta Layout naturgemäß den Konzepten der Eclipse-Plattform. Aufgrund der OSGi-Natur von Eclipse erzeugt dies wenigstens die Abhängigkeit zu einer OSGi-konformen Plattform. Das Meta Layout verwendet den Extension-Point-Mechanismus, welcher eine Erweiterung des OSGi-Standards darstellt, um eine lose Kopplung der Layout-Algorithmen an KIELER zu erreichen. Einige der Layout-Algorithmen verwenden *Preference Stores* zur Speicherung kritischer Parameter für ihre Verwendung. Diese Umstände sind bei der Wahl einer Architektur für den Server zu beachten. Generell bestehen zwei Möglichkeiten, KWebS zu implementieren:

- die Entkopplung von KIML und der Layout-Algorithmen von der Eclipse-Plattform, und dadurch letzten Endes von OSGi, oder
- das Verwenden einer Plattform, die OSGi unterstützt.

Es ist denkbar, die Abhängigkeiten des Meta Layouts zur Eclipse-Plattform aufzulösen. Dies würde bedeuten, zentrale Konzepte der Eclipse-Plattform erneut zu implementieren, um von ihr nicht abhängig zu sein. Dieser Ansatz steht grundsätzlich im Widerspruch mit der Verwendung einer RCP, da eine RCP als Grundlage eines Software-Projektes die Implementierung durch Bereitstellung geeigneter Infrastrukturen vereinfachen soll. Um weiterhin in KIELER den vollen Nutzen aus der Eclipse-Plattform ziehen zu können, würde eine Entkopplung somit einen getrennten Entwicklungszweig des Meta Layouts bedeuten. Die Weiterentwicklung und Pflege von zwei getrennten Entwicklungszweigen mit thematisch identischer Funktionalität steht grundsätzlich im Widerspruch mit der Forderung der Einfachheit und auch der der Wartbarkeit.

Besser geeignet ist somit die zweite Möglichkeit, auch wenn sie die Anzahl verfügbarer Plattformen einschränkt (die Industrie ist derzeit in vielen Bereichen im Begriff, vorhandene Produkte, insbesondere Anwendungsserver, auf den OSGi-Standard umzustellen). Zwei der möglichen Plattformen sind GlassFish, Oracles Referenzimplementierung eines Java EE-kompatiblen Anwendungsservers, und Eclipse Virgo[6],

[6] `http://www.eclipse.org/virgo`

ein auf der Eclipse-Plattform aufgebauter Anwendungsserver. Die Verwendung eines Anwendungsservers erhöht jedoch die Komplexität der Gesamt-Architektur, da eine Technologie eingeführt wird, die auch über den Rahmen dieser Arbeit hinweg gepflegt werden muss. Des Weiteren werden viele der von einem Anwendungsserver gebotenen Aspekte von KWebS nicht benötigt, wie etwa Persistenz. Von der Verwendung eines Anwendungsservers für den Dienst wird daher abgesehen, da die der Architektur hinzugefügte Komplexität für KWebS keinen Mehrwert zur Folge hat und im Gegensatz zur Forderung der Einfachheit steht.

Diese Arbeit verfolgt den Ansatz, für die Implementierung von KWebS die Eclipse-RCP zu verwenden. Da KIELER selbst als Eclipse-RCP konzipiert ist, wird das Meta Layout so auf natürliche Weise in KWebS eingebettet. Diese Wahl kommt der Forderung der Einfachheit nach, denn es werden keine Abhängigkeiten zu weiteren Produkten eingeführt. Des Weiteren wird die Forderung der Wartbarkeit erfüllt, da das Meta Layout in KWebS in derselben Plattform Anwendung findet, wie in KIELER selbst. Daher sind keine dienstbezogenen Anpassungen notwendig.

5. Implementierung

In diesem Kapitel betrachten wir die Implementierung von KWebS. Wie wir in Abschnitt 4.3.2 sahen, ist KWebS als Eclipse-RCA konzipiert. Die RCA stellt den Server dar, der zur Veröffentlichung der angebotenen *Dienstvarianten* dient. In Abschnitt 5.1 lernen wir daher zunächst die Implementierung des Servers kennen.

KWebS verwendet das KIELER Meta Layout zur Berechnung der Nutzermodelle auf Grundlage des KGraph-Metamodells. Die Schnittstelle des Dienstes hingegen erlaubt es Nutzern, ihre Modelle in unterschiedlichen Formaten zu übermitteln. Die notwendige Anbindung von KWebS an das Meta Layout betrachten wir in Abschnitt 5.2. Die dort vorgestellte Architektur stellt die auf Seiten des Servers verwendete Infrastruktur zur Berechnung des Layouts beliebiger Nutzermodelle zur Verfügung. Des Weiteren lernen wir die von KWebS am Meta Layout vorgenommenen Erweiterungen kennen. Sie sind zum einen notwendig, um den Nutzern von KWebS geeignete Metadaten bezüglich des gebotenen Layouts zur Verfügung zu stellen, und zum anderen für die Integration des dienstbasierten Layouts in KIELER.

Darauf folgend betrachten wir in Abschnitt 5.3 die Implementierung der Dienstvarianten. Sie dienen als Adapter zur zuvor vorgestellten Layout-Infrastruktur.

In Abschnitt 5.4 lernen wir abschließend das auf Ecore basierende Metamodell ServiceData kennen, welches KWebS für die Darstellung seiner Metadaten verwendet.

5.1. Implementierung des Servers

In diesem Abschnitt betrachten wir den Server. Er dient zur Veröffentlichung der Dienstvarianten und besteht im Kern aus drei Klassen, welche Abbildung 5.1 zeigt.

Die Eclipse-Plattform stellt die Schnittstelle IApplication als Eintrittspunkt für eine RCA zur Verfügung. Sie definiert zwei Methoden, die von der Anwendungsklasse einer RCA realisiert werden müssen. Sie dienen zur Handhabung ihres Lebenszyklus:

- Object start(IApplicationContext context) wird von der Eclipse-Plattform aufgerufen, um die Anwendung zu starten. Der Parameter context definiert die Umgebung der Anwendung, wie z.B. ein definiertes *Branding* oder die beim Start übergebenen Argumente der Kommandozeile.
- void stop() beendet die Anwendung. Sie wird ebenfalls durch die Eclipse-Plattform aufgerufen.

KWebS realisiert diese Schnittstelle mittels der Klasse Application und verwendet die start-Methode für notwendige Initialisierungen:

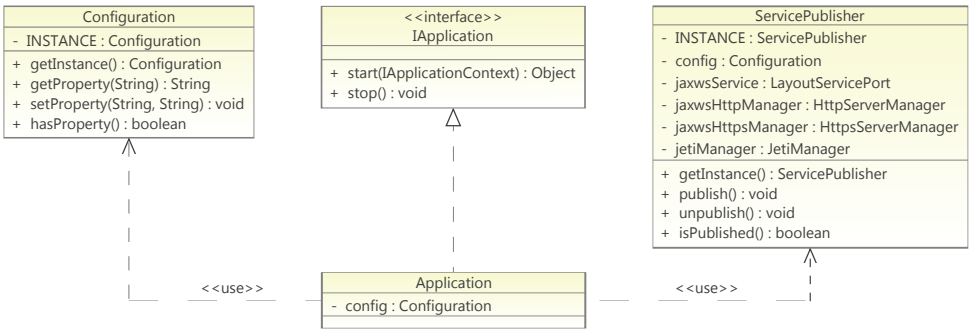


Abbildung 5.1.: Die Klassenstruktur des Servers von KWebS

- das Laden der Konfiguration,
- das Starten des Loggings,
- das Starten der Management-Schnittstelle,
- das Konfigurieren der Layout-Algorithmen und
- das Veröffentlichen der Dienstvarianten.

Im Folgenden betrachten wir die einzelnen Schritte des Startvorgangs.

5.1.1. Startvorgang des Servers

Der Zweck der Singleton-Klasse Configuration ist es, die Konfiguration des Servers anwendungsweit zentralisiert zur Verfügung zu stellen. Sie wird zur Startzeit vom Server initialisiert. Die Initialisierung beginnt mit dem Laden der Standardkonfiguration, welche im Plug-In des Servers enthalten ist. Die Standardkonfiguration enthält für die meisten Konfigurationsparameter Standardbelegungen, mit denen der Server grundsätzlich einsetzbar ist. Der nächste Schritt während der Initialisierung überprüft, ob die Konfiguration des Servers durch den Betreiber angepasst wurde. Die Anpassung erfolgt durch eine Datei mit Namen kwebs.user. Ist dies der Fall, werden die vom Betreiber vorgenommenen Einstellungen in die Server-Konfiguration übernommen, wobei die entsprechenden Einstellungen der Standardkonfiguration ersetzt werden. Abschließend wird überprüft, ob beim Start des Servers Parameter der Kommandozeile verwendet wurden, um die Konfiguration des Servers anzupassen. Auch in diesem Fall werden die Einstellungen übernommen. Die Liste der verfügbaren Optionen zur Konfiguration des Servers ist in Anhang B.1 angegeben.

Das Logging wird gestartet, nachdem die Konfiguration des Servers initialisiert wurde, denn es ist durch Parameter der Konfiguration beeinflussbar. Dies gilt ebenso für die Management-Schnittstelle. Sie ist ausschließlich lokal vom Server aus nutzbar, der dazu verwendete Port wird durch die Konfiguration festgelegt.

Der darauf folgende Schritt setzt Parameter, die für eine ordnungsgemäße Funktion einiger Layout-Algorithmen erforderlich sind:

- Auf Graphviz basierende Algorithmen benötigen den Pfad zur ausführbaren Datei „dot“.
- Graphviz und auch OGDF stellen binäre Bibliotheken bereit, deren Anbindung das Meta Layout streambasiert handhabt. Die notwendigen Timeouts für die Kommunikation mit den Bibliotheken können angepasst werden.

In KIELER erfolgt deren Konfiguration anhand einer Preference Page. Der Server ist *Headless* umgesetzt, d.h. er stellt zur Laufzeit keine *Workbench* und damit keine GUI zur Verfügung. Somit kann die Konfiguration der Parameter nicht anhand der Prefererence Page erfolgen. Daher setzt der Server sie während seiner Initialisierung programmatisch. Für die Timeouts von Graphviz und OGDF sind in der Standardkonfiguration bereits Werte hinterlegt. Die Definition des Pfades zur ausführbaren Datei „dot“ ist von der Installation von Graphviz abhängig. Sie erfolgt daher durch den Betreiber in der Konfigurationsdatei kwebs.user.

Nach dem Setzen der Parameter der Algorithmen folgt abschließend die Veröffentlichung der Dienstvarianten, welche auf Grundlage von zwei Plattformen angeboten werden: Der SOAP-Dienst verwendet die Referenzimplementierung von JAX-WS, der jETI-Dienst die Java Electronic Tool Integration-Plattform. Der jETI-Dienst verwendet auf der Transportschicht ein proprietäres Protokoll. Im Gegensatz dazu wird der SOAP-Dienst, in Abhängigkeit der Konfiguration des Betreibers, über HTTP und HTTPS veröffentlicht. Es liegen daher drei Dienstvarianten vor. Die Singleton-Klasse ServicePublisher dient ihrer serverweit zentralisierten Verwaltung.

5.1.2. Management der Dienstvarianten

Das Management der durch den Server angebotenen Dienstvarianten erfolgt serverweit durch die Klasse ServicePublisher. Sie verwendet für jede Variante einen entsprechenden *Manager*, um deren Lebenszyklus zu steuern:

- den HttpServerManager für SOAP/HTTP,
- den HttpsServerManager für SOAP/HTTPS und
- den JetiServerManager für jETI.

Abbildung 5.2 zeigt die Klassenstruktur der Manager. Jeder Manager erbt von einer abstrakten Basisklasse AbstractServerManager, welche ihrerseits das Interface IServerManager realisiert. Jeder Manager stellt die folgenden zwei Methoden zur Steuerung der von ihm veröffentlichten Dienstvariante zur Verfügung:

- void publish(Object serviceObject) veröffentlicht die gegebene Instanz serviceObject der Implementierung einer Dienstvariante im Endpunkt, der durch den Manager verwaltet wird.

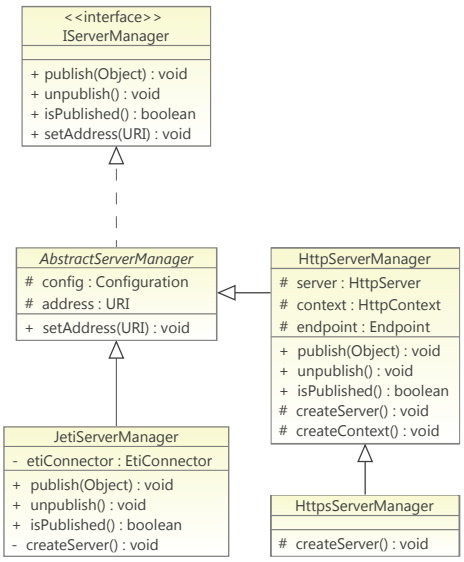


Abbildung 5.2.: Die Klassenstruktur der Server-Manager

- void unpublish() nimmt die Veröffentlichung zurück.

Zwei weitere Methoden dienen zur Feststellung des Status einer Variante und der Konfiguration des Endpunkts, unter dem sie veröffentlicht wird:

- boolean isPublished() stellt fest, ob die Dienstvariante gegenwärtig veröffentlicht ist.
- void setAddress(URI serviceAddress) setzt die Adresse, den Port und den Kontext des Endpunkts, unter dem der Manager die Dienstvariante veröffentlicht. Adresse, Port und Kontext werden aus der übergebenen URI abgeleitet.

ServicePublisher verwendet die Konfiguration des Servers um festzustellen, welche Dienstvarianten seitens des Betreibers zur Veröffentlichung ausgewählt wurden. Aufrufe an die Methoden publish, unpublish und isPublished des ServicePublisher werden von ihm auf geeignete Weise an die entsprechenden Methoden der Manager delegiert.

5.2. Anbindung der Dienstvarianten an das Meta Layout

In diesem Abschnitt befassen wir uns mit der Anbindung der durch den Server veröffentlichten Dienstvarianten an das Meta Layout von KIELER. Abbildung 5.3 gibt einen Überblick über die auf Seiten des Servers verwendete Layout-Infrastruktur. Die Implementierungen der Dienstvarianten, JaxWsService und JetiService, dienen lediglich der Adaption der plattformspezifischen Art ihrer Nutzung an die Layout-Infrastruktur des Servers. Wir lernen sie in Abschnitt 5.3 kennen. Die Klassen AbstractService und ServerLayoutDataService stellen die eigentliche Funktionalität bereit.

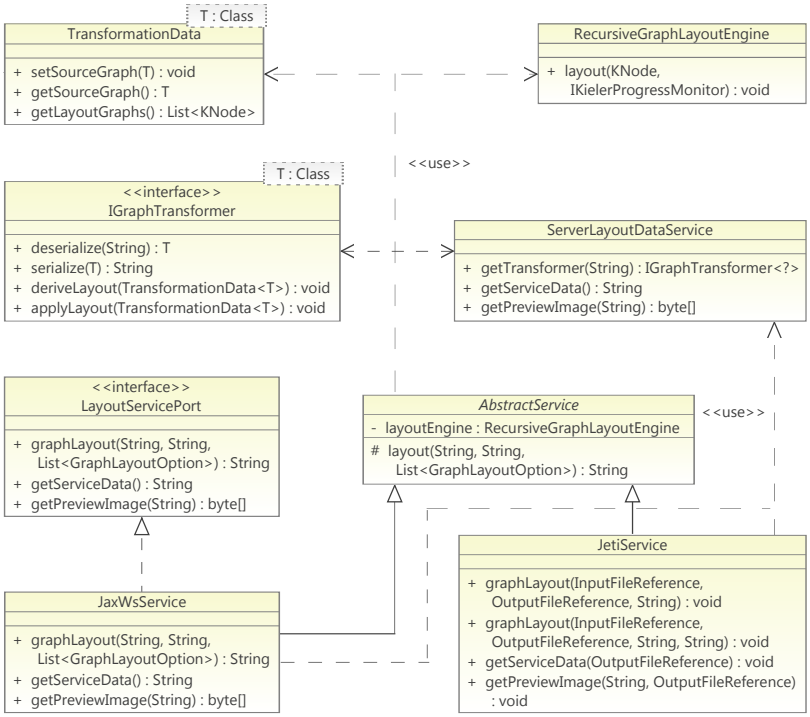


Abbildung 5.3.: Die Layout-Infrastrukur des Servers

Die von einem Nutzer gestellten Anfragen sind grundsätzlich zweierlei Natur:

- die Berechnung des Layouts eines Modells durch Aufruf der Operation graphLayout, oder
- die Anfrage von Metadaten durch Aufruf der Operationen getServiceData und getPreviewImage.

Im Folgenden befassen wir uns mit diesen beiden Aspekten der auf Seiten des Servers verwendeten Layout-Infrastruktur.

5.2.1. Verlauf eines dienstseitigen Layouts

KWebS verwendet für die Berechnung eines Layouts das Meta Layout von KIELER. Der Verlauf eines dienstseitig berechneten Layouts wird von Abbildung 5.4 dargestellt.

Für die Anbindung an das Meta Layout verwendet KWebS dessen Klasse RecursiveGraphLayoutEngine. Sie führt die Berechnung des Layouts eines KGraph-Modells durch. Das von einem Nutzer übergebene Modell kann in verschiedenen Formaten vorliegen, d.h. ihm können verschiedene Metamodelle zugrundeliegen und es können veschiedene Arten der Serialisierung verwendet werden. Um das Layout berechnen

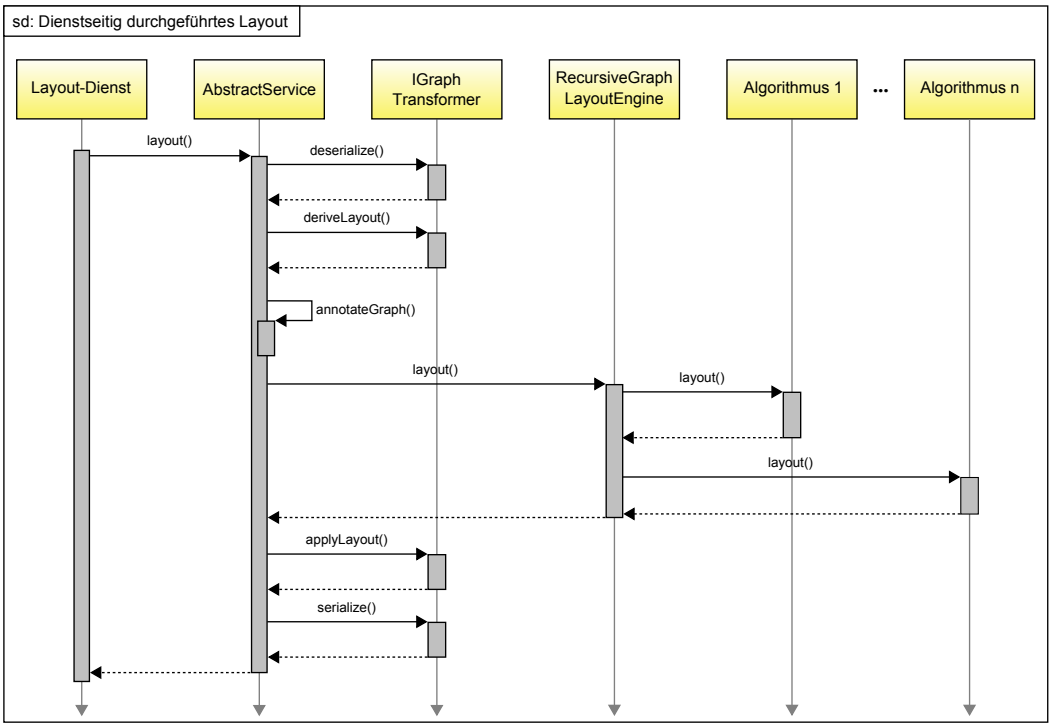


Abbildung 5.4.: Verlauf eines dienstseitigen Layouts

zu können, muss daher, nach der Deserialisierung, eine Übersetzung des Nutzermodells in ein KGraph-Modell stattfinden. Die Klasse AbstractService verwendet daher zum Format des Nutzermodells kompatible Realisierungen der Schnittstelle IGraph-Transformer, die *Transformer*, um das Nutzermodell aus der seriellen Darstellung zu lesen und ein dazu strukturell identisches KGraph-Modell abzuleiten. Danach wird, sofern der Nutzer in der Anfrage Layout-Optionen definiert hat, das KGraph-Modell mit diesen annotiert.

Die Layout-Optionen sind an spezifische Typen von Elementen des KGraph-Modells gebunden. Während eine Option nur auf Kanten anwendbar ist, gilt eine andere nur für Knoten. Diese Typisierung der Optionen wird während der Annotation des KGraph-Modells berücksichtigt, es werden aber grundsätzlich alle zu einer Option typkompatiblen Elemente annotiert. Die Annotation eines spezifischen Modellelements mit einer Option muss durch den Nutzer im Modell geschehen. Elementspezifische Annotationen werden nicht durch die dienstseitige Annotation mit den beim Aufruf übergebenen Optionen ersetzt.

Es folgt die Berechnung des Layouts des abgeleiteten KGraph-Modells. Im Anschluss verwendet AbstractService erneut den zuvor ermittelten Transformer, um das berechnete Layout auf das Nutzermodell anzuwenden. Abschließend verwendet sie einen dem Ausgabeformat entsprechenden Transformer, um das Modell des Nutzers

für die Rückgabe an ihn zu serialisieren.

Implementierte Transformationen

Die Schnittstelle von KWebS ist für die Unterstützung beliebiger Formate ausgelegt, um Software-Systemen die Nutzung des dienstbasierten Layouts einfach zu gestalten. Neben dem KGraph unterstützt der Dienst gegenwärtig vier weitere Formate:

- Graphviz DOT[1],
- GraphML[2],
- Open Graph Markup Language (OGML)[3] und
- Matrix Market[4].

5.2.2. Erweiterungen des Meta Layouts

Einem Nutzer des dienstbasierten Layouts müssen die Metadaten des Dienstes bekannt sein. Wesentlicher Bestandteil dieser Daten sind, neben den Formaten die er zum Austausch von Modellen unterstützt, die Metadaten des Layouts. KIELER verwendet die Singleton-Klasse LayoutDataService, um die Metadaten zur Laufzeit zentralisiert und programmatisch nutzbar zur Verfügung zu stellen. Für dienstbasiertes Layout ist zusätzlich deren Darstellung in maschinell verarbeitbarer Form notwendig. Um den Nutzern von KWebS diese Metadaten zur Verfügung zu stellen, und für die Integration des dienstbasierten Layouts in KIELER (siehe Abschnitt 7.1.1), wurde dessen Meta Layout im Rahmen dieser Arbeit erweitert. Wir befassen uns in diesem Kapitel mit der Implementierung des Dienstes. Die für die Nutzung des dienstbasierten Layouts in KIELER vorgenommen Erweiterungen lernen wir an dieser Stelle ebenfalls kennen. Die Klassenstruktur der Erweiterungen zeigt Abbildung 5.5.

LayoutDataService ist weiterhin als Singleton-Klasse implementiert, unterstützt nun aber verschiedene Betriebsarten, von denen die benötigte mit der Methode setMode gewählt werden kann. Jede Betriebsart wird durch eine Klasse implementiert, die von LayoutDataService indirekt erbt, und sich bei ihr zum Zeitpunkt der Instanziierung mit der Methode addService registriert. Die möglichen Betriebsarten werden von LayoutDataService durch String-Konstanten vordefiniert:

- ECLIPSEDATASERVICE für den Betrieb von KIELER mit lokalem Layout,
- REMOTEDATASERVICE für den Betrieb von KIELER mit dienstbasiertem Layout und

[1] http://www.graphviz.org/content/dot-language
[2] http://graphml.graphdrawing.org/specification.html
[3] https://eldorado.tu-dortmund.de/bitstream/2003/23280/1/Endbericht_PG478.pdf
[4] http://math.nist.gov/MatrixMarket/reports/MMformat.ps

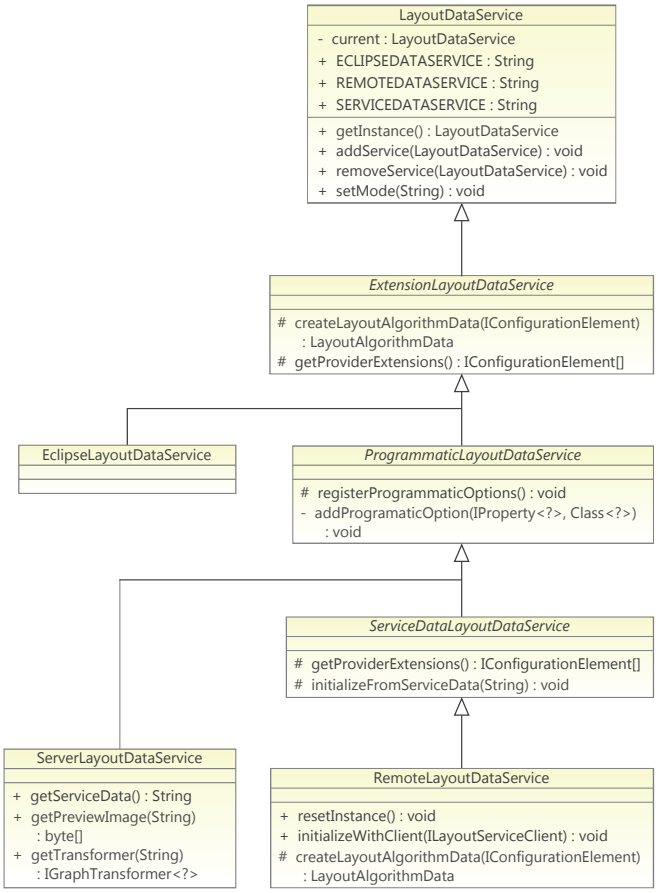


Abbildung 5.5.: Erweiterungen des Meta Layouts

- SERVICEDATASERVICE für den Betrieb im Server.

Die verschiedenen Betriebsarten werden durch die folgenden Klassen implementiert, von denen LayoutDataService bei Aufruf der Methode getInstance die der momentanen Betriebsart entsprechende Instanz zur Verfügung stellt:

- EclipseLayoutDataService implementiert den Betriebsmodus von KIELER mit lokal ausgeführtem Layout, definiert durch die Konstante ECLIPSEDATASERVICE. Sie erbt von der abstrakten Klasse ExtensionLayoutDataService und initialisiert die Metadaten aus der lokalen Installation von KIELER.

- RemoteLayoutDataService implementiert den Betriebsmodus von KIELER mit dienstseitig ausgeführtem Layout, definiert durch die Konstante REMOTEDATASERVICE. Sie erbt von der abstrakten Klasse ServiceDataLayoutDataService. Die Initialisierung der Metadaten erfolgt durch Aufruf der Methode initializeFromServiceData. Das dazu notwendige, in XMI serialisierte Modell (siehe

Abschnitt 5.4) wird von KWebS bei Aufruf der Operation getServiceData geliefert.

- ServerLayoutDataService implementiert den serverseitig verwendeten Betriebsmodus, definiert durch die Konstante SERVICEDATASERVICE. Sie erbt von der abstrakten Klasse ProgrammaticLayoutDataService und registriert die für die Verarbeitung der Nutzermodelle notwendigen programmatisch definierten Optionen. Sie stellt den Dienstvarianten die Metadaten zentralisiert zur Verfügung. Dazu erzeugt sie zur Startzeit anhand der Extension Registry das für die Darstellung der Metadaten verwendete Modell (siehe Abschnitt 5.4). Des Weiteren hält sie die Vorschaubilder vor. Auch die Instanzen der benötigten Transformer zur Übersetzung der Nutzermodelle erzeugt sie zur Startzeit anhand der Extension Registry und stellt diese den Dienstvarianten zentralisiert zur Verfügung.

Die in Abbildung 5.5 abgebildeten abstrakten, von LayoutDataService erbenden Klassen dienen der Ausgliederung von mehrfach benötigter Funktionalität:

- ExtensionLayoutDataService stellt die Funktionalität zur Initialisierung der Metadaten aus der Extension Registry zur Verfügung. EclipseLayoutDataService erbt von dieser Klasse, um die Metadaten aus der lokalen Installation von KIELER zu initialisieren.

- ProgrammaticLayoutDataService stellt die in KIELER ausschließlich programmatisch definierten Layout-Optionen zur Verfügung. Diese Optionen sind nicht durch Extensions definiert und sind daher nicht in den Metadaten enthalten. Sie werden in KIELER rein programmatisch verwendet. Ein Modell kann jedoch mit diesen Optionen annotiert sein. Um KGraph-Modelle sowohl auf Seiten des Dienstes als auch in KIELER korrekt serialisieren und deserialisieren zu können, müssen diese Optionen bekannt sein. Zu diesem Zweck können sie mit der Methode registerProgrammaticOptions von erbenden Klassen in den Metadaten registriert werden.

- Die Klasse ServiceDataLayoutDataService erbt von ProgrammaticLayoutDataService, und dadurch indirekt auch von ExtensionLayoutDataService. Sie verwendet registerProgrammaticOptions für die Registrierung der programmatischen Layout-Optionen. Die von ihr zur Verfügung gestellte Methode initializeFromServiceData kann von erbenden Klassen verwendet werden, um die Metadaten anhand eines Dienstes zu initialisieren. Die Metadaten werden in Form eines in XMI serialisierten Modells (siehe Abschnitt 5.4) erwartet, welches bei Aufruf der Methode übergeben wird. Die Initialisierung erfolgt durch Überschreiben der Methode getProviderExtensions. Das Ergebnis dieser Methode ist ein Array von Objekten des Typs IConfigurationElement, welches von ExtensionLayoutDataService für die Initialisierung der Metadaten verwendet wird. In Eclipse repräsentieren diese Objekte die zu einem Extension-Point definierten

Extensions. Das für die Darstellung der Metadaten verwendete Modell ist der Struktur der beteiligten Extension-Points nachempfunden, die für die Initialisierung notwendigen Objekte des Typs IConfigurationElement können daher von ServiceDataLayoutDataService leicht aus dem Modell abgeleitet werden.

5.3. Implementierung der Dienstvarianten

KWebS bietet zwei Varianten des dienstbasierten Layouts: als SOAP Web Service und als jETI-Tool. In diesem Abschnitt betrachten wir deren Implementierung.

5.3.1. Implementierung des SOAP-Dienstes

In Abschnitt 4.3.1 entschieden wir uns für die Referenzimplementierung von JAX-WS als Framework für die Implementierung und Bereitstellung des SOAP-Dienstes. Sie verwendet die in den Java Specification Requests (JSRs) 181 [4] und 224 [24] definierten Java-Annotationen für die Anreicherung von Java-Klassen und Schnittstellen mit Web Service-bezogenen Informationen. Wie wir in Abschnitt 3.5 sahen, bestehen zwei Möglichkeiten, einen Web Service zu erstellen: Code-First und Contract-First. Die aus technischer Sicht zu bevorzugende Methode ist Contract-First. Sie bietet, neben präziseren sprachlichen Mitteln zur Formulierung des Vertrages, einen höheren Grad an Abstraktion von der zugrundeliegenden Plattform. Die Syntax und Semantik des Vertrages wird des Weiteren nicht von dem verwendeten Web Service-Framework beeinflusst, wodurch die Neutralität des Vertrages gewährleistet wird.

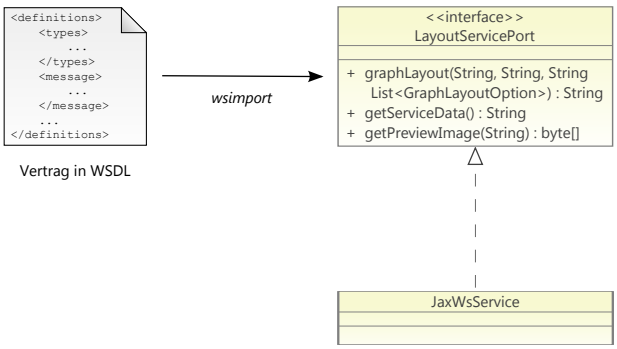


Abbildung 5.6.: Die Implementierung des SOAP-Dienstes

Für die Implementierung des SOAP-Dienstes verwenden wir daher die Contract-First-Methode. Einen Ausschnitt des in WSDL formulierten Vertrages zeigt Anhang C.1. Der vollständige Vertrag ist aufgrund seiner Größe zu umfangreich. Er kann bei Bedarf unter der Adresse `http://rtsys.informatik.uni-kiel.de:9442/layout?wsdl` eingesehen werden. Abbildung 5.6 zeigt schematisch den Verlauf der Implementierung.

Sie beginnt mit der Formulierung des Vertrages in WSDL und definiert damit die Schnittstelle des Dienstes auf plattformunabhängige Art. Die Code-Generierung geschieht mit dem von JAX-WS dafür bereitgestellten Werkzeug wsimport. Es erzeugt das SEI des Dienstes: die mit Annotationen versehene Java-Schnittstelle LayoutServicePort. Die generierte Schnittstelle ist im Anhang C.2 abgebildet, an dieser Stelle verwenden wir als Beispiel die auf die Operation graphLayout eingeschränkte Version, welche Listing 5.1 zeigt. Import-Deklarationen und Kommentare sind aus Gründen der Übersichtlichkeit nicht abgebildet.

```
@WebService(
  name = "LayoutServicePort",
  targetNamespace = "http://rtsys.informatik.uni-kiel.de/layout"
)
@XmlSeeAlso({ObjectFactory.class})
public interface LayoutServicePort {
  @WebMethod
  @WebResult(targetNamespace = "")
  @RequestWrapper(
    localName = "graphLayout",
    targetNamespace = "http://rtsys.informatik.uni-kiel.de/layout",
    className = "de.cau.cs.kieler.kwebs.jaxws.GraphLayout"
  )
  @ResponseWrapper(
    localName = "graphLayoutResponse",
    targetNamespace = "http://rtsys.informatik.uni-kiel.de/layout",
    className = "de.cau.cs.kieler.kwebs.jaxws.GraphLayoutResponse"
  )
  @Action(
    input = "http://rtsys.informatik.uni-kiel.de/"
          + "layout/LayoutServicePort/graphLayoutRequest",
    output = "http://rtsys.informatik.uni-kiel.de/"
           + "layout/LayoutServicePort/graphLayoutResponse",
    fault = {
      @FaultAction(
        className = ServiceFault_Exception.class,
        value = "http://rtsys.informatik.uni-kiel.de/"
              + "layout/LayoutServicePort/ServiceFault"
      )
    }
  )
  public String graphLayout(
    @WebParam(name = "serializedGraph", targetNamespace = "")
    String serializedGraph,
    @WebParam(name = "informat", targetNamespace = "")
    String informat,
    @WebParam(name = "outformat", targetNamespace = "")
    String outformat,
    @WebParam(name = "options", targetNamespace = "")
```

```
    List<GraphLayoutOption> options
  ) throws ServiceFault_Exception;
}
```

Listing 5.1: Mit wsimport generierte Schnittstelle des SOAP-Dienstes

Die Annotation @WebService definiert die generierte Java-Schnittstelle als SEI eines Web Service. Die Attribute der Annotation, name und targetNamespace, bilden den Bezug zu der durch das portType-Element des Vertrages abstrakt definierten Schnittstelle. Die graphLayout-Operation des Vertrages wurde in eine Java-Methode gleichen Namens übersetzt. Sie ist mit mehreren Annotationen versehen:

- @WebMethod: Deklariert die Methode als Operation des Web Service. Der Bezeichner der Methode ist aus dem Bezeichner der Operation im Vertrag abgeleitet. Diese Annotation ist nicht zwingend notwendig, da die graphLayout-Methode als öffentlich deklariert ist, und daher standardmäßig als Operation des Web Service gilt.
- @WebResult: Wird ein Web Service Code-First entwickelt, dann wird der Vertrag des Web Service zur Laufzeit generiert. Mit dieser Annotation kann die Abbildung des Rückgabewertes der Methode auf ein part-Element des Vertrages beeinflusst werden. In der abgebildeten Schnittstelle hat die Annotation keine Bedeutung.
- @RequestWrapper, @ResponseWrapper: Die für den Aufruf einer Operation notwendigen Argumente und das Ergebnis der Operation werden auf der Transportschicht in XML-Nachrichten dargestellt, im Allgemeinen durch SOAP-Nachrichten. Diese Annotationen beeinflussen das Unmarshaling der Nachrichten aus der XML-Darstellung in Instanzen von Java-Klassen, bzw. das Marshaling in umgekehrter Richtung. Die Attribute localName und targetNamespace der Annotationen definieren das XML-Element und dessen Namensraum, das Attribut className den qualifizierten Namen der Java-Klasse, die für das Marshalling und Unmarshalling des Elements verwendet wird. Die im abgebildeten Beispiel deklarierten Klassen werden während der Code-Generierung erzeugt.
- @Action: Diese Annotation verwendet den Standard WS-Addressing[5] und dient prinzipiell der Entkopplung von SOAP-Endpunkten und -Nachrichten von der Transportschicht. Sie wird während der Code-Generierung standardmäßig erzeugt. Da der SOAP-Dienst WS-Addressing nicht verwendet, ist die @Action-Annotation im Kontext dieser Arbeit nicht von Bedeutung.

Die Implementierung des SOAP-Dienstes realisiert diese Schnittstelle. Sie dient als Adapter zur in 5.2 beschriebenen Layout-Infrastruktur des Servers und delegiert Aufrufe von Operationen an die entsprechenden Methoden der Klassen AbstractService und ServerLayoutDataService.

[5] http://www.w3.org/2002/ws/addr/

5.3.2. Implementierung des jETI-Dienstes

Im Vergleich zum Aufwand der Implementierung des SOAP-Dienstes ist die des jETI-Dienstes eher als leichtgewichtig einzustufen. Dies ist darin begründet, dass jETI nicht als Web Service-Plattform konzipiert ist, sondern der Integration von Legacy-Anwendungen in das jABC-Framework dient. Aufgrund der Java-Natur beider Produkte ist Plattformunabhängigkeit nicht von Bedeutung. Dadurch ist insbesondere die Formulierung eines Vertrages nicht erforderlich, was z.B. bei SOAP Web Services aufgrund der verschiedenen beteiligten XML-Notationen einen hohen Aufwand erfordert.

Für die Bereitstellung des jETI-Dienstes benötigen wir zwei Komponenten:

- *Tool-Klasse*: Die Implementierung des Dienstes besteht aus einer normalen Java-Klasse, es bestehen keine Anforderungen an Realisierung einer Schnittstelle, Vererbung von einer Klasse oder Annotation.
- *Tool-Descriptor*: Der Toolserver von jETI verwendet einen Tool-Descriptor, der die Abbildung der von ihm bereitgestellten Tools auf die öffentlichen Methoden einer Tool-Klasse definiert.

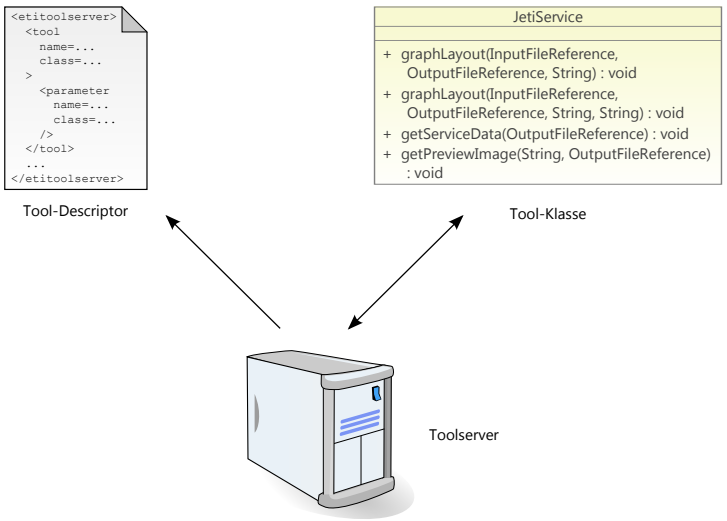


Abbildung 5.7.: Die Implementierung des jETI-Dienstes

Die Klasse JetiService dient als Tool-Klasse des jETI-Dienstes, der verwendete Tool-Descriptor ist in Anhang C.3 abgebildet. Die Implementierung der Tool-Klasse weicht von der in Abschnitt 4.2.3 definierten Schnittstelle ab, wie Abbildung 5.7 zeigt. Dies ist in der Architektur der jETI-Plattform begründet. Legacy-Anwendungen arbeiten im Allgemeinen auf Dateien. Da jETI für die Anbindung von Legacy-Anwendungen konzipiert ist, werden die von einem Tool zu bearbeitenden Daten

in Form *virtueller Dateien* auf den Toolserver geladen, und das Ergebnis der Toolausführung ebenfalls in Form virtueller Dateien bereitgestellt. Als Beispiel betrachten wir die in Listing 5.2 gezeigte graphLayout-Methode der Tool-Klasse, der zugehörige Tool-Descriptor ist in Listing 5.3 abgebildet.

```
public void graphLayout(InputFileReference inRef, OutputFileReference outRef,
        String format, String serializedOptions);
```

Listing 5.2: Die graphLayout-Methode des jETI-Dienstes

```
<etitoolserver>

  <tool name="graphLayout" active="true" method="graphLayout"
        class="de.cau.cs.kieler.kwebs.server.service.JetiService">
    <parameter name="INPUT_GRAPH" required="true"
               class="de.unido.ls5.eti.toolserver.InputFileReference"/>
    <parameter name="OUTPUT_GRAPH" required="true"
               class="de.unido.ls5.eti.toolserver.OutputFileReference"/>
    <parameter name="INPUT_FORMAT" required="true"
               class="java.lang.String"/>
    <parameter name="INPUT_OPTIONS" required="false"
               class="java.lang.String"/>
  </tool>

</etitoolserver>
```

Listing 5.3: Der Tool-Descriptor des graphLayout-Tools

Das tool-Element des Descriptors bildet den Aufruf des graphLayout-Tools auf die zuvor dargestellte Methode der Tool-Klasse ab. Die Instanz der Tool-Klasse wird nach Bedarf durch den Toolserver erzeugt. Innerhalb des tool-Elements definieren parameter-Elemente vier formale Parameter des Tools: INPUT_GRAPH, OUTPUT_GRAPH, INPUT_FORMAT und INPUT_OPTIONS. Jeder Parameter wird durch die Angabe einer qualifizierten Java-Klasse getypt.

Das Modell, für welches das Layout berechnet werden soll, wird durch den Nutzer in Form einer virtuellen Datei auf dem Toolserver abgelegt. Die Nutzung der Tools geschieht sitzungsorientiert, und der Toolserver stellt für jede Sitzung dazu ein virtuelles Dateisystem bereit. Der Nutzer des Tools bestimmt den Namen der Eingabedatei durch den Parameter INPUT_GRAPH und den Namen der Ausgabedatei durch den Parameter OUTPUT_GRAPH. Die Namen werden vom Toolserver um den serverseitigen Pfad des zur Sitzung gehörenden Dateisystems erweitert und bei Aufruf der graphLayout-Methode auf deren Argumente inRef und outRef abgebildet. Deren Typen, InputFileReference und OutputFileReference, stellt jETI für den Zugriff auf virtuelle Dateien bereit. Die Parameter INPUT_FORMAT und INPUT_OPTIONS, beide vom Typ String, definieren das Format des zu berechnenden Modells und mögliche Layout-Optionen.

Die formalen Parameter eines Tools können als *erforderlich* (required=„true“) oder *optional* (required=„false“) deklariert werden. Darin liegt der Grund, warum die Tool-Klasse zwei Methoden für das graphLayout-Tool implementiert. Der Toolserver wählt die für ein Tool zu verwendende Methode nicht anhand der durch den Tool-Descriptor formal definierten Signatur, sondern nach deren Belegung durch die vom Nutzer übergebenen Argumente. INPUT_OPTIONS ist der einzig optionale Parameter des graphLayout-Tools, und daher nicht zwingend in einer Anfrage enthalten. In diesem Fall verwendet der Toolserver die in Listing 5.4 gezeigte Methode.

```
public void graphLayout(InputFileReference inRef,
        OutputFileReference outRef, String format);
```

Listing 5.4: Alternative graphLayout-Methode des jETI-Dienstes

Eine weitere Abweichung von der in Abschnitt 4.2.3 definierten Schnittstelle ist die Darstellung der Layout-Optionen. Im Gegensatz zur definierten Schnittstelle verwendet das Tool einen Parameter von Typ String. Der Grund dafür ist das Marshaling und Unmarshaling von jETI. Ein Parameter eines Tools kann durch eine beliebige, qualifizierte Java-Klasse getypt sein. jETI verwendet für die Kommunikation zwischen Nutzer und Toolserver ein proprietäres Anwendungsprotokoll, das Streaming Eti Performance Protocol (SEPP). Wie Listing 5.5 zeigt, müssen die bei Aufruf eines Tools übergebenen Argumente vom Typ String sein.

```
public void exec(String tool, Map<String, String> parameters) throws EtiLocalException,
        EtiRemoteException;
```

Listing 5.5: exec-Methode des SEPP-Protokolls

Das Argument tool definiert den Namen des auszuführenden Tools, parameters die an das Tool übergebenen Argumente. Für die Verwendung eines Tools muss der Nutzer die beteiligten Argumente eigenständig serialisieren. Für Basis-Typen von Java, wie z.B. Boolean oder Integer, kann er dazu deren toString-Methode verwenden. Für selbst definierte Typen muss er die Serialisierung eigenständig implementieren, das Marshaling obliegt daher dem Nutzer. Der Toolserver unterstützt das Unmarshaling insofern, als er für die Instanziierung des zugehörigen Typen einen Konstruktor voraussetzt, der als Einziges ein Argument vom Typ String besitzt. Für Java-Basistypen ist dies im Allgemeinen der Fall, für eigenständig definierte Typen muss der Nutzer das Unmarshaling ebenfalls selbst implementieren.

Die in Abschnitt 4.2.3 definierte Schnittstelle beschreibt die Layout-Optionen als Liste von Objekten des Typs GraphLayoutOption, einem eigenständig definierten Typ. Des Weiteren besteht nicht die Möglichkeit, Parameter eines Tools als Liste eines Typen zu definieren. In der Definition eines Tools können Parameter zwar als Array eines Typen deklariert werden. In der Praxis erwies sich die Verwendung eines Array-Parameters mit eigenständig definierten Typen jedoch als nicht verwendbar. Daher ist das Marshaling und Unmarshaling der Layout-Optionen eigenständig implementiert und ihre serielle Notation wird als String-Parameter an den Dienst übermittelt.

5.4. Darstellung der Metadaten

Wie wir in Abschnitt 4.2.3 sahen, stellt KWebS seinen Nutzern die Meta Daten in einer auf XML basierenden Notation zur Verfügung und vereinfacht damit die Integration in Software-Systeme.

Für die Darstellung der Metadaten verwendet KWebS ein Ecore-Metamodell. Auf diese Weise vereinfacht KWebS die Integration des dienstbasierten Layouts in Software-Systeme, die auf Grundlage der Java-Plattform umgesetzt sind. Die Metadaten basieren auf einem strukturierten Modell und sind programmatisch einfach zu handhaben. Die XML-Notation entsteht durch Serialisierung des Metadaten-Modells nach XMI. Abbildung 5.8 zeigt das für die Notation der Metadaten verwendete Ecore-Metamodell ServiceData.

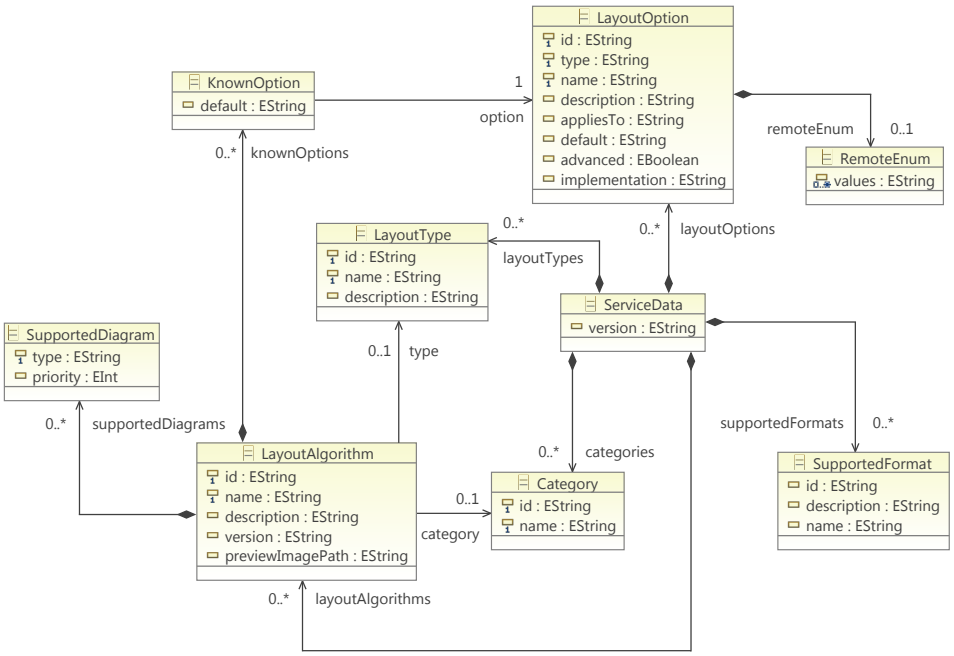


Abbildung 5.8.: Das für die Darstellung der Metadaten verwendete ServiceData-Metamodell

Die Struktur des Metamodells ist von der Struktur des Extension-Point layoutProviders abgeleitet. Er wird in KIELER von layoutbezogenen Plug-Ins für die Integration in das Meta Layout verwendet. KWebS erweitert diese Struktur um dienstspezifische Informationen.

Das Hauptelement des Modells, ServiceData, enthält alle layout- und dienstspezifischen Modellelemente. Sein Attribut version dient der Angabe der Version des Dienstes.

Die Elemente LayoutAlgorithm, LayoutOption, LayoutType und Category und deren Attribute dienen der Spezifikation von Algorithmen, deren Optionen, ihrer Klassifi-

kation nach dem Typ des berechneten Layouts (z.B. hierarchisch oder zirkulär) und ihrer Kategorisierung nach dem Anbieter (z.B. KIELER oder Graphviz).

Das Element RemoteEnum erweitert die durch den Extension-Point layoutProviders definierte Struktur. Es bildet die möglichen Belegungen einer Layout-Option ab, die eine Aufzählung darstellt. Die möglichen Belegungen werden in KIELER nicht von der zugehörigen Extension deklariert, sondern durch die Java-Enumeration, die die Option implementiert. Diese Erweiterung ist daher notwendig, da ein Nutzer die möglichen Belegungen kennen muss.

Das Element SupportedFormat repräsentiert die von KWebS unterstützten Formate. Dessen Attribut id deklariert den eindeutigen Identifikator eines Formats. Ein Nutzer übergibt ihn als informat-Parameter bei Aufruf der Operation graphLayout, um das Format seines Modells zu definieren. Er kann ihn auch für den outformat-Parameter verwenden, um ein alternatives Ausgabeformat zu wählen. Die Attribute name und description bieten den lesbaren Namen des Formats und dessen textuelle Beschreibung.

Der für das Herunterladen eines Vorschaubilds benötigte Verweis ist in dem Element LayoutAlgorithm in Form des Attributs previewImagePath hinterlegt.

6. Deployment

Das Management eines umfangreichen Softwareprojekts umfasst viele Aspekte, von denen die effiziente Integration von Entwicklungszweigen und das automatisierte Bauen und Testen des Produkts ein wesentlicher Bestandteil ist. KIELER verwendet *Subversion*[1] für das Code-Management und *Hudson*[2] [12] als Buildsystem, um diesem Aspekt des Managements gerecht zu werden.

KWebS wird von der Arbeitsgruppe auf zwei Arten angeboten. Zunächst steht der Server als Download zur Verfügung. Dies ermöglicht die Nutzung des dienstbasierten Layouts in internen Netzwerken und damit dessen Verwendung in Umgebungen, in denen die Vertraulichkeit der Modelle von Bedeutung ist. Des Weiteren stellt KWebS seine Dienste zur konkreten Nutzung über das Internet zur Verfügung. Die Grundlage für beide Arten der Veröffentlichung ist das Bauen des Servers.

In diesem Kapitel lernen wir die Integration von KWebS in den Buildprozess des KIELER Projekts kennen. Dazu stellt Abschnitt 6.1 zunächst Hudson vor. Im darauf folgenden Abschnitt 6.2 betrachten wir den Buildprozess von KWebS, welcher neben dem Bauen des Servers auch dessen Installation auf der virtuellen Maschine durchführt, die zur Veröffentlichung des dienstbasierten Layouts über das Internet dient.

6.1. Hudson

In einem Softwareprojekt, das von vielen beteiligten Programmierern entwickelt wird, ist die *Integration* der verschiedenen, lokal vorliegenden Entwicklungszweige in den Hauptzweig ein wichtiger Schritt. Verschiedene Programmierer arbeiten zur gleichen Zeit an verschiedenen Komponenten des Projekts. Dies führt möglicherweise zu Inkompatibilitäten zwischen lokalen Entwicklungszweigen. Sie werden im Allgemeinen erst entdeckt, nachdem die lokalen Zweige in den Hauptzweig integriert wurden und ein Build erfolgt ist. Abhängig von der Entwicklungsdauer eines lokalen Zweiges sind auftretende Fehler nur schwer zu beheben. Hudson ist ein Buildsystem, welches den Entwicklungs- und Integrationsprozess komplexer Softwareprojekte durch *Continuous Integration* (CI) unterstützt.

CI ist eine der Hauptpraktiken des *Extreme Programming*. Sie fordert die häufige (mindestens einmal pro Tag) Integration lokaler Zweige in den Hauptzweig. Das Bauen des Hauptzweiges erfolgt automatisch nach jedem Integrationsschritt. Auftretende Fehler werden unverzüglich den verantwortlichen Programmierern mitgeteilt, z.B. per

[1] `http://subversion.apache.org`
[2] `http://jenkins-ci.org`

Email. Deren Erinnerung an vorgenommene Änderungen sind aufgrund der kurzen Integrationsphase sehr detailliert. Die von ihnen benötigte Zeit für das Beheben der Fehler ist daher im Allgemeinen sehr kurz. CI ermöglicht somit das frühzeitige Erkennen und Beheben von Fehlern im Hauptzweig, was letzten Endes der Qualität des Produkts zugutekommt.

6.1.1. Arbeiten mit Hudson

Das Bauen komplexer Softwaresysteme besteht im Allgemeinen aus mehreren einzelnen, aufeinander aufbauenden Arbeitsschritten, von denen der grundlegende Schritt die Kompilation selbst ist. Auf der erfolgreichen Kompilation aufbauend werden meist *Unit-Tests* durchgeführt, die die ordnungsgemäße Funktion des Produkts sicherstellen, oder es werden *Code-Metriken* gegen ihre Anforderungen geprüft. Die Veröffentlichung des Produkts zum Download stellt einen weiteren möglichen Arbeitsschritt dar.

Ein Buildsystem muss diese vielseitigen Aspekte des Bauens berücksichtigen und geeignete Mittel zur strukturierten Formulierung des Buildprozesses bereitstellen. Hudson verwendet zu diesem Zweck *Jobs*. Ein Job kann grundlegende Einheiten des Buildprozesses definieren, wie z.B. die Kompilation. Höherwertige Jobs können auf ihnen aufbauend weitere Aspekte des Buildprozesses umsetzen. Zu diesem Zweck definiert ein Job einen *Auslöser*. Dieser bewirkt die Ausführung des Jobs, sobald definierte Ereignisse stattgefunden haben, wie z.B. das erfolgreiche Bauen der Jobs, von denen er abhängig ist. Ein Job kann des Weiteren *Batch-Tasks* festlegen. Ein Batch-Task führt, wie es der Name bereits vermuten lässt, ein Shell-Skript aus. Er kann für die manuelle Ausführung, oder für den automatischen Start nach der erfolgreichen Ausführung des Jobs konfiguriert werden.

Die meisten komplexen Softwareprojekte verwenden heutzutage ein Source Code Management (SCM). Für die Ausführung des Buildprozesses muss der Hauptzweig des Projekts zur Verfügung stehen; ein Buildsystem muss somit in der Lage sein, mit dem SCM zu interagieren. Hudson unterstützt standardmäßig das Concurrent Version System (CVS) und Subversion. Für viele weitere SCMs, wie z.B. Git oder Perforce, kann die Unterstützung durch *Plug-Ins* nachgerüstet werden. Wird ein Job ausgelöst, so lädt Hudson den für die Ausführung des Jobs notwendigen Quellcode aus dem SCM. Der Zugriff auf die benötigten Quellen wird innerhalb des Jobs konfiguriert.

Ein Softwareprojekt ist im Regelfall eng mit einer Buildumgebung verbunden, die auf den Arbeitsplätzen der Programmierer für das Bauen ihrer lokalen Zweige eingesetzt wird. Populäre Beispiele solcher Umgebungen sind z.B. Apache Ant[3] oder Apache Maven[4]. Das Bauen des Hauptzweiges sollte unter gleichen Bedingungen erfolgen, damit die Nachvollziehbarkeit des Buildprozesses gegeben ist. Aus diesem Grund bietet Hudson nativ Unterstützung für auf Maven basierende Projekte. Hudson kann durch Plug-Ins erweitert werden, sodass auch andere Buildumgebungen wie

[3] `http://ant.apache.org`
[4] `http://maven.apache.org`

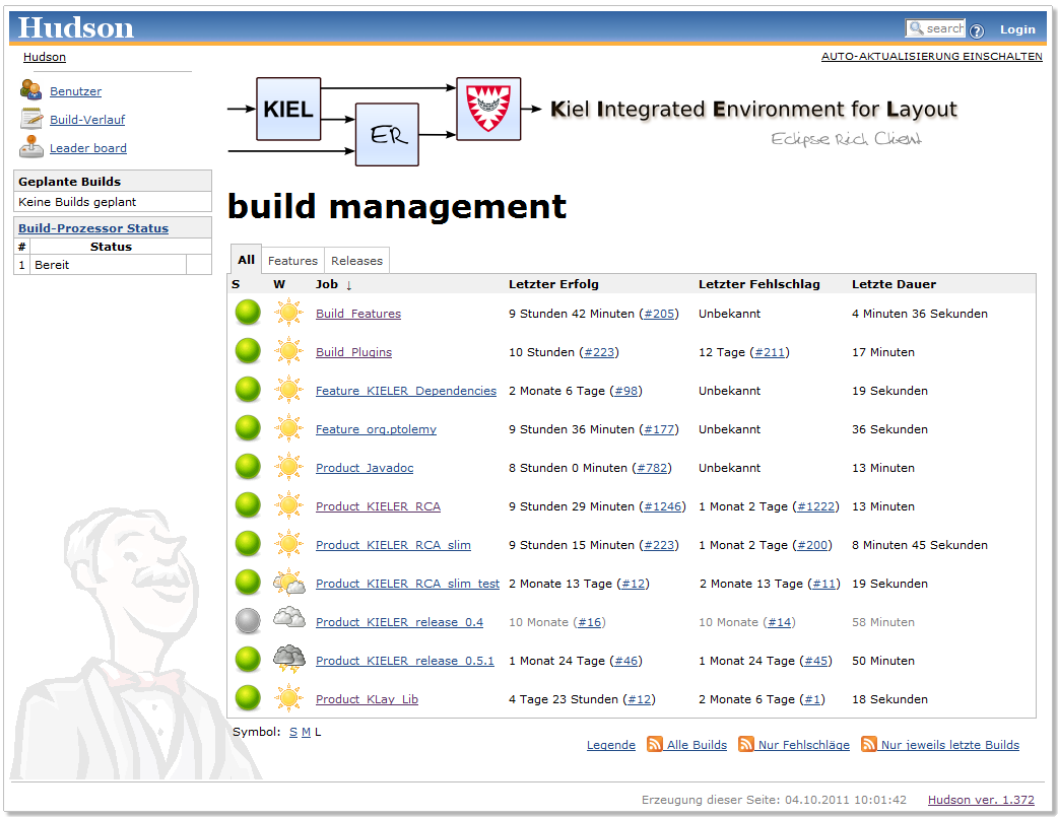


Abbildung 6.1.: Übersicht der in Hudson definierten Jobs

z.B. *MSBuild* verwendet werden können.

Eines der Kernfeatures von Hudson ist die Visualisierung von Metainformationen bezüglich der konfigurierten Jobs. Sie helfen bei der Beurteilung ihrer Stabilität. Ist ein fehlgeschlagenes Bauen ein Einzelfall, oder geschieht dies häufig? Zu diesem Zweck hält Hudson zu jedem Job einen Verlauf vor. Abbildung 6.1 zeigt die Übersicht der in Hudson konfigurierten Jobs. Auf der linken Seite eines jeden Jobs ist in den meisten Fällen eine Sonne zu sehen. Dies bedeutet, dass der Job sich in einem stabilen Zustand befindet und das Bauen (wenn überhaupt) nur selten fehlschlägt. Hudson verwendet für die Visualisierung verschiedene Wettersymbole, welche es erlauben, die Stabilität der Jobs schnell und intuitiv einzuschätzen. Drei der abgebildeten Jobs haben ein anderes Symbol. Ein paar Wolken in der Sonne symbolisieren das gelegentliche Fehlschlagen des Bauens, wolkiges Wetter ein regelmäßiges. Wenn Gewitter herrscht bedeutet das, dass das Bauen nur selten ordnungsgemäß abgeschlossen wird. Neben der Stabilität hält Hudson viele weitere Informationen bezüglich der Jobs vor, wie z.B. deren Konsolenausgaben zur Fehleranalyse oder auch die durch Tests festgestellten Metriken. Dies hilft bei der Beurteilung der Qualität der Jobs, und damit letzten Endes der Qualität des Produkts, auf Grundlage eines längeren

Zeitraums.

6.2. Buildprozess und Installation

KWebS verwendet Hudson für zwei getrennte Aufgaben. Die Erste baut den Server und installiert ihn anschließend auf der für die Veröffentlichung verwendeten virtuellen Maschine. Die Zweite betrifft die Nutzung des dienstbasierten Layouts in KIELER. Die KIELER Modellierungsumgebung stellte während der Entwicklung naturgemäß den ersten Anwendungsfall des dienstbasierten Layouts dar (siehe Abschnitt 7.1.1). Die Infrastruktur für die Nutzung des dienstbasierten Layouts ist in KIELER direkt integriert. KWebS bietet das Layout auf zwei Arten an: als SOAP- und als jETI-Dienst. Für beide Plattformen wurden Clients implementiert. Die Wahl einer Plattform obliegt dem Nutzer, und so werden die Clients als *Feature* angeboten, die nicht fest in KIELER integriert sind. Ein Nutzer kann sie nachträglich über die Update-Site von KIELER[5] installieren:

- das KIELER JAX-WS Client Feature für die Nutzung des SOAP-Dienstes und
- das KIELER jETI Client Feature für die Nutzung des jETI-Dienstes.

Für das Bauen des Servers und dessen Installation auf der virtuellen Maschine und für das Bauen der Features verwendet KWebS zwei Jobs:

- den Job Product_KWebS_Server und
- den Job Build_KWebS_Client_Features.

KIELER und auch KWebS sind Eclipse-RCAs, und daher verwenden die Jobs für das Bauen die PDE von Eclipse. Sie bietet zu diesem Zweck vorkonfigurierte Build-Skripte für Ant. Die Jobs von KWebS basieren daher auf Ant als Buildumgebung und nutzen die Skripte der PDE.

6.2.1. Der Job Build_KWebS_Client_Features

Der weniger aufwendige Job von KWebS ist Build_KWebS_Client_Features. Er erzeugt die für die Nutzung des dienstbasierten Layouts in KIELER notwendigen Features. Sie basieren auf Plug-Ins, und so ist es notwendig, dass das Bauen dieser Plug-Ins bereits abgeschlossen ist. Aus diesem Grund wird der Job Build_KWebS_Client_Features durch den Job Build_Plugins ausgelöst. Abbildung 6.2 zeigt die Konfiguration des Auslösers.

Build_Plugins ist ein grundlegender Job in KIELER, er erzeugt die KIELER Plug-Ins aus ihrem Quellcode. Nachdem er ausgelöst wurde, verwendet Build_KWebS_Client_Features das Ant-Target *build.multiple.features* des PDE-Skriptes. Abbildung 6.3 zeigt die Konfiguration des Builds.

[5] http://rtsys.informatik.uni-kiel.de/~kieler/updatesite/nightly/

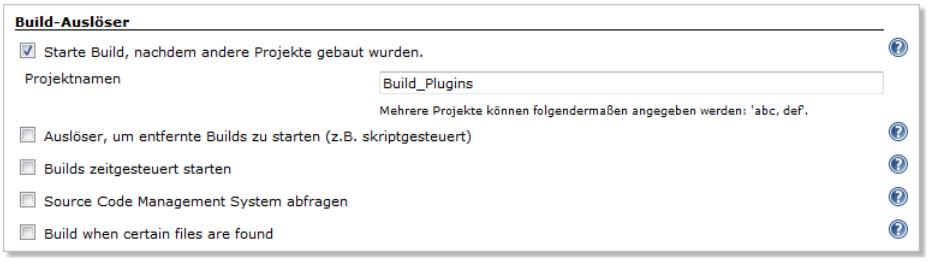


Abbildung 6.2.: Auslöser des Jobs für das Bauen der Client-Features

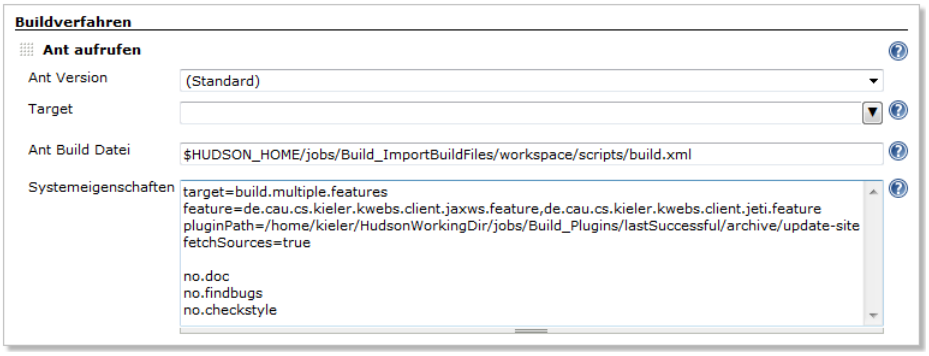


Abbildung 6.3.: Bauen der Client-Features

Die zu bauenden Features, de.cau.cs.kieler.kwebs.client.jaxws.feature und de.cau.cs.kieler.kwebs.client.jeti.feature, hat Build_KWebS_Client_Features zuvor aus dem SCM in seinen Arbeitsbereich geladen.

6.2.2. Der Job Product_KWebS_Server

Der komplexere Job von KWebS ist Product_KWebS_Server. Er dient dem Bauen des Servers und dessen Installation auf der virtuellen Maschine. Der Server von KWebS ist als Eclipse-RCA implementiert und definiert ein *Produkt*. Ein Produkt dient in Eclipse dazu, eine RCA mit Metainformationen bezüglich deren Ausführung auf verschiedenen Plattformen (Windows, Solaris, Linux, Mac) anzureichern. Auch Lizenzinformationen und Anpassungen der Benutzeroberfläche, das *Branding*, können mit einem Produkt definiert werden. Entscheidend ist, dass ein Produkt die Anwendungsklasse der RCA definiert und die von der RCA benötigten Plug-Ins bzw. Features festlegt (ein Produkt kann auf Plug-Ins oder Features basieren, jedoch nicht auf beiden gleichzeitig).

Der Server verwendet ein auf Features basierendes Produkt. Die Produktdefinition und die notwendigen Features lädt Build_KWebS_Client_Features zu Beginn seiner Ausführung aus dem SCM. Er wird ausgelöst, sobald der Job Build_Features abgeschlossen ist (siehe Abbildung 6.4). Dieser ist, genau wie Build_Plugins, ein

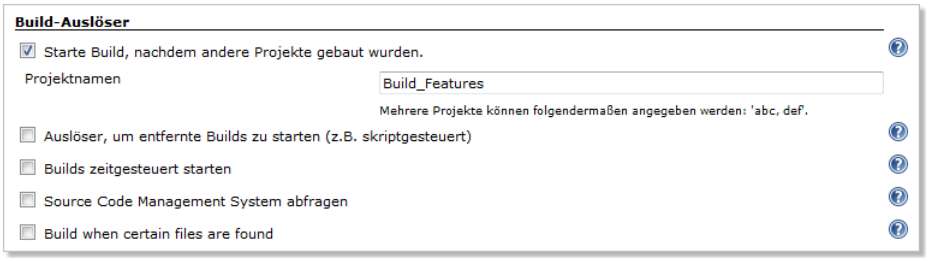


Abbildung 6.4.: Auslöser des Jobs für das Bauen des Servers

grundlegender Job von KIELER. Er wird nach dem erfolgreichen Bauen der Plug-Ins ausgelöst und erzeugt die KIELER Features.

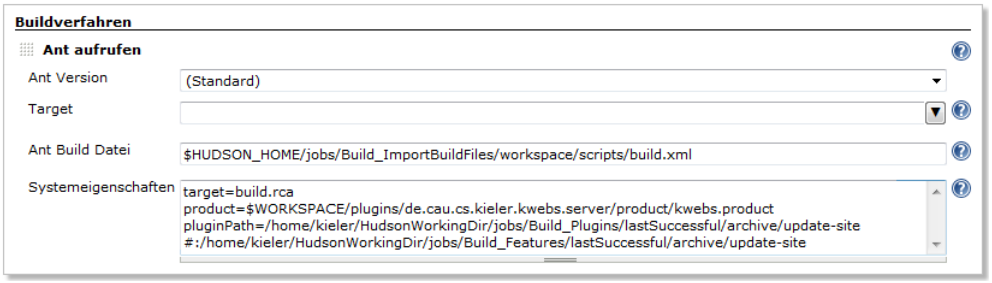


Abbildung 6.5.: Bauen des Servers

Product_KWebS_Server verwendet dasselbe PDE-Skript wie Build_KWebS_Client_Features, jedoch nun mit dem Ant-Target *build.rca*. Dieses Target dient dem Erstellen einer RCA, basierend auf einem Produkt. Das Ergebnis des Bauens ist eine Ansammlung von Archiven, die im Arbeitsbereich des Jobs abgelegt sind. Sie enthalten jeweils eine vollständige Version des Servers für eine der unterstützten Plattformen. Alle benötigten Artefakte wie z.B. die notwendigen Plug-Ins, Konfigurationsdateien für die Eclipse-Plattform und die für die Ausführung auf der Zielplattform benötigte binäre Datei sind in dem Archiv enthalten.

Nach dem erfolgreichen Bauen des Servers installiert Product_KWebS_Server ihn auf der virtuellen Maschine, die der Veröffentlichung von KWebS dient. Hierfür verwendet der Job einen Batch-Task, der die folgenden Aufgaben durchführt:

- *Vorbereiten der virtuellen Maschine*: Stellt sicher, dass die für die Installation des Servers benötigten Verzeichnisstrukturen vorhanden und leer sind.

- *Kopieren der aktuellen Version*: Kopiert das Archiv, welches die zuletzt gebaute Version des Servers enthält, aus dem Arbeitsbereich des Jobs auf die virtuelle Maschine.

- *Stoppen des Servers*: Beendet den gegenwärtig auf der virtuellen Maschine ausgeführten Server. Dazu führt der Batch-Task das Shell-Skript kwebs_stop.sh auf der virtuellen Maschine aus, welches Abschnitt C.5 des Anhangs zeigt.

- *Sicherung der Betreibereinstellungen*: Der Betreiber kann den Server anpassen. Dies betrifft dessen Konfiguration (siehe Abschnitt B.1), aber z.B. auch die für den Betrieb mit HTTPS benötigten Java-Keystores. Zur Sicherung dieser Einstellungen führt der Batch-Task das Shell-Skript kwebs_backup.sh aus, welches Abschnitt C.6 des Anhangs zeigt.

- *Deployment der neuen Version*: Die bisher installierte Version des Servers wird gelöscht. Dadurch werden auch die Einstellungen des Betreibers entfernt. Anschließend wird das zuvor heruntergeladene Archiv, welches den aktuellen Build des Servers enthält, entpackt.

- *Wiederherstellung der Betreibereinstellungen*: Die zuvor gesicherten, betreiberseitig vorgenommenen Einstellungen werden in die aktuelle Version des Servers übernommen. Hierzu führt der Batch-Task das Shell-Skript kwebs_restore.sh aus, welches Abschnitt C.7 des Anhangs zeigt.

- *Starten des Servers*: Startet die neu installierte Version des Servers. Der Batch-Task verwendet dazu das Shell-Skript kwebs_start.sh, welches Abschnitt C.8 des Anhangs zeigt.

7. Evaluation

In diesem Kapitel befassen wir uns mit der Bewertung von KWebS. Wir nehmen sie anhand der in Abschnitt 4.1 gestellten Anforderungen vor. Eine Anforderung an das dienstbasierte Layout ist dessen Einfachheit. Die verschiedenen Aspekte dieser Anforderung beurteilen wir in Abschnitt 7.1. Insbesondere betrachten wir für die Bewertung des Nutzungsaufwands zwei Anwendungsfälle. Der Erste ist naturgemäß KIELER selbst, da es der Entwicklung von KWebS diente. Den zweiten Fall stellt das jABC-Framework der Technischen Universität Dortmund dar. Inwieweit die Anforderung der Wartbarkeit von KWebS eingehalten wird, diskutieren wir in Abschnitt 7.2. Abschließend bewerten wir in Abschnitt 7.3 die Effizienz des dienstbasierten Layouts mit KWebS.

7.1. Einfachheit

Das dienstbasierte Layout sollte nicht wesentlich zur Komplexität des KIELER Projekts beitragen. Ein Maß für die Komplexität ist die Anzahl der Plug-Ins und Features, die KWebS dem KIELER Projekt hinzufügt. Aus dieser Sicht ist das dienstbasierte Layout mit insgesamt zwölf eigenen und sechs Fremdanbieter-Komponenten (für die Unterstützung von jETI) als relativ komplex zu beurteilen. Dies ist in der Tatsache begründet, dass das Konzept des dienstbasierten Layouts zu Beginn der Arbeit in KIELER technisch nicht vorgesehen war, und die notwendige Infrastruktur dafür erst geschaffen werden musste.

Die hinzugefügten Abhängigkeiten zu externen Komponenten bieten ein weiteres Maß zur Beurteilung der Komplexität. KWebS bietet das dienstbasierte Layout in zwei Varianten an. Für den SOAP-Dienst verwendet KWebS JAX-WS. Seit Version 6 ist JAX-WS in die Java-Plattform integriert. Der SOAP-Dienst ruft daher vom Grundsatz her keine Abhängigkeiten zu externen Komponenten hervor. Die in Java integrierte Version von JAX-WS ist jedoch recht alt, daher verwendet KWebS deren Referenzimplementierung mittels des *endorsed*-Mechanismus von Java. Die in diesem Kontext hauptsächlich erzeugte Komplexität rührt von jETI her, nicht nur durch die sechs hinzugefügten Fremd-Komponenten, sondern durch die neu eingeführte, proprietäre Architektur.

Ein weiterer Aspekt ist die Komplexität der angebotenen Schnittstelle. Diese ist mit den drei Operationen graphLayout, getServiceData und getPreviewImage einfach gestaltet und zu nutzen. Die Unterstützung verschiedener, weit verbreiteter Formate wie z.B. DOT oder GraphML erleichtert die Nutzung von KWebS zusätzlich. Derzeit nicht unterstützte Formate können KWebS auf einfache Weise durch Implementierung eines entsprechenden Transformers hinzugefügt werden.

KWebS verwendet ein Ecore-Metamodell für die Darstellung seiner Metadaten. Das vereinfacht die Integration des dienstbasierten Layouts in Softwaresysteme weiter, da die Metadaten in einer strukturierten, programmatisch einfach zu handhabenden Form vorliegen. Das bedingt, dass auf der Nutzerplattform Ecore-Modelle programmatisch verarbeitet werden können. In Java ist diese Möglichkeit mit EMF gegeben, auch für C++ steht mit *EMF4CPP*[1] ein Framework zur Verfügung. Auf Plattformen, die über keine Unterstützung von EMF verfügen, kann die Integration der Metadaten durch die Verwendung eines XML-Parsers geschehen, da die serielle Notation der Metadaten auf XMI basiert.

Vom Standpunkt der Implementierung betrachtet entspricht KWebS der Forderung der Einfachheit soweit wie möglich. Die Bereitstellung eines dienstbasierten Layouts stellt eine Facette von KIELER dar, die zu Beginn dieser Arbeit konzeptionell nicht vorgesehen war. Die durch KWebS hinzugefügte Komplexität ist daher notwendig und in ihrem Umfang vertretbar.

Die Nutzung des dienstbasierten Layouts ist anhand der bereitgestellten Schnittstelle ebenfalls als einfach zu bewerten. Zur Beurteilung des Aufwands der Integration in Software-Systeme betrachten wir im Folgenden die bereits genannten Anwendungsfälle.

7.1.1. Anwendungsfall: KIELER

Die Integration des dienstbasierten Layouts in KIELER besteht aus zwei Teilen. Da KIELER bereits über ein umfangreiches automatisches Layout verfügt und das von KWebS angebotene Layout dem von KIELER entspricht, ist das dienstbasierte Layout als optional anzusehen. Ein Nutzer muss daher zwischen lokalem und dienstbasiertem Layout wählen können. Er muss zu jedem Zeitpunkt auf einfache, nicht aufdringliche Weise über die momentan verwendete Art des Layouts informiert werden. Das dienstbasierte Layout kann des Weiteren von verschiedenen Betreibern angeboten werden. Ein typisches Szenario ist z.B. der Betrieb von KWebS in einem firmeninternen Netzwerk. Ein Nutzer muss daher den zu verwendenden Anbieter konfigurieren können.

Ein Teil der Integration besteht somit darin, dem Nutzer über die GUI von KIELER geeignete Mittel für den Umgang mit dem dienstbasierten Layout zur Verfügung zu stellen. Der andere Teil liegt in der Integration des dienstbasierten Layouts auf technischer Ebene. Dieser beinhaltet die Ausführung des Layouts unter Verwendung des Dienstes. Des Weiteren müssen seine Metadaten auf transparente Weise in KIELER integriert werden.

Im Folgenden gehen wir zunächst auf die Erweiterungen der GUI ein, bevor wir darauf die technischen Aspekte der Integration betrachten.

[1] `http://catedrasaes.inf.um.es/trac/wiki/EMF4CPP`

Integration in die GUI

KWebS signalisiert die Art des momentan verwendeten Layouts durch ein Symbol in der Statusleiste von KIELER. Abbildung 7.1(a) zeigt das Symbol, das einem Nutzer während des lokalen Layouts und Abbildung 7.1(b) das Symbol, das ihm bei dienstbasiertem Layout angezeigt wird.

(a) Lokales Layout (b) Dienstbasiertes Layout

Abbildung 7.1.: Die Symbole der Statusleiste

Der Nutzer kann auf einfache Weise zwischen lokalem und dienstbasiertem Layout wechseln. Zu diesem Zweck ist dem Symbol in der Statusleiste ein Kontextmenü hinterlegt. Es öffnet sich nach einem Klick mit der rechten Maustaste auf das Symbol. Abbildung 7.2 zeigt das Kontextmenü. Es bietet einem Nutzer drei Optionen. Er kann das lokale Layout von KIELER aktivieren, oder das dienstbasierte mit KWebS. Die dritte Option öffnet einen Dialog, der statistische Daten über die zuletzt dienstbasiert ausgeführten Layouts zeigt.

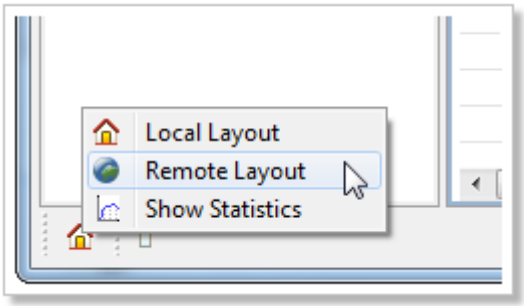


Abbildung 7.2.: Das Kontextmenü des Statussymbols

Für die Konfiguration von Anbietern und die Auswahl des gegenwärtig verwendeten Anbieters bietet KWebS eine Preference Page. Ein Nutzer kann sie mit einem Doppelklick auf das Statussymbol, oder auch über den entsprechenden Eintrag im Hauptmenü von KIELER aufrufen. Wie Abbildung 7.3 zeigt, kann er dort zunächst ebenfalls zwischen lokalem und dienstbasiertem Layout wählen. Die Preference Page verfügt des Weiteren über eine Liste der gegenwärtig hinterlegten Anbieter. Die beiden Anbieter KIELER Demo Layout Service (HTTP) und KIELER Demo Layout Service

(jETI) sind vordefiniert und können nicht geändert oder gelöscht werden. Sie sind daher mit einem Schloss-Symbol markiert. Der gegenwärtig für das dienstbasierte Layout gewählte Anbieter, der *aktive* Anbieter, ist mit einem grünen „Ok"-Symbol markiert, wie im Beispiel der Anbieter RTSYS HTTP.

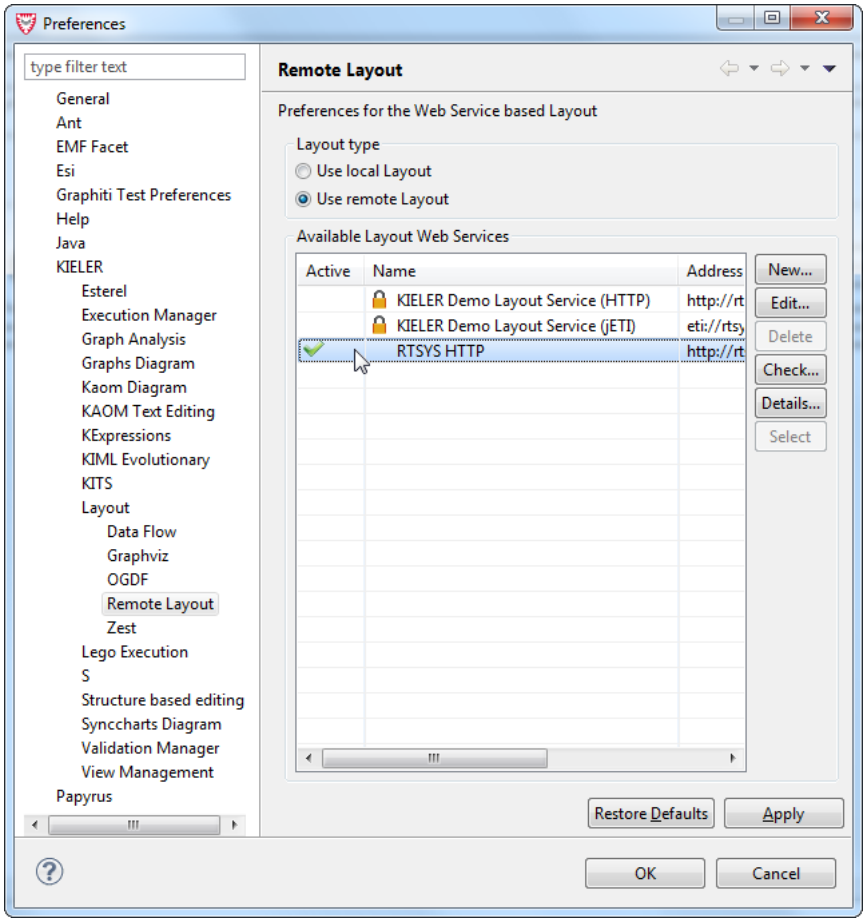


Abbildung 7.3.: Die Preference Page von KWebS

Ein Nutzer kann durch Klicken des New...-Buttons einen neuen Anbieter zur Liste hinzufügen, oder den Edit...-Button verwenden, um die Konfiguration eines vorhandenen Anbieters anzupassen. Durch Klicken des Check...-Buttons kann ein Nutzer überprüfen, ob der in der Liste selektierte Anbieter gegenwärtig verfügbar ist. Der Details...-Button öffnet einen Dialog, der einen Nutzer detailliert über den vom selektierten Anbieter veröffentlichten Dienst informiert. Er bietet Informationen bezüglich der nutzbaren Algorithmen und verwendbaren Formate. Das Löschen eines Anbieters ist ebenso möglich, solange es sich nicht um einen vordefinierten oder aktiven Anbieter handelt. Dies ist in Abbildung 7.3 der Fall, daher ist der Delete-Button deaktiviert. Der Select-Button setzt den gegenwärtig in der Liste markierten

Anbieter als aktiv. In Abbildung 7.3 ist er deaktiviert, da bereits der aktive Anbieter selektiert ist.

Für das Anlegen oder Bearbeiten eines Anbieters öffnet sich ein Dialog, wie er von Abbildung 7.4 dargestellt wird. Ein Nutzer konfiguriert den Namen des Anbieters unter Configuration Name und die Adresse des Dienstes unter Address. Ebenso kann er den *Trust Store* und das notwendige Passwort für einen durch HTTPS gesicherten Dienst konfigurieren. Diese Optionen sind im gezeigten Beispiel nicht aktiviert. Sie werden automatisch freigeschaltet, sobald die Adresse des Dienstes als Protokoll HTTPS definiert. Der Check Connection-Button bietet einem Nutzer die Möglichkeit zu testen, ob der konfigurierte Dienst verfügbar ist.

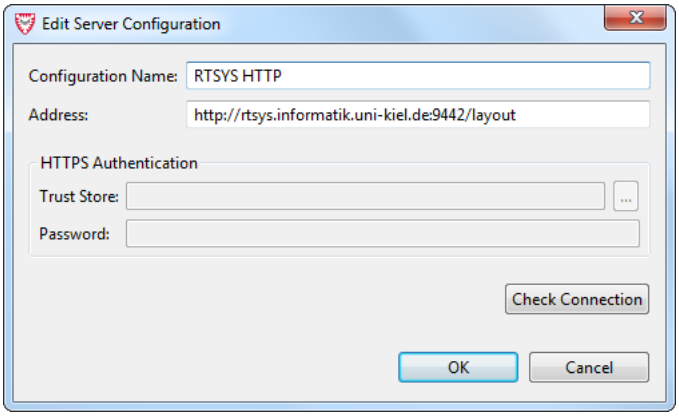


Abbildung 7.4.: Bearbeiten eines Anbieters

Aus Sicht des Anwenders integriert sich das dienstbasierte Layout auf einfach zu nutzende Weise in KIELER. Die Konfiguration des für ein Diagramm verwendeten Layouts findet, wie im Falle des lokalen Layouts, über die gewohnten Elemente der GUI statt. Die von KWebS vorgenommenen Erweiterungen, das Statussymbol und die Preference Page zur Konfiguration, bieten dezent gestaltete, einfach zu nutzende Bedienelemente zur Administration des dienstbasierten Layouts.

Integration des Layouts

Das Meta Layout bietet mit der Klasse DiagramLayoutEngine einen zentralen Zugang zum automatischen Layout von KIELER. Sie verwendet die Klasse RecursiveGraphLayoutEngine für die konkrete Berechnung des Layouts eines KGraph-Modells. Für die Integration des dienstbasierten Layouts führt KWebS eine zusätzliche Klasse ein: die RemoteGraphLayoutEngine. Beide Klassen realisieren die Schnittstelle IGraphLayoutEngine, wie Abbildung 7.5 darstellt.

Das dienstbasierte Layout ist in KIELER optional, ein Nutzer kann zwischen lokalem und dienstbasiertem Layout wechseln. Die Einstellungen bezüglich des Layouts hinterlegt KWebS in Preferences. Für jeden Layout-Vorgang prüft DiagramLayoutEngine anhand der zugehörigen Preference, welche Art des Layouts ein Nutzer gewählt

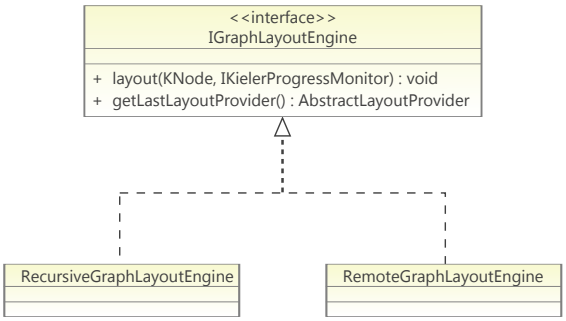


Abbildung 7.5.: Die vom Meta Layout für lokales und dienstbasiertes Layout verwendeten Klassen

hat und verwendet die entsprechende Realisierung der Schnittstelle IGraphLayoutEngine. Den Verlauf eines dienstbasierten Layouts zeigt Abbildung 7.6.

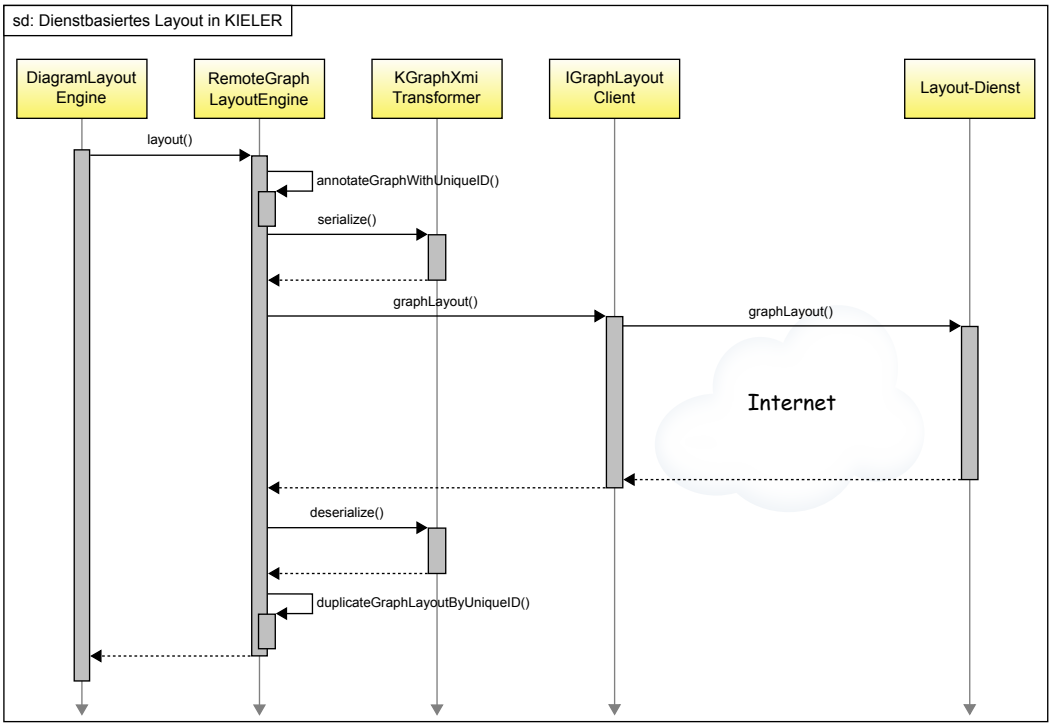


Abbildung 7.6.: Integration des dienstbasierten Layouts in KIELER

Das Meta Layout verwendet das KGraph-Metamodell für die Berechnung des Layouts von Diagrammen auf Grundlage von deren Struktur. Für die Anwendung des berechneten Layouts auf das Diagramm ist es notwendig, die Korrelation zwischen

Elementen des Diagramms und Elementen des Strukturmodells vorzuhalten. Das Meta Layout verwendet hierfür die Instanzen der beteiligten Modellelemente. KWebS verwendet serielle Notationen für die Interaktion mit seinen Nutzern. Daher ist eine Korrelation der Elemente des berechneten Strukturmodells zum ursprünglichen Diagramm nicht gegeben. Zur Anwendung des berechneten Layouts annotiert RemoteGraphLayoutEngine die Elemente des Strukturmodells mit einem eindeutigen Identifikator, bevor sie das dienstbasierte Layout in Anspruch nimmt. Hierfür verwendet sie die Methode annotateGraphWithUniqueID.

Das von KWebS unterstützte Format bei Verwendung von KGraph-Modellen ist deren serielle Notation in XMI. Daher verwendet die RemoteGraphLayoutEngine die Klasse KGraphXmiTransformer, eine Realisierung der in Abschnitt 5.2.1 vorgestellten Schnittstelle IGraphTransformer, für die Abbildung des Strukturmodells nach XMI.

Anschließend ruft RemoteGraphLayoutEngine die graphLayout-Operation des Dienstes auf. Sie übergibt als Parameter das zuvor serialisierte Strukturmodell. Dessen Format kennzeichnet sie mit dem in KWebS dafür vorgesehenen, String-basierten Identifikator FORMAT_KGRAPH_XMI.

Gibt ein Nutzer kein abweichendes Ausgabeformat an, so liefert KWebS das berechnete Modell in demselben Format, das er für die Inanspruchnahme des Layouts gewählt hat. Dies ist für KIELER der Fall, und daher liegt das berechnete Strukturmodell ebenfalls in XMI vor. RemoteGraphLayoutEngine verwendet darum erneut den KGraphXmiTransformer, um eine Instanz des berechneten Modells abzuleiten. Anschließend überträgt sie anhand der zuvor erzeugten Identifikatoren das berechnete Layout in das ursprüngliche Strukturmodell. Dessen Korrelation zum Ursprungsdiagramm wurde durch das dienstbasierte Layout nicht beeinflusst.

Nach der Aktualisierung des korrelierten Strukturmodells ist der dienstbasierte Anteil des Layouts abgeschlossen. Die darauf folgenden Schritte entsprechen denen des lokalen Layouts. Das dienstbasierte Layout bettet sich somit auf transparente Art in das Meta Layout von KIELER ein.

Die Integration des dienstbasierten Layouts in KIELER ist grundsätzlich als einfach zu bewerten. Die Kernfunktionalität bezüglich des Layouts stellen wenige Klassen bereit. Zu ihnen zählt, neben den Client-Implementierungen und der Transformation der KGraph-Modelle nach XMI, vor allem die RemoteGraphLayoutEngine, welche das dienstbasierte Layout in Anspruch nimmt. Der Großteil des Aufwands betraf die Bereitstellung geeigneter GUI-Konzepte für die Administration des dienstbasierten Layouts durch den Nutzer, welche wir zu Beginn dieses Abschnitts kennengelernt haben.

Integration der Metadaten

Die Berechnung des Layouts ist kein statischer Prozess, sie kann durch einen Nutzer beeinflusst werden. Er kann aus einer Menge von Algorithmen denjenigen wählen, der in seinem Ermessen für ein gegebenes Diagramm am Besten geeignet ist. Diesen kann er auf verschiedene Weise durch Optionen beeinflussen. Die Möglichkeiten der Einflussnahme werden einem Nutzer durch die GUI von KIELER vermittelt, wobei

sie auf die Metadaten zurückgreift, die ihr das Meta Layout bereitstellt.

Ein Dienst verfügt nicht notwendigerweise über dieselben Fähigkeiten wie eine lokale Installation von KIELER. Verschiedene Anbieter können unterschiedliche Versionen des Dienstes anbieten. Die komponentenbasierte Natur von Eclipse bietet einem Anbieter des Weiteren die Möglichkeit, individuelle Implementierungen von Algorithmen anzubieten. Die Metadaten eines Dienstes müssen daher dynamisch zur Laufzeit in KIELER eingebettet werden. Zu diesem Zweck wurde das Meta Layout im Rahmen dieser Arbeit erweitert, wie es uns Abschnitt 5.2.2 vorgestellt hat.

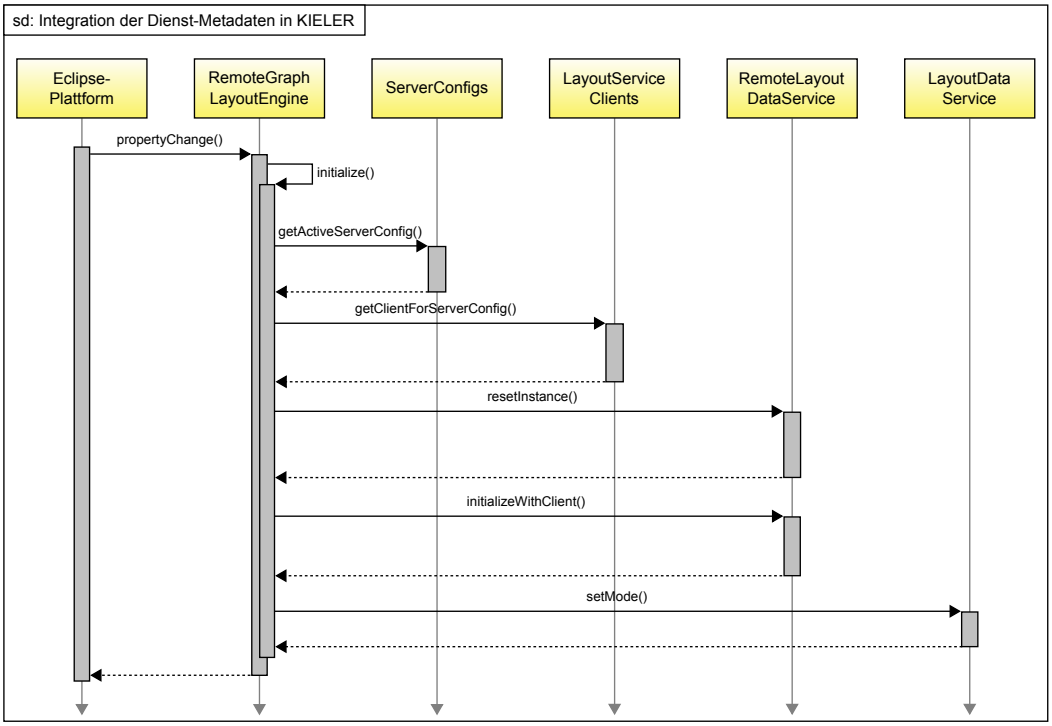


Abbildung 7.7.: Integration der Dienst-Metadaten in KIELER

Abbildung 7.7 zeigt, wie die Metadaten eines Dienstes in KIELER integriert werden. Der Vorgang beginnt, sobald ein Nutzer über die zuvor beschriebene Preference Page Änderungen an der Konfiguration des dienstbasierten Layouts vorgenommen hat. Änderungen an Preferences werden von der Eclipse-Plattform mittels des *observer pattern* publiziert. Die Klasse RemoteGraphLayoutEngine realisiert die Schnittstelle IPropertyChangeListener und registriert sich als *listener* des entsprechenden Preference Stores. Sie wird dadurch über Änderungen an Preferences informiert, die in Bezug zum dienstbasierten Layout stehen.

Die vom Nutzer konfigurierten Anbieter werden zur Laufzeit von der Singleton-Klasse ServerConfigs verwaltet. Die RemoteGraphLayoutEngine verwendet sie, um den vom Nutzer gewählten, aktiven Anbieter festzustellen. Danach überprüft sie, ob die

vom Nutzer ausgeführten Änderungen einen Wechsel des Anbieters zur Folge hatten. Dies ist nicht notwendigerweise der Fall. Er kann z.B. von lokalem auf dienstbasiertes Layout umgestellt haben. In diesem Fall bleibt der aktive Anbieter erhalten. KWebS bietet das dienstbasierte Layout in verschiedenen Varianten an: als SOAP- und als jETI-Dienst. Für jede der Varianten ist zur Nutzung eine eigenständige Client-Implementierungen notwendig. Bei einem Wechsel des Anbieters nutzt RemoteGraphLayoutEngine die Singleton-Klasse LayoutServiceClients, um eine Instanz der zum aktiven Anbieter kompatiblen Client-Implementierung zu erhalten. Mit ihrer Hilfe ermittelt die Klasse RemoteLayoutDataService anschließend die Metadaten des Dienstes und integriert sie in das Meta Layout von KIELER. Der Integrationsvorgang endet mit dem Umschalten in den dienstbasierten Betriebsmodus von KIELER.

Für die Initialisierung der layoutbezogenen Metadaten verwendet das Meta Layout die Extension Registry. Im Gegensatz dazu stellt der Dienst seine Metadaten in Form eines ServiceData-Modells zur Verfügung. Die in Abschnitt 5.2.2 beschriebenen Erweiterungen des Meta Layouts geben von ExtensionLayoutDataService erbenden Klassen die Möglichkeit, eigene Konfigurationselemente der Extension Registry für die Initialisierung zu definieren. Der RemoteLayoutDataService macht davon Gebrauch und führt eine Übersetzung des ServiceData-Modells in Konfigurationselemente der Extension Registry durch. Dieser Prozess ist leichtgewichtig, da das ServiceData-Metamodell in seiner Struktur dem vom Meta Layout definierten Extension Point layoutProviders zur Deklaration der Metadaten nachempfunden ist. Durch die Nutzung von Konfigurationselementen zur Initialisierung wird die vom Meta Layout definierte Infrastruktur zur Einbettung von Metadaten bezüglich des Layouts wiederverwendet. Die Integration der Metadaten des Dienstes geschieht damit auf transparente Art.

Um die Einfachheit der Integration zu beurteilen, betrachten wir die Erweiterungen des Meta Layouts aus Abschnitt 5.2.2. Der wesentliche Anteil dieser Erweiterungen wurde für die Integration des dienstbasierten Layouts in KIELER vorgenommen. Die Anbindung des dienstbasierten Layouts erscheint aufgrund der Menge an neu eingeführten Klassen als aufwendig. Viele der Klassen implementieren jedoch eine als leichtgewichtig zu bezeichnende Funktionalität, wie z.B. ProgrammaticLayoutDataService und RemoteLayoutDataService, oder sie entstanden durch Auslagerung bereits implementierter Funktionalität, wie z.B. ExtensionLayoutDataService und EclipseLayoutDataService. Der hauptsächliche Aufwand für die Integration lag in der Klasse ServiceDataLayoutDataService, welche die konkrete Integration der Metadaten durchführt. Aufgrund der strukturierten Natur des für die Darstellung der Metadaten verwendeten ServiceData-Metamodells war der Aufwand jedoch angemessen gering.

7.1.2. Anwendungsfall: jABC-Framework

Um den Aufwand zu beurteilen, der für die Integration und Nutzung des dienstbasierten Layouts in Software-Systeme notwendig ist, wurde im Rahmen dieser Arbeit ein externer Anwendungsfall betreut: das jABC-Framework, welches in Abbildung 7.8 zu sehen ist.

Im Hintergrund ist der Anwendungsgraph zu erkennen, für den ein Layout berechnet werden soll. Die Integration des dienstbasierten Layouts in die GUI des jABC-Frameworks verwendet einen Dialog, der eine Auswahl zwischen den lokal im jABC-Framework vorhandenen Algorithmen und denen des Dienstes ermöglicht. Er ist in Abbildung 7.8 im Vordergrund zu sehen. Im abgebildeten Fall hat der Nutzer den KIELER Layout Algorithmus gewählt. Diese Notation ist etwas irreführend, da die Wahl dieser Option zunächst nur die Möglichkeiten zur Konfiguration des dienstbasierten Layouts aktiviert. Anschließend kann der Nutzer zwischen den verschiedenen vom Dienst angebotenen Algorithmen wählen. Im dargestellten Fall hat er sich für „Planarization“ entschieden und die vom ausgewählten Algorithmus unterstützten Optionen geeignet konfiguriert.

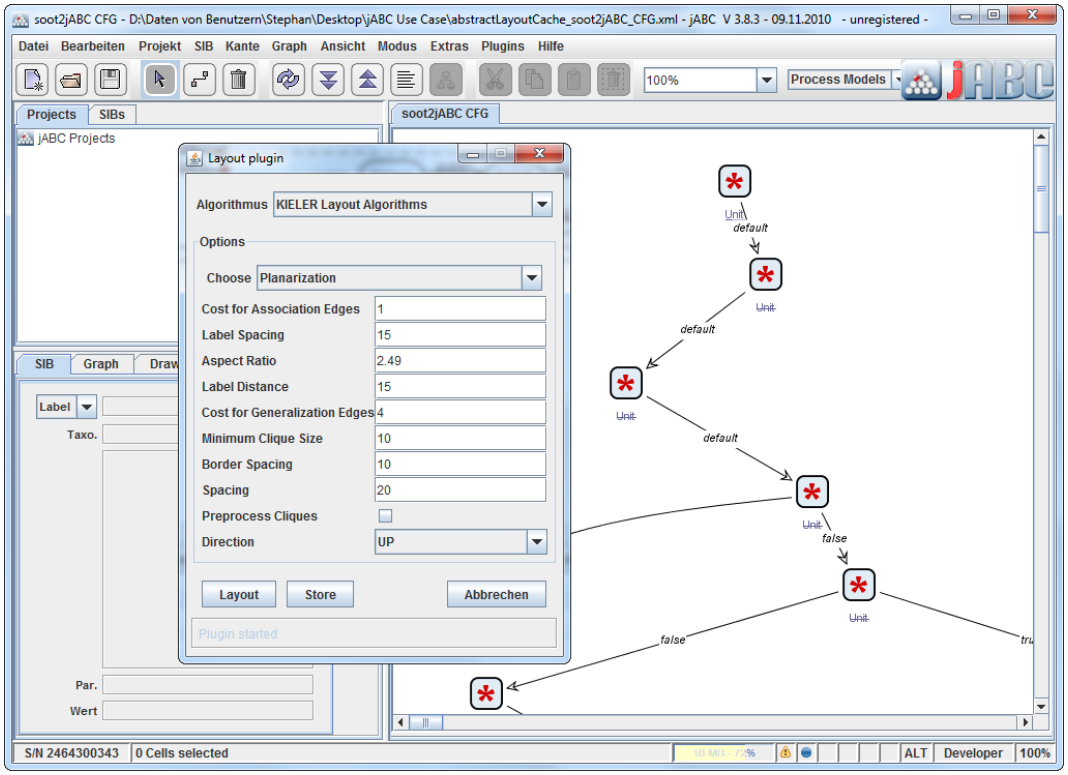


Abbildung 7.8.: Dienstbasiertes Layout im jABC

Vom technischen Standpunkt gesehen ließ sich das dienstbasierte Layout auf einfache Weise in das jABC-Framework integrieren. Die vertragsbasierte Natur des SOAP-Dienstes von KWebS ermöglichte die Generierung des notwendigen Client-Codes, was den manuellen Implementierungsaufwand zur Nutzung des dienstbasierten Layouts deutlich reduzierte. Das ServiceData-Metamodell bot eine geeignete Grundlage für die einfache Integration der Konfiguration des dienstbasierten Layouts in die bereits vorhandenen GUI-Elemente.

Das jABC-Framework verwendet eine proprietäre Notation der Anwendermodelle. Für die Nutzung des dienstbasierten Layouts war es daher notwendig, eine Transformation der jABC-Modelle in ein von KWebS unterstütztes Format zu implementieren. Zu diesem Zweck wurde das KGraph-Metamodell verwendet. Aufgrund seiner Ecore-Natur bietet es strukturierte Methoden zur Manipulation von auf ihm basierenden Modellen. Dadurch war die Implementierung der Transformation auf einfache Weise möglich.

Für die Nutzung des dienstbasierten Layouts stellt KWebS eine Anzahl von Plug-Ins zur Verfügung, die dessen Integration in Java-basierte Anwendungen auf programmatischer Ebene erleichtern. Die Integration dieser in Maven, dem vom jABC-Framework verwendeten Buildsystem, erwies sich aufgrund der sich von Eclipse unterscheidenden Art des Managements von Abhängigkeiten als schwierig.

An dieser Stelle ist anzumerken, dass während der Integration des dienstbasierten Layouts in das jABC-Framework nur das KGraph-Format zur Verfügung stand. Dadurch war die Nutzung der Plug-Ins zwingend erforderlich, da sie das KGraph-Metamodell enthalten. Die Schwierigkeiten bezüglich der Integration in das Buildsystem können nun aufgrund der Unterstützung weiterer Formate seitens KWebS überwunden werden, indem z.B. GraphML oder OGML zur Darstellung der Modelle eingesetzt wird, und damit die Plug-Ins nicht mehr erforderlich sind.

7.2. Wartbarkeit

Die Anforderung der Wartbarkeit ist für das KIELER Projekt von großer Wichtigkeit. Eine wartbare Architektur ermöglicht die gleichzeitige Nutzung des Meta Layouts sowohl von KIELER selbst als auch von KWebS.

Im Verlauf dieser Arbeit wurde das Meta Layout wie in Abschnitt 5.2.2 beschrieben erweitert, um die benötigten Betriebsbedingungen zur Verfügung zu stellen. Des Weiteren war es zu Beginn dieser Arbeit auch nicht nebenläufig. Das beeinträchtigte die Nutzbarkeit von KWebS, da Anfragen nicht gleichzeitig bearbeitet werden konnten. Im Verlauf dieser Arbeit wurde das Meta Layout daher dahingehend erweitert, dass es die nebenläufige Bearbeitung von Anfragen erlaubt. Die genannten Erweiterungen sind nicht auf KWebS beschränkt. Sie werden ebenfalls von KIELER verwendet und bedeuten daher keine Verletzung der Anforderung. Des Weiteren verwendet KWebS dieselben Implementierungen der Layout-Algorithmen, wie sie auch in KIELER Anwendung finden. Speziell für den dienstseitigen Betrieb angepasste Versionen sind nicht erforderlich. KWebS erfüllt somit die gestellte Anforderung der Wartbarkeit.

7.3. Effizienz

Das dienstbasierte Layout führt sowohl auf Seiten des Nutzers als auch auf Seiten des Dienstes eine Menge an Operationen ein, die neben der eigentlichen Berechnung

des Layouts notwendig sind. Diese unterstützenden Operationen gehen zu Lasten der Effizienz. Auch der Transfer der Modelle über das Netzwerk beeinflusst die Effizienz.

Das dienstbasierte Layout verwendet serielle Notationen für den Austausch der Modelle. Die notwendige Serialisierung und Deserialisierung erzeugt zusätzlichen Aufwand. KWebS übersetzt des Weiteren die Nutzermodelle in ein strukturell identisches KGraph-Modell, um deren Layout mithilfe des Meta Layouts zu berechnen. Anschließend annotiert KWebS das abgeleitete Strukturmodell mit den vom Nutzer angegebenen Layout-Optionen. Nach der Berechnung des Layouts übernimmt KWebS dieses in das Nutzermodell, bevor es an den Nutzer übermittelt wird. Die Ableitung des Strukturmodells, die Annotation mit Optionen und das Anwenden des Layouts auf das Nutzermodell erzeugt weiteren Aufwand. Auf Seiten des Nutzers, wie im Falle von KIELER, ist möglicherweise die Identifikation der Modellelemente erforderlich, um das berechnete Layout nach der Inanspruchnahme von KWebS anhand der Identifikatoren in das Ursprungsmodell zu übernehmen. Dieser Vorgang erzeugt ebenfalls Aufwand.

Im Folgenden beurteilen wir die Effizienz des dienstbasierten Layouts mit KWebS basierend auf der Messung der genannten Aufwände. Der erste Testfall besteht aus 145 Modellen im KGraph-Format unterschiedlicher Komplexität. Die Messung simuliert das dienstbasierte Layout in KIELER. Sie führt daher, neben der Serialisierung und Deserialisierung der Modelle, alle in KIELER benötigten unterstützenden Operationen aus, wie z.B. die Annotation eines Modells mit Identifikatoren und die Übernahme des Layouts in das Ursprungsmodell. Für das Layout der Modelle nutzt der Testfall den DOT-Algorithmus von Graphviz.

Die Messung der statistischen Daten findet im lokalen Netzwerk der Arbeitsgruppe statt. Sowohl der Client als auch der Server laufen in einer eigenständigen virtuellen Maschine, welche jeweils auf der in Tabelle 7.9 aufgeführten Hardware ausgeführt wird.

	Client	Server
Prozessor	Intel(R) Xeon(R) CPU E5540	Intel(R) Xeon(R) CPU 5140
Taktrate	2.53GHz	2.33GHz
L2-Cache	8 MB	6 MB
Hauptspeicher	2 GB	1 GB

Abbildung 7.9.: Die für den Testfall verwendete Client- und Serverhardware

Die Erhebung der Daten geschieht wie folgt: Aus dem Testfall wird ein Modell zufällig ausgewählt und das dienstbasierte Layout auf ihm angewendet. Sowohl der Client als auch der Server messen die beteiligten Aufwände und annotieren das KGraph-Modell mit den gemessenen Werten, welche der Client anschließend in einer CSV-Datei ablegt. Dieser Vorgang wiederholt sich solange, bis eine ausreichende Basis an statistischen Daten ermittelt wurde.

Dem Diagramm in Abbildung 7.10 liegt eine Basis von 132.000 Messungen zu-

grunde. Es zeigt die Gesamtzeit, die ein dienstbasiertes Layout in Abhängigkeit der Komplexität der Modelle im Mittel benötigt. Sie ist unterteilt in die Zeit, die das eigentliche Berechnen des Layouts auf Seiten des Dienstes in Anspruch nimmt (Orange) und die zusätzlich benötigte Zeit, die durch das dienstbasierte Layout entsteht (Gelb). Diese enthält die Dauer der unterstützenden Operationen sowohl auf Client- als auch auf Dienstseite und die Dauer des Netzwerktransfers. Die Komplexität ist jeweils auf der horizontalen Achse abgebildet, gemessen in der Anzahl an Elementen, die ein Modell enthält. Die Gesamtzeit ist auf der vertikalen Achse dargestellt, gemessen in Sekunden.

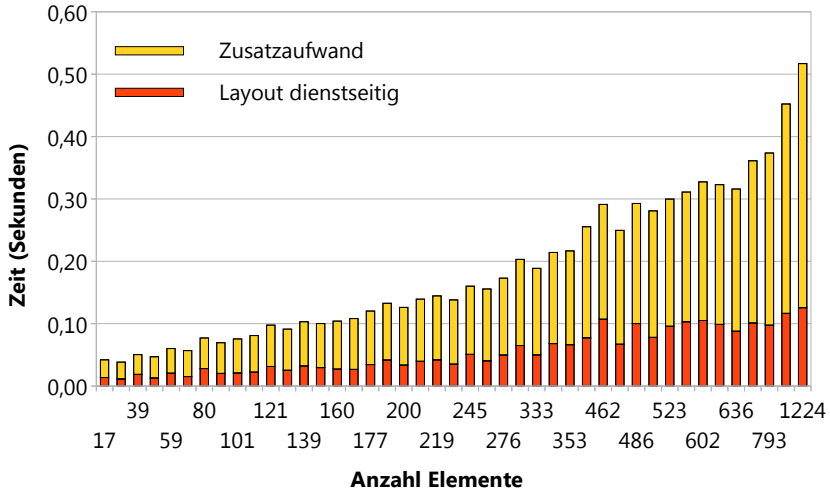


Abbildung 7.10.: Gesamtdauer des dienstbasierten Layouts im ersten Testfall in Abhängigkeit der Modellkomplexität

Das Diagramm verdeutlicht, dass die für ein dienstbasiertes Layout benötigte Zeit proportional ist zu der Zeit, die das eigentliche Berechnen des Layouts benötigt. Es ist des Weiteren ersichtlich, dass das dienstbasierte Layout selbst bei Modellgrößen von deutlich über 1.000 Elementen mit einer Dauer von einer halben Sekunde für seine Nutzer keine Einschränkung im Umgang mit ihren Modellen bedeutet.

Abbildung 7.11 zeigt den prozentualen Anteil der unterstützenden Operationen und des Netzwerktransfers an der Gesamtdauer des dienstbasierten Layouts. Wie zuvor zeigt die horizontale Achse die Komplexität der Modelle. Das Diagramm verdeutlicht, dass der Anteil mit im Durchschnitt 70 Prozent erheblich, aber konstant ist.

Wie in Abbildung 7.12(a) zu erkennen ist, entsteht mit steigender Komplexität der Modelle der Großteil des zusätzlichen Aufwands durch die unterstützenden Operationen. Der Aufwand des Netzwerktransfers nimmt mit steigender Komplexität ab, wie Abbildung 7.12(b) zeigt. Der Grund dafür ist, dass mit steigender Komplexität der Modelle eine bessere Kompressionsrate von deren serieller Notation auf der Transportschicht erreicht wird.

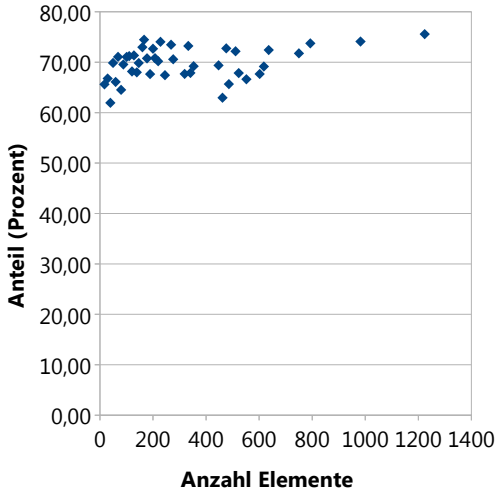
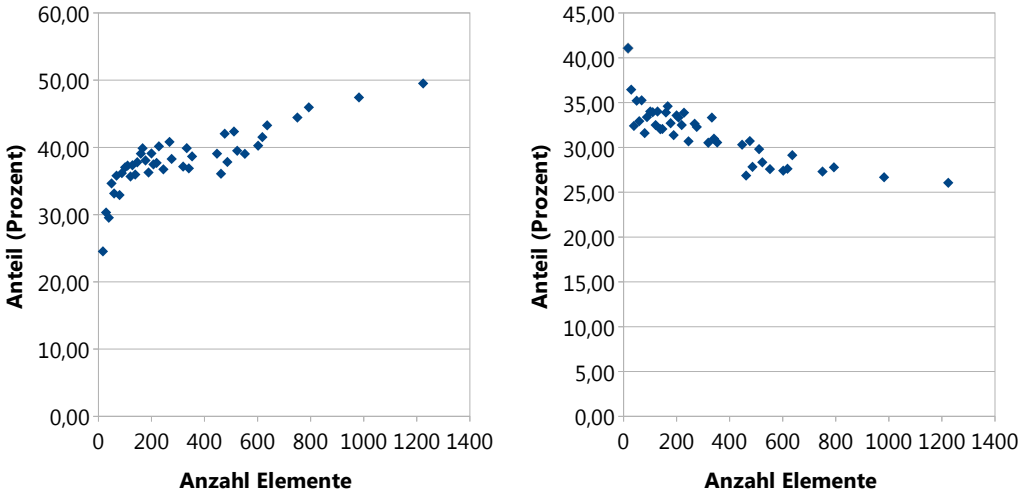


Abbildung 7.11.: Anteil der unterstützenden Operationen und des Netzwerktransfers an der Gesamtdauer in Abhängigkeit der Modellkomplexität



(a) Anteil unterstützender Operationen

(b) Anteil des Netzwerktransfers

Abbildung 7.12.: Prozentualer Anteil der beteiligten Aufwände in Abhängigkeit der Modellkomplexität

Eine Analyse der unterstützenden Operationen zeigt, dass die Serialisierung und Deserialisierung der Modelle im beschriebenen Testfall mit ca. 80 Prozent den wesentlichen Anteil deren zusätzlichen Aufwands erzeugt. Im Falle von KGraph-Modellen, wie sie für den Testfall verwendet werden, nutzen diese Operationen die durch EMF bereitgestellte Infrastruktur zur Persistierung von Ecore-Modellen. Sie ist hinrei-

chend effizient, sodass durch die Serialisierung und Deserialisierung kein unnötiger Aufwand entsteht. Die Annotation mit Identifikatoren erzeugt ca. 5 Prozent des Aufwands und ist somit nur zu einem geringen Anteil am Gesamtaufwand beteiligt. Die übrigen 15 Prozent entfallen auf die Übernahme des berechneten Layouts in das Strukturmodell. Dieser Schritt ist erforderlich, und dessen Implementierung benötigt nahezu lineare Zeit in Abhängigkeit der Komplexität eines Modells. Er erzeugt daher ebenfalls keinen unnötigen Aufwand.

Das Verhältnis zwischen den am dienstbasierten Layout beteiligten Komponenten hängt auch vom zugrundeliegenden Netzwerk ab. Die Messungen für diesen Testfall fanden im lokalen Netzwerk der Arbeitsgruppe statt. Daher spielt der Netzwerktransfer eine geringere Rolle, als wenn z.B. das Layout über das Internet stattfindet, und dadurch besonders die Übermittlung des Modells an den Dienst durch die Datenrate des Uploads eingeschränkt ist.

Die Struktur der Modelle und die damit verbundene Dauer des Layouts spielt ebenfalls eine Rolle. Abbildung 7.13 zeigt erneut die Dauer des dienstbasierten Layouts (gemessen in Sekunden) in Abhängigkeit der Modellkomplexität und Abbildung 7.14 den prozentualen Anteil der unterstützenden Operationen und des Netzwerktransfers an der Gesamtdauer. Dem dargestellten Testfall liegen dieselbe Umgebung und dieselbe Messmethode wie im zuvor beschrieben Fall zugrunde, allerdings besteht die Testbasis aus 658 zufällig generierten KGraph-Modellen und den Diagrammen liegt eine Basis von 36.000 Messungen zugrunde.

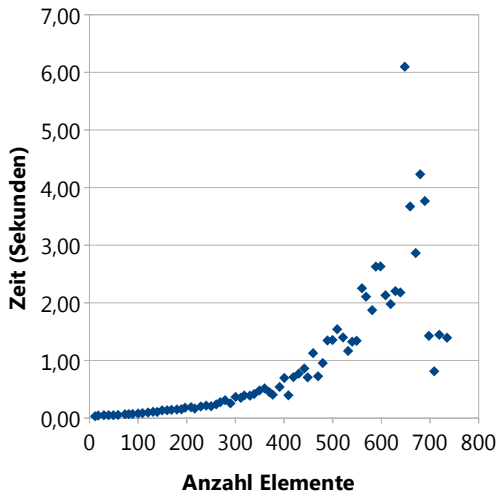


Abbildung 7.13.: Gesamtdauer des dienstbasierten Layouts im zweiten Testfall in Abhängigkeit der Modellkomplexität

Im Gegensatz zum ersten Testfall sind die Modelle dieses Falls nicht hierarchisch und der verwendete Algorithmus (DOT) zeigt ein nichtlineares Verhalten. Daher nimmt der Anteil der unterstützenden Operationen und des Netzwerktransfers mit steigender Modellkomplexität ab. Der zusätzliche Aufwand des dienstbasierten Lay-

outs spielt daher mit steigender Komplexität der Modelle eine untergeordnete Rolle.

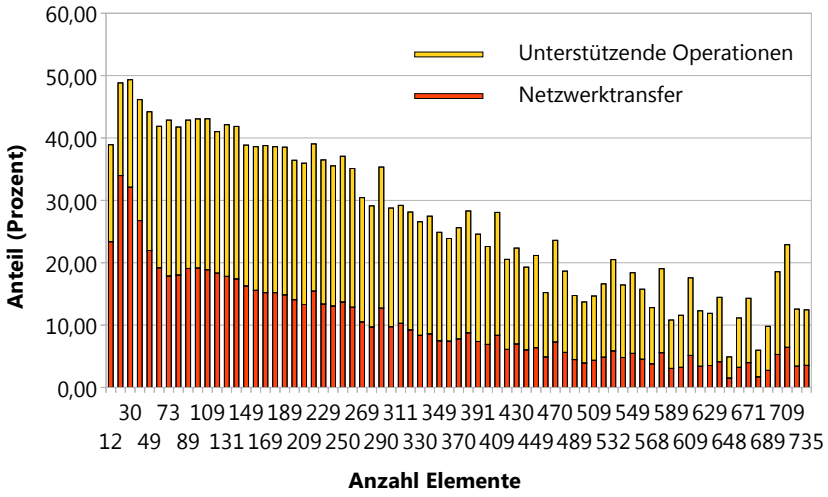


Abbildung 7.14.: Aufwände des dienstbasierten Layouts im zweiten Testfall in Abhängigkeit der Modellkomplexität

Die beschriebenen Testfälle verwenden das KGraph-Format. Daher ist das Ableiten eines Strukturmodells zur Berechnung des Layouts und das Anwenden des Layouts auf das Nutzermodell seitens des Dienstes nicht erforderlich. Deswegen wurden für die GraphML-, DOT- und Matrix Market-Formate ebenfalls Messungen vorgenommen, das OGML-Format wurde mangels einer geeigneten Testbasis nicht verwendet.

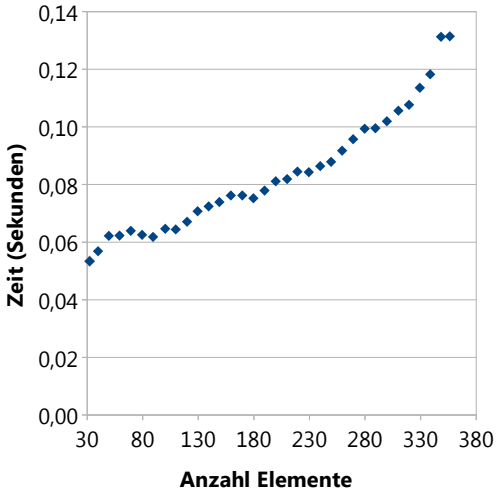


Abbildung 7.15.: Gesamtdauer des dienstbasierten Layouts im dritten Testfall in Abhängigkeit der Modellkomplexität

Als dritter Testfall dient uns das dienstbasierte Layout unter Verwendung von

GraphML. Für ihn wurde die *Rome Library* von Matthias Stegmeier als Testbasis verwendet, sie stammt aus dem Open Graph Archive [1]. Sie besteht aus 11.534 Graphen mit einer Knotenzahl von 10 bis 110 und einer Kantenzahl von 9 bis 158. Wie Abbildung 7.15 zeigt, ist das Verhalten des dienstbasierten Layouts ebenfalls nahezu linear.

7.4. Zusammenfassung

In diesem Kapitel haben wir verschiedene Aspekte des dienstbasierten Layouts betrachtet und sie anhand der gestellten Anforderungen bewertet. In Abschnitt 7.1 stellten wir fest, dass es dem KIELER Projekt in vertretbarem Umfang Komplexität hinzufügt. Anhand der vorgestellten Anwendungsfälle sahen wir, dass der Integationsaufwand zur Nutzung angemessen ausfällt. Die Anforderung der Einfachheit ist somit erfüllt. Aufgrund der Verwendung der Eclipse-RCP für den Server und von JAX-WS für den SOAP-Dienst erfüllt das dienstbasierte Layout des Weiteren die Anforderung der Wartbarkeit. Wie wir in Abschnitt 7.3 sahen, ist der zusätzliche Aufwand im Vergleich zu lokal ausgeführtem Layout von verschiedenen Faktoren abhängig, nicht zuletzt von der Struktur der Modelle selbst. Obwohl das dienstbasierte Layout eine zum lokal ausgeführten Layout proportionale Zeit in Anspruch nimmt, ist der entstehende Aufwand zum Teil erheblich. Für Modelle aus dem Bereich der grafischen Modellierung von Softwaresystemen liegt die Gesamtdauer eines Layoutvorgangs jedoch in einem Bereich, der für seine Nutzer keine Einbuße im pragmatischen Umgang mit ihren Modellen bedeutet. Das gedachte Anwendungsgebiet von KWebS sind Werkzeuge zur Modellierung von Softwaresystemen, daher erfüllt KWebS die Anforderung der Effizienz in diesem Kontext ebenfalls.

8. Abschluss

Diese Arbeit verfolgte das Ziel, das automatische Layout des KIELER Forschungsprojekts auf Grundlage eines Web Service zur Verfügung zu stellen, und es durch dessen Plattformunabhängigkeit einem breiteren Feld von Nutzern zugänglich zu machen. Sie präsentiert mit dem KIELER Web Service for Layout (KWebS) einen Dienst für das automatische Layout von Graphen, der mit wenig Aufwand in Softwaresysteme integriert werden kann.

KWebS verwendet das KIELER Meta Layout, welches neben den KIELER Algorithmen auch andere, frei verfügbare Bibliotheken für das Layout einsetzt, wie z.B. Graphviz oder auch OGDF. Nutzer können ihre Modelle in verschiedenen, standardisierten Notationen übermitteln, die erforderlichen Übersetzungen zur Berechnung des Layouts übernimmt KWebS. Seinen Nutzern stellt KWebS damit einen einfach zu verwendenden Zugang zu einer heterogenen Landschaft für das automatische Layout ihrer Modelle bereit und trägt damit seinen Teil dazu bei, die Nutzbarkeit und Akzeptanz grafischer Modellierung zu verbessern.

KWebS bietet das dienstbasierte Layout in zwei Varianten an. Der SOAP Web Service ist vom Entwurf her sitzungslos. Aufgrund der möglichen Code-Generierung reduziert er den notwendigen Implementierungsaufwand auf Seiten seiner Nutzer auf ein Minimum. Die auf jETI basierende Variante bietet aufgrund der datei- und sitzungsorientierten Natur von jETI einem Nutzer mehr Flexibilität im Umgang mit den Modellen, für die er ein Layout benötigt.

Die Schnittstelle von KWebS ist mit drei Operationen übersichtlich gestaltet. Sie bietet neben dem eigentlichen Layout seinen Nutzern notwendige layoutbezogene Metadaten. Sie beschreiben die Möglichkeiten der Einflussnahme auf den Layout-Prozess, z.B. welche Algorithmen verwendet und mit welchen Optionen sie angepasst werden können. Für die Darstellung der Metadaten verwendet KWebS ein Ecore-Metamodell, dessen strukturierte Natur Nutzern ihre programmatisch einfache Handhabung ermöglicht.

Im Rahmen dieser Arbeit wurde das dienstbasierte Layout mit KWebS in zwei Anwendungsfällen eingesetzt. Den ersten Fall stellte naturgemäß KIELER selbst dar, da es der Entwicklung von KWebS diente. Das jABC-Framework der Technischen Universität Dortmund diente anschließend als externer Anwendungsfall und bestätigte das hinter KWebS stehende Konzept.

Diese Arbeit konzentrierte sich auf die Bereitstellung eines dienstbasierten Layouts. Web Services bieten KIELER jedoch allgemein eine geeignete Grundlage dafür, weitere Bestandteile des Projekts als Dienst anzubieten, wie z.B. einen Dienst für die Analyse von Graphen. Die Zukunft wird zeigen, in welche Richtung sich das KIELER Projekt in dieser Hinsicht entwickeln wird.

A. Literaturverzeichnis

[1] Bachmaier, Christian, Franz J. Brandenburg, Philip Effinger, Carsten Gutwenger, Jyrki Katajainen, Karsten Klein, Miro Spönemann, Matthias Stegmaier und Michael Wybrow: *The Open Graph Archive: A Community-Driven Effort.* Poster at the 19th International Symposium on Graph Drawing, Technische Universiteit Eindhoven, Netherlands, September 2011.

[2] Battista, Giuseppe Di, Amadeo Giammarco, Giuseppe Santucci und Roberto Tamassia: *The Architecture of Diagram Server.* In: *Proceedings of the 1990 IEEE Workshop on Visual Languages*, Seiten 60–65, 1990.

[3] Battista, Giuseppe Di, Giuseppe Liotta und Francesco Vargiu: *Diagram Server.* Journal of Visual Languages & Computing, 6(3):275–298, 1995.

[4] BEA Systems, Inc: *Web Services Metadata for the Java™ Platform.* online, Februar 2005. `http://download.oracle.com/otndocs/jcp/web_services_metadata-2.0-mrel-oth-JSpec/`.

[5] Bridgeman, Stina, Ashim Garg und Roberto Tamassia: *A Graph Drawing and Translation Service on the WWW.* In: *Proceedings of Graph Drawing '96*, Seiten 45–52. Springer-Verlag, 1999.

[6] Charles André: *Semantics of SyncCharts.* Technischer Bericht ISRN I3S/RR–2003–24–FR, I3S Laboratory, Sophia-Antipolis, France, April 2003.

[7] Christensen, E. and Curbera, F. and Meredith, G. and Weerawarana, S.: *Web Services Description Language (WSDL) 1.1.* W3C Note, World Wide Web Consortium, March 2001. `http://www.w3.org/TR/wsdl`.

[8] Eiglsperger, Markus, Sándor P. Fekete und Gunnar W. Klau: *Orthogonal Graph Drawing.* In: Kaufmann, Michael und Dorothea Wagner (Herausgeber): *Drawing Graphs*, Band 2025 der Reihe *Lecture Notes in Computer Science*, Seiten 121–171. Springer, 1999.

[9] Fuhrmann, Hauke und Reinhard von Hanxleden: *On the Pragmatics of Model-Based Design.* In: Choppy, Christine und Oleg Sokolsky (Herausgeber): *Monterey Workshop*, Band 6028 der Reihe *Lecture Notes in Computer Science*, Seiten 116–140. Springer, 2008.

[10] Gretarsson, Brynjar, Svetlin Bostandjiev, John O'Donovan und Tobias Höllerer: *WiGis: A Framework for Scalable Web-Based Interactive Graph Visualizations.* In: Eppstein, David und Emden R. Gansner (Herausgeber): *Graph Drawing*, Band 5849 der Reihe *Lecture Notes in Computer Science*, Seiten 119–134. Springer, 2009.

[11] IBM: *Eclipse Platform Technical Overview.* online, 2006. `http://eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.pdf`.

[12] John Ferguson Smart: *Jenkins: The Definitive Guide.* online, Juli 2011. `http://wakaleo.com/download-jenkins-the-definitive-guide-pdf?view=document`.

[13] Margaria, Tiziana, Ralf Nagel und Bernhard Steffen: *jETI: A Tool for Remote Tool Integration.* In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, Band 3440 der Reihe *Lecture Notes in Computer Science*, Seiten 572–577. Springer, April 2005.

[14] Merten, Maik, Bernhard Steffen, Falk Howar und Tiziana Margaria: *Next Generation LearnLib.* In: *Seventeenth International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2011.

[15] Michi Henning: *The rise and fall of CORBA.* Communications of the ACM, 51(8):52–57, 2008.

[16] Naujokat, Stefan: *jETI: Redesign der ETI-Plattform. Eine Client/Server-Architektur für entfernte Ausführung dateibasierter Tools.* Studienarbeit, Technische Universität Dortmund, 2009.

[17] OSGi Alliance: *About the OSGi Service Platform*, Juni 2007. `http://www.osgi.org/wiki/uploads/Links/OSGiTechnicalWhitePaper.pdf`.

[18] P. Eades: *A Heuristic for Graph Drawing.* Congressus Numerantium, 42:149–160, 1984.

[19] Paul Prescod: *Roots of the REST/SOAP Debate.* In: *Proceedings of the Extreme Markup Languages 2002 Conference, Montréal, Quebec, Canada*, August 2002.

[20] Pautasso, Cesare, Olaf Zimmermann und Frank Leymann: *RESTful Web Services vs. "Big" Web Services: Making the Right Architectural Decision.* In: Huai, Jinpeng, Robin Chen, Hsiao-Wuen Hon, Yunhao Liu, Wei-Ying Ma, Andrew Tomkins und Xiaodong Zhang (Herausgeber): *WWW*, Seiten 805–814. ACM, 2008.

[21] Roy Thomas Fielding: *Architectural Styles and the Design of Network-based Software Architectures.* Doktorarbeit, University of California, Irvine,

2000. http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

[22] Stahl, Thomas, Markus Völter, Sven Efftinge und Arno Haase: *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management.* dpunkt.verlag, Heidelberg, 2. Auflage, 2007.

[23] Steffen, Bernhard, Tiziana Margaria, Ralf Nagel, Sven Jörges und Christian Kubczak: *Model-Driven Development with the jABC.* In: Bin, Eyal, Avi Ziv und Shmuel Ur (Herausgeber): *Haifa Verification Conference*, Band 4383 der Reihe *Lecture Notes in Computer Science*, Seiten 92–108. Springer, 2006.

[24] Sun Microsystems, Inc: *The Java API for XML-Based Web Services (JAX-WS) 2.0.* online, April 2006. http://download.oracle.com/otndocs/jcp/jaxws-2_0-fr-eval-oth-JSpec/.

[25] Sun Microsystems, Inc: *The Java™ Architecture for XML Binding (JAXB) 2.0.* online, April 2006. http://download.oracle.com/otndocs/jcp/jaxb-2.0-fr-eval-oth-JSpec/.

[26] W3C: *SOAP Version 1.2 Part 0: Primer (Second Edition).* online, April 2007. W3C Recommendation. http://www.w3.org/TR/soap12-part0/.

B. Anleitungen

B.1. Konfiguration des Servers

Dieser Abschnitt beschreibt die zur Konfiguration des Servers von KWebS zur Verfügung stehenden Optionen. Sie können in der Betreiberkonfiguration kwebs.user verwendet werden, oder beim Start des Servers in der Kommandozeile angegeben werden. Jede Option wird durch einen String identifiziert.

- de.cau.cs.kieler.kwebs.jaxws.publishHttp:

  Standardbelegung: true

  Mit dieser Option kann festgelegt werden, ob der SOAP-Dienst über HTTP veröffentlicht wird.

  Mögliche Werte: true, false

- de.cau.cs.kieler.kwebs.jaxws.httpAddress:

  Standardbelegung: http://localhost:8442/layout

  Mit dieser Option wird der für die Veröffentlichung des SOAP-Dienstes über HTTP verwendete Endpunkt konfiguriert. Von der definierten Adresse werden dazu der Host, der Port und der Kontext verwendet.

- de.cau.cs.kieler.kwebs.jaxws.publishHttps:

  Standardbelegung: true

  Mit dieser Option kann festgelegt werden, ob der SOAP-Dienst über HTTPS veröffentlicht wird. Für den ordnungsgemäßen Betrieb ist ein Server-Zertifikat erforderlich.

  Mögliche Werte: true, false

- de.cau.cs.kieler.kwebs.jaxws.httpsAddress:

  Standardbelegung: https://localhost:8443/layout

  Mit dieser Option wird der für die Veröffentlichung des SOAP-Dienstes über HTTPS verwendete Endpunkt konfiguriert. Von der definierten Adresse werden dazu der Host, der Port und der Kontext verwendet.

- de.cau.cs.kieler.kwebs.httpsKeystore.jks.path:

  Standardbelegung: server/kwebs/security/keystores/server.jks

Mit dieser Option wird der Pfad zum Java-Keystore definiert, der das Zertifikat für die Veröffentlichung des SOAP-Dienstes über HTTPS enthält. Das Plug-In des Servers (de.cau.cs.kieler.kwebs.server) enthält einen Default-Keystore, der ein selbst zertifiziertes Zertifikat beinhaltet. Dieses sollte für den realen Betrieb durch ein von einer vertrauenswürdigen Zertifizierungsstelle ausgestelltes Zertifikat ersetzt werden.

- de.cau.cs.kieler.kwebs.httpsKeystore.jks.pass:

  Standardbelegung: server

  Mit dieser Option wird das Passwort definiert, welches für den Zugriff auf den Keystore bei Veröffentlichung des SOAP-Dienstes über HTTPS benötigt wird.

- de.cau.cs.kieler.kwebs.publishJeti:

  Standardbelegung: true

  Mit dieser Option kann festgelegt werden, ob der jETI-Dienst veröffentlicht wird.

  Mögliche Werte: true, false

- de.cau.cs.kieler.kwebs.jeti.provider.id:

  Standardbelegung: KIELERLAYOUT

  Mit dieser Option wird die Anbieterkennung des jETI-Toolservers festgelegt, der von KWebS für die Veröffentlichung des jETI-Dienstes verwendet wird.

- de.cau.cs.kieler.kwebs.jeti.sessions.timeout:

  Standardbelegung: 300000

  Diese Option definiert die Zeit in Millisekunden, nach deren Verstreichen der jETI-Toolserver eine nicht aktive Sitzung verwirft.

- de.cau.cs.kieler.kwebs.jeti.sessions.checkinterval:

  Standardbelegung: 30000

  Diese Option definiert das Intervall in Millisekunden, in dem der jETI-Toolserver auf nicht aktive Sitzungen prüft.

- de.cau.cs.kieler.kwebs.jeti.debug:

  Standardbelegung: false

  Diese Option definiert, ob der von KWebS verwendete jETI-Toolserver im Debugmodus betrieben wird.

  Mögliche Werte: true, false

- de.cau.cs.kieler.kwebs.jeti.toolxml:

  Standardbelegung: server/jeti/config/tools/tools.xml

Diese Option definiert den Pfad, unter dem der Tool-Descriptor des in KWebS integrierten jETI-Toolservers hinterlegt ist.

- de.cau.cs.kieler.kwebs.jeti.server.hostname:

  Standardbelegung: localhost

  Diese Option definiert den Host, unter dem der jETI-Toolserver den Dienst veröffentlicht.

- de.cau.cs.kieler.kwebs.jeti.connector.sepp.port:

  Standardbelegung: 9867

  Diese Option definiert den Port, unter dem der jETI-Toolserver den Dienst veröffentlicht.

- de.cau.cs.kieler.kwebs.jeti.sessions.folder:

  Standardbelegung: server/jeti/sessions

  Diese Option definiert den Pfad des Verzeichnisses, in dem der jETI-Toolserver die temporären Sitzungsverzeichnisse anlegt und verwaltet.

- de.cau.cs.kieler.kwebs.jeti.log4j.config:

  Standardbelegung: server/jeti/config/log4j/log4j.properties

  Diese Option definiert den Pfad zur Konfigurationsdatei des von jETI intern verwendeten Loggers *Log4J*.

- de.cau.cs.kieler.kwebs.jeti.logpath:

  Standardbelegung: server/jeti/logs/jeti.log

  Diese Option definiert den Pfad zur Logdatei von jETI.

- de.cau.cs.kieler.kwebs.server.poolsize:

  Standardbelegung: 10

  Diese Option definiert die maximale Menge an von KWebS nebenläufig bearbeiteten Anfragen.

- de.cau.cs.kieler.kwebs.server.backlog:

  Standardbelegung: 75

  Diese Option legt fest, wie viele Anfragen KWebS zur Bearbeitung vorhält, bevor weitere Anfragen abgelehnt werden.

- de.cau.cs.kieler.kwebs.log.path:

  Standardbelegung: server/kwebs/logs/kwebs.log

  Mit dieser Option wird der Pfad zur Logdatei von KWebS definiert.

- de.cau.cs.kieler.kwebs.log.size:

  Standardbelegung: 1

  Mit dieser Option wird die maximale Größe der Logdatei von KWebS in Megabyte definiert.

- de.cau.cs.kieler.kwebs.graphviz.path:

  Mit dieser Option wird der Pfad zur ausführbaren Datei „dot“ des Graphviz-Paketes definiert.

- de.cau.cs.kieler.kwebs.graphviz.timeout:

  Standardbelegung: 60000

  Diese Option kann für die Definition des Timeouts für Graphviz-basiertes Layout verwendet werden. Der angegebene Wert wird in Millisekunden gewertet.

- de.cau.cs.kieler.kwebs.ogdf.timeout:

  Standardbelegung: 60000

  Diese Option kann für die Definition des Timeouts für OGDF-basiertes Layout verwendet werden. Der angegebene Wert wird in Millisekunden gewertet.

- de.cau.cs.kieler.kwebs.supportServerAddress:

  Standardbelegung: http://localhost:8444

  Mit dieser Option wird der für die Veröffentlichung des Support Servers verwendete Endpunkt konfiguriert. Von der definierten Adresse werden dazu der Host und der Port verwendet.

- de.cau.cs.kieler.kwebs.publishSupportServer:

  Standardbelegung: true

  Mit dieser Option kann festgelegt werden, ob der Support Server veröffentlicht wird.

  Mögliche Werte: true, false

- de.cau.cs.kieler.kwebs.management.port:

  Standardbelegung: 23456

  Mit dieser Option wird der Port definiert, unter dem die Management-Schnittstelle lokal erreichbar ist.

C. Codebeispiele

C.1. Der Vertrag des SOAP-Dienstes

Dieser Abschnitt zeigt einen Ausschnitt des Vertrages, der für die Code-Generierung des SEI des SOAP-Dienstes und auch für die Generierung des in KIELER verwendeten Clients verwendet wurde. Der abgebildete Vertrag beschränkt sich auf die graphLayout-Operation. Der vollständige Vertrag kann unter der Adresse `http://rtsys.informatik.uni-kiel.de:9442/layout?wsdl` eingesehen werden.

```
<?xml version='1.0' encoding='UTF-8'?>
<definitions
  name="LayoutService"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://rtsys.informatik.uni-kiel.de/layout"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://rtsys.informatik.uni-kiel.de/layout">

  <types>
    <xs:schema
      version="1.0"
      xmlns:tns="http://rtsys.informatik.uni-kiel.de/layout"
      xmlns:xs="http://www.w3.org/2001/XMLSchema"
      targetNamespace="http://rtsys.informatik.uni-kiel.de/layout">
      <xs:element name="graphLayout" type="tns:graphLayout"/>
      <xs:element name="graphLayoutResponse" type="tns:graphLayoutResponse"/>
      <xs:element name="ServiceFault" type="tns:ServiceFault"/>
      <xs:complexType name="graphLayout">
        <xs:sequence>
          <xs:element name="serializedGraph" type="xs:string"/>
          <xs:element name="informat" type="xs:string"/>
          <xs:element name="outformat" type="xs:string" nillable="true"/>
          <xs:element name="options" type="tns:graphLayoutOption"
                      nillable="true" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
      </xs:complexType>
      <xs:complexType name="graphLayoutResponse">
        <xs:sequence>
        <xs:element name="return" type="xs:string"/>
        </xs:sequence>
      </xs:complexType>
      <xs:complexType name="graphLayoutOption">
        <xs:sequence>
          <xs:element name="id" type="xs:string"/>
          <xs:element name="value" type="xs:string"/>
        </xs:sequence>
      </xs:complexType>
```

```
      <xs:complexType name="ServiceFault">
        <xs:sequence>
          <xs:element name="code" type="xs:int"/>
          <xs:element name="message" type="xs:string"/>
        </xs:sequence>
      </xs:complexType>
    </xs:schema>
  </types>

  <message name="graphLayout">
    <part name="parameters" element="tns:graphLayout"/>
  </message>

  <message name="graphLayoutResponse">
    <part name="parameters" element="tns:graphLayoutResponse"/>
  </message>

  <message name="ServiceFault">
    <part name="parameters" element="tns:ServiceFault"/>
  </message>

  <portType name="LayoutServicePort">
    <operation name="graphLayout">
      <input message="tns:graphLayout"
             wsam:Action="http://rtsys.informatik.uni-kiel.de/layout/
                          LayoutServicePort/graphLayoutRequest"/>
      <output message="tns:graphLayoutResponse"
             wsam:Action="http://rtsys.informatik.uni-kiel.de/layout/
                          LayoutServicePort/graphLayoutResponse"/>
      <fault name="ServiceFault" message="tns:ServiceFault"
             wsam:Action="http://rtsys.informatik.uni-kiel.de/layout/
                          LayoutServicePort/ServiceFault"/>
    </operation>
  </portType>

  <binding name="LayoutServicePortBinding" type="tns:LayoutServicePort">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
    <operation name="graphLayout">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </input>
      <output>
        <soap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </output>
      <fault name="ServiceFault">
         <soap:fault name="ServiceFault"/>
      </fault>
    </operation>
  </binding>

  <service name="LayoutService">
    <port name="LayoutServicePort" binding="tns:LayoutServicePortBinding">
      <soap:address location="http://localhost:8442/layout"/>
    </port>
  </service>

</definitions>
```

Listing C.1: Ausschnitt des Vertrages des SOAP-Dienstes

C.2. SEI des SOAP-Dienstes

Dieser Abschnitt zeigt das aus dem Vertrag des SOAP-Dienstes generierte SEI.

```
package de.cau.cs.kieler.kwebs.jaxws;

import java.util.List;
import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.WebService;
import javax.xml.bind.annotation.XmlSeeAlso;
import javax.xml.ws.Action;
import javax.xml.ws.FaultAction;
import javax.xml.ws.RequestWrapper;
import javax.xml.ws.ResponseWrapper;

import de.cau.cs.kieler.kwebs.GraphLayoutOption;

/**
 * This class was generated by the JAX-WS RI.
 * JAX-WS RI 2.2.3-b01-
 * Generated source version: 2.2
 *
 */
@WebService(
  name = "LayoutServicePort",
  targetNamespace = "http://rtsys.informatik.uni-kiel.de/layout"
)
@XmlSeeAlso({ObjectFactory.class})
public interface LayoutServicePort {

  /**
   *
   * @param serializedGraph
   * @param format
   * @param options
   * @return
   * returns java.lang.String
   * @throws ServiceFault_Exception
   */
  @WebMethod
  @WebResult(targetNamespace = "")
  @RequestWrapper(
    localName = "graphLayout",
    targetNamespace = "http://rtsys.informatik.uni-kiel.de/layout",
    className = "de.cau.cs.kieler.kwebs.jaxws.GraphLayout"
  )
  @ResponseWrapper(
    localName = "graphLayoutResponse",
    targetNamespace = "http://rtsys.informatik.uni-kiel.de/layout",
    className = "de.cau.cs.kieler.kwebs.jaxws.GraphLayoutResponse"
```

```
  )
  @Action(
    input = "http://rtsys.informatik.uni-kiel.de/layout/LayoutServicePort/graphLayoutRequest",
    output = "http://rtsys.informatik.uni-kiel.de/layout/LayoutServicePort/graphLayoutResponse",
    fault = {
      @FaultAction(
        className = ServiceFault_Exception.class,
        value = "http://rtsys.informatik.uni-kiel.de/layout/LayoutServicePort/ServiceFault"
      )
    }
  )
  public String graphLayout(
    @WebParam(name = "serializedGraph", targetNamespace = "")
    String serializedGraph,
    @WebParam(name = "informat", targetNamespace = "")
    String informat,
    @WebParam(name = "outformat", targetNamespace = "")
    String outformat,
    @WebParam(name = "options", targetNamespace = "")
    List<GraphLayoutOption> options
  ) throws ServiceFault_Exception;

  /**
   *
   * @return
   * returns java.lang.String
   * @throws ServiceFault_Exception
   */
  @WebMethod
  @WebResult(targetNamespace = "")
  @RequestWrapper(
    localName = "getServiceData",
    targetNamespace = "http://rtsys.informatik.uni-kiel.de/layout",
    className = "de.cau.cs.kieler.kwebs.jaxws.GetServiceData")
  @ResponseWrapper(
    localName = "getServiceDataResponse",
    targetNamespace = "http://rtsys.informatik.uni-kiel.de/layout",
    className = "de.cau.cs.kieler.kwebs.jaxws.GetServiceDataResponse")
  @Action(
    input = "http://rtsys.informatik.uni-kiel.de/layout/LayoutServicePort/getServiceDataRequest",
    output = "http://rtsys.informatik.uni-kiel.de/layout/LayoutServicePort/getServiceDataResponse",
    fault = {
      @FaultAction(
        className = ServiceFault_Exception.class,
        value = "http://rtsys.informatik.uni-kiel.de/layout/LayoutServicePort/ServiceFault"
      )
    }
  )
  public String getServiceData() throws ServiceFault_Exception;

  /**
   *
   * @param previewImage
   * @return
   * returns byte[]
   * @throws ServiceFault_Exception
   */
  @WebMethod
  @WebResult(targetNamespace = "")
```

```
  @RequestWrapper(
    localName = "getPreviewImage",
    targetNamespace = "http://rtsys.informatik.uni-kiel.de/layout",
    className = "de.cau.cs.kieler.kwebs.jaxws.GetPreviewImage"
  )
  @ResponseWrapper(
    localName = "getPreviewImageResponse",
    targetNamespace = "http://rtsys.informatik.uni-kiel.de/layout",
    className = "de.cau.cs.kieler.kwebs.jaxws.GetPreviewImageResponse"
  )
  @Action(
    input = "http://rtsys.informatik.uni-kiel.de/layout/LayoutServicePort/getPreviewImageRequest",
    output = "http://rtsys.informatik.uni-kiel.de/layout/LayoutServicePort/getPreviewImageResponse",
    fault = {
      @FaultAction(
        className = ServiceFault_Exception.class,
        value = "http://rtsys.informatik.uni-kiel.de/layout/LayoutServicePort/ServiceFault"
      )
    }
  )
  public byte[] getPreviewImage(
    @WebParam(name = "previewImage", targetNamespace = "")
    String previewImage
  ) throws ServiceFault_Exception;

}
```

Listing C.2: Aus dem Vertrag generiertes SEI des SOAP-Dienstes

C.3. Tool-Descriptor des jETI-Dienstes

Dieser Abschnitt zeigt den Tool-Descriptor, den KWebS für die Bereitstellung des jETI-Dienstes verwendet.

```
<etitoolserver>

  <tool name="graphLayout" active="true" method="graphLayout"
        class="de.cau.cs.kieler.kwebs.server.service.JetiService">
    <description></description>
    <longDescription></longDescription>
    <parameter name="INPUT_GRAPH" required="true"
               class="de.unido.ls5.eti.toolserver.InputFileReference"/>
    <parameter name="OUTPUT_GRAPH" required="true"
               class="de.unido.ls5.eti.toolserver.OutputFileReference"/>
    <parameter name="INPUT_FORMAT" required="true"
               class="java.lang.String"/>
    <parameter name="INPUT_OPTIONS" required="false"
               class="java.lang.String"/>
  </tool>

  <tool name="getServiceData" active="true" method="getServiceData"
        class="de.cau.cs.kieler.kwebs.server.service.JetiService">
    <description></description>
    <longDescription></longDescription>
    <parameter name="OUTPUT_SERVICEDATA" required="true"
```

```
                class="de.unido.ls5.eti.toolserver.OutputFileReference"/>
  </tool>

  <tool name="getPreviewImage" active="true" method="getPreviewImage"
        class="de.cau.cs.kieler.kwebs.server.service.JetiService">
    <description></description>
    <longDescription></longDescription>
    <parameter name="INPUT_PREVIEWIMAGE" required="true"
                class="java.lang.String"/>
    <parameter name="OUTPUT_PREVIEWIMAGE" required="true"
                class="de.unido.ls5.eti.toolserver.OutputFileReference"/>
  </tool>

</etitoolserver>
```

Listing C.3: Tool-Descriptor des jETI-Dienstes

C.4. Batch-Task zur Installation des Servers

Dieser Abschnitt zeigt das Skript des Batch-Tasks, der vom Job Product_KWebS_-Server nach dem erfolgreichen Bauen des Servers für dessen Installation auf der virtuellen Maschine verwendet wird.

```
# Distribute the nightly KWebS build on the server
# responsible for publishing the layout service
#
# Author swe
# Last edit 10.08.2011

# Create and/or clear necessary folders if absent or clear its content
sudo -u kieler ssh layout@layout 'mkdir -p /home/layout/kwebs-nightly;
  mkdir -p /home/layout/kwebs-backup;
  rm -rf /home/layout/kwebs-nightly/kwebs_*-linux.gtk.x86_64.zip;
  rm -rf /home/layout/kwebs-backup/*'

# Copy the latest linux x64 build to the server
sudo -u kieler scp `ls $WORKSPACE/N.kwebs/kwebs_*-linux.gtk.x86_64.zip`
  layout@layout:kwebs-nightly

# Set scripts to executable
sudo -u kieler ssh layout@layout 'cd kwebs; chmod -R u+x *.sh'

# Stop the layout service if it is running
sudo -u kieler ssh layout@layout '[ -f /home/layout/kwebs/kwebs_stop.sh ] && cd kwebs;
  ./kwebs_stop.sh'

# Backup user configurable files
sudo -u kieler ssh layout@layout '[ -f /home/layout/kwebs/kwebs_backup.sh ] && cd kwebs;
  ./kwebs_backup.sh'

# Delete the prior distribution
sudo -u kieler ssh layout@layout '[ -d /home/layout/kwebs ] && rm -rf /home/layout/kwebs'

# Unzip the newly uploaded distribution
sudo -u kieler ssh layout@layout 'unzip kwebs-nightly/kwebs_*-linux.gtk.x86_64.zip'
```

```
# Extract kwebs management scripts from server plugin archive
sudo -u kieler ssh layout@layout 'chmod u+x bootstrap.sh; ./bootstrap.sh'

# Set scripts to executable, executable status may have changed after fresh deployment
sudo -u kieler ssh layout@layout 'cd kwebs; chmod -R u+x *.sh'

# Restore user configurable files
sudo -u kieler ssh layout@layout 'cd kwebs; ./kwebs_restore.sh'

# Set scripts to executable (just to be sure) and start the layout service
sudo -u kieler ssh layout@layout 'cd kwebs; ./kwebs_start.sh'
```

Listing C.4: Batch-Task zur Installation des Servers

C.5. Skripte

Dieser Abschnitt zeigt die während der Installation des Servers auf der virtuellen Maschine ausgeführten Skripte.

C.5.1. kwebs_stop.sh

```
#!/bin/bash

# Shutdown script for the KWebS server
# Version 0.00
#
# Author swe
# Last edit 10.08.2011

# !!!!!
# Please remember making this script executable by
# doing chmod u+x kwebs_stop.sh
# !!!!!

PIDFILE=kwebs.pid

if [ -f $PIDFILE ];
then

  # Read the pid from file
  read PID < $PIDFILE

  # Kill child processes of the given process id
  for i in `ps -ef | awk '$3 == '$PID' { print $2 }'`
  do
      kill -9 $i
  done

  # Kill process with the given process id
  kill -9 ${PID}

  # Delete the file containing the process id
  rm -rf $PIDFILE
```

```
  # Wait some time just to be sure...
  sleep 5

fi
```

Listing C.5: Stoppen des Servers

C.5.2. kwebs_backup.sh

```
#!/bin/bash

# Backup script for the KWebS server
# Version 0.00
#
# Author swe
# Last edit 10.08.2011

# !!!!!
# Please remember making this script executable by
# doing chmod u+x kwebs_backup.sh
# !!!!!

KWEBS_BACKUP=../kwebs-backup

# Assure backup folder exists and is empty

if [ ! -d $KWEBS_BACKUP ];
then
  mkdir -p $KWEBS_BACKUP
else
  rm -rf $KWEBS_BACKUP/*
fi

# Backup user server configuration file
if [ -f kwebs.user ];
then
  mv kwebs.user ../kwebs-backup
fi

# Backup server configuration and content folder
if [ -d server ];
then
  mkdir -p $KWEBS_BACKUP/server
  mv server/* $KWEBS_BACKUP/server
fi
```

Listing C.6: Sicherung der Betreibereinstellungen

C.5.3. kwebs_restore.sh

```
#!/bin/bash

# Restore script for the KWebS server
# Version 0.00
#
```

```
# Author swe
# Last edit 10.08.2011

# !!!!!
# Please remember making this script executable by
# doing chmod u+x kwebs_restore.sh
# !!!!!

KWEBS_BACKUP=../kwebs-backup

# Assure backup folder exists

if [ ! -d $KWEBS_BACKUP ];
then
  echo Backup does not exist
  exit 1
fi

# Restore user server configuration file
if [ -f $KWEBS_BACKUP/kwebs.user ];
then
  mv $KWEBS_BACKUP/kwebs.user .
fi

SERVER_SSL_CONTENT=server/kwebs/security/keystores
WEB_SSL_CONTENT=server/kwebs/web/security

# Restore server SSL configuration
if [ -d $KWEBS_BACKUP/$SERVER_SSL_CONTENT ];
then
  mkdir -p $SERVER_SSL_CONTENT
  mv $KWEBS_BACKUP/$SERVER_SSL_CONTENT/*.jks $SERVER_SSL_CONTENT
fi

# Restore user SSL configuration
if [ -d $KWEBS_BACKUP/$WEB_SSL_CONTENT ];
then
  mkdir -p $WEB_SSL_CONTENT
  mv $KWEBS_BACKUP/$WEB_SSL_CONTENT/client.jks $WEB_SSL_CONTENT
fi
```

Listing C.7: Wiederherstellung der Betreibereinstellungen

C.5.4. kwebs_start.sh

```
#!/bin/bash

# Startup script for the KWebS server
# Version 0.00
#
# Author swe
# Last edit 10.08.2011

# !!!!!
# Please remember making this script executable by
# doing chmod u+x kwebs_start.sh
# !!!!!
```

```
# Start the server ignoring the HUP signal so the server
# keeps running after closing a terminal connection
echo Starting server...
nohup ./kwebs >kwebs.out 2>kwebs.err </dev/null &

# Remember process id
PID=$!

# Wait some time just to be sure...
echo Waiting for server to finish bootup process...
sleep 10

# Test if server is running
if [ -f /proc/$PID/exe ]; then

  # If so, store process id in file
  echo KWebS server started, process id is $PID
  echo $PID > kwebs.pid

else

  # If not, exit with an error code
  echo KWebS server failed to start, see kwebs.out and kwebs.err for details
  echo

  echo kwebs.out:
  cat kwebs.out

  echo
  echo kwebs.err:
  cat kwebs.err

  exit 1

fi
```

Listing C.8: Starten des Servers