Entwicklung eines Editors in Eclipse zur Erstellung eines Bitemporal Complex Event Processing System

Masterarbeit

Zur Erlangung des akademischen Grades
Master of Science (MSc)
im Masterstudium
Wirtschaftsinformatik

Autor
Sebastian Hochgatterer, BSc

Betreuung
o. Univ.-Prof. DI Dr. Michael Schrefl

Mitbetreuung
Mag. Michael Huemer

Institut
Institut für Wirtschaftsinformatik
Data & Knowledge Engineering
Johannes Kepler Universität Linz

Linz, Juni 2014

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Die vorliegende Masterarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Linz, 6. Juni 2014

Sebastian Hochgatterer

Kurzfassung

Diese Arbeit beschreibt die Entwicklung eines visuellen Editors, der es ermöglicht, ein Bitemporal Complex Event Processing System (BiCEPS) zu modellieren und anschließend auszuführen. Für die Umsetzung des Editors wird ein Eclipse Plug-in implementiert, welches die bestehenden Frameworks EMF und Graphiti verwendet. EMF wird für die Datenverwaltung und Graphiti für die visuelle Darstellung der Objekte in Eclipse eingesetzt. Die Objekte im Editor sind über eine standardisierte Schnittstelle (XMI) parametrisierbar. Zusätzlich wird noch ein Mapper, für die Umwandlung des visuellen BiCEPS Modells in ein ausführbares Programm, implementiert.

Abstract

This paper describes the implementation of a visual editor which enables to model an executable bitemporal complex event processing system (BiCEPS) in a drag and drop environment. The editor is implemented as an Eclipse plug-in and uses EMF for the data layer and Graphiti for the object presentation. These objects are specified by the use of the XML Metadata Interchange (XMI) standard. Moreover, a mapper is implemented for the purpose of transforming the modeled BiCEPS to an executable program.

Inhaltsverzeichnis

1. Einleitung

In diesem Kapitel wird der Gegenstand und das Ziel der Masterarbeit beschrieben. Zuerst wird diese Arbeit motiviert, bevor das Ziel dieser Arbeit erörtert und ein Fallbeispiel eingeführt wird. Abschließend wird die Struktur der Arbeit vorgestellt.

1.1. Motivation

In den letzten Jahren hat sich gezeigt, dass die automatische Verarbeitung von Informationen einen immer höheren Stellenwert in Informations- und Kommunikationssystemen einnimmt – nicht nur für Unternehmen, sondern für jedeN TeilnehmerIn der heutigen Informationsgesellschaft. Die Menge an verfügbaren Daten, z.B. im World Wide Web, steigt rasant und es wird einerseits immer wichtiger wesentliche von unwesentlicher Information zu unterscheiden und andererseits diese Informationen für Entscheidungsfindungen zu verknüpfen. Dadurch, dass jeder im World Wide Web Informationen über Ereignisse aus der realen Welt veröffentlichen kann ist eine schier unüberschaubare Menge an Informationen verfügbar, jedoch nur ein Bruchteil davon, unter Umständen von unterschiedlichen Quellen, für eine Entscheidungsfindung relevant. Zum Beispiel, um einen Freund rechtzeitig vom Bahnhof abzuholen ist nur eine einzelnes Zugankunftsereignis relevant, welches jedoch verknüpft mit der aktuellen Verkehrssituation wesentlich an Wert gewinnt. Abhängig von den gefilterten und anschließend aggregierten Ereignissen werden unterschiedliche Aktionen durchgeführt, z.B. mit dem Auto oder mit den öffentlich Verkehrsmitteln zum Bahnhof zu fahren.

Um eine systematische, zeitnahe und automatische Verarbeitung von Ereignissen zu ermöglichen, wurden Complex Event Processing Systeme (CEP) entwickelt. Unter CEP versteht man alle Methoden, Werkzeuge und Techniken, welche angewandt werden, um sinnvoll auf erkannte Ereignisse automatisch reagieren zu können (Eckert & Bry, 2009). PeaCE-full Web Event Extraction and Processing ist ein Projekt der Johannes Kepler Univerisät Linz und der University of Oxford, welches sich zum Ziel setzt Ereignisse von unterschiedlichen Quellen aus dem Web zu extrahieren, diese Ereignisse zu komplexen Ereignissen zu verarbeiten und aufgrund dieser komplexen Ereignisse Aktionen durchzuführen (Furche et al., 2013b, S.1-2). Um den Anforderungen zur Verarbeitung von Ereignissen aus dem Web, welche veränderlich sind, gerecht zu werden, sind Ereignisse in PeaCE bitemporal, d.h. sie verfügen über zwei Zeitdimensionen, einen Eintrittszeitpunkt und einen Wahrnehmungszeitpunkt (Furche et al., 2013a). Im weiteren dieser Arbeit wird ein System welches auf diesem Ansatz fundiert *Bitemporale Complex Event Processing System* (BiCEPS) genannt. Derzeit gibt es keine Möglichkeit, ein BiCEPS ohne Expertenwissen zu erstellen und ist somit für den durchschnittlichen Anwender nicht möglich. Aus diesem Grund soll im Rahmen dieser Arbeit ein Editor zum einfachen Erstellen einer solchen Anwendungen implementiert werden.

1.2. Zielsetzung und Lösungsansatz

Ziel dieser Masterarbeit ist die Erstellung einer visuellen Benutzerschnittstelle in Form eines Editors, welcher einem Benutzer erlaubt ein ausführbares Programm auf Basis des bereits existierenden BiCEPS Frameworks zu erzeugen. Dieses BiCEPS Framework ist die Implementierung des Ansatzes welcher in (Furche et al., 2013a) und (Furche et al., 2013b) vorgestellt wird, und besteht aus drei Komponenten, einem Event Detector zum Extrahieren von Ereignissen aus dem Web, einem Bitemporal Complex Event Processor zum Verarbeiten dieser Ereignisse und einem Action Executor zum Ausführen von Aktionen als Reaktion auf erkannte komplexe Ereignisse. Weiters soll dieser Editor über eine standardisierte Schnittstelle parametrisierbar und um neue Komponenten erweiterbar sein.

Um diese Anforderungen zu erfüllen, muss eine Anbindung an das bestehenden BiCEPS Framework in Form eines Editors erstellt werden. Dazu muss eine visuelle Repräsentation der Komponenten eines BiCEPS im Editor geschaffen werden. Diese Komponenten sollten einfach über eine Benutzerschnittstelle für den jeweiligen Anwendungsfall konfigurierbar sein. Dies bedeutet, dass eine Definition ihrer Eigenschaften ohne Expertenwissen möglich sein muss. Durch die Verbindungen der Komponenten, welche den Ereignisfluss zwischen den Komponenten darstellen, wird ein visuelles Modell erstellt, welches über das BiCEPS Framework ausführbar ist. Wichtig dabei ist, dass die Funktionalität der BiCEPS Komponenten durch den Benutzer erweitert werden kann. Der Editor sollte darum von außen konfigurierbar und erweiterbar sein, ohne dass die Implementierung des Editors geändert werden muss. Für die Umsetzung des visuellen Editors wird die weit verbreitete und bewährte Entwicklungsumgebung Eclipse verwendet.

1.3. Fallbeispiel

Um die Erklärungen für das BiCEPS in der Masterarbeit zu erleichtern und um ein Verständnis für dieses System zu schaffen, wird folgendes fortlaufende Beispiel verwendet. Diese Beschreibung dient als Rahmen und wird in dieser Arbeit als Beispiel herangezogen und erweitert.

Das Beispiel dreht sich um ein Meeting von zwei Kollegen, wobei der eine Kollege den anderen vom Bahnhof abholt. Ein möglicher Anwendungsfall könnte folgendermaßen aussehen:

> Zwei Kollegen vereinbaren ein Treffen für den 20.05.2014 in Linz. Der eine wohnt in Linz, der andere wohnt in Wien. Der Wiener bucht für dieses Datum einen Schnellzug von Wien nach Linz, der am 20.05.2014 planmäßig um 12:00 in Linz ankommen wird. Der Linzer Kollege wird ihn am Bahnhof abholen, dazu muss er sich eine halbe Stunde vor Ankunft des Zuges auf den Weg machen, um rechtzeitig am Bahnhof anzukommen. Um diese Ankunft nicht zu verpassen, möchte er 30 Minuten vor Zugankunft erinnert werden.

Der Linzer Kollege will jedoch aufgrund seines engen Zeitplans keine unnötige Wartezeit am Bahnhof in Kauf nehmen. Er will im Falle einer Verspätung des Zuges sich später auf den Weg machen, um noch Erledigungen im Büro durchführen zu können. Um diese Informationen abzufragen, muss er ständig den Status der Zugankunft abrufen. Dies kann beispielsweise über eine Webapplikation der nationalen Zuggesellschaft passieren.

Die Überwachung der Zugankunft und die damit verbundenen Web-Abfragen sind für den Linzer Kollegen sehr mühsam und aufwändig.

1.4. Struktur der Arbeit

Diese Masterarbeit ist in drei große Teile gegliedert. Im ersten Teil werden die Grundlagen, die für die Implementierung des Editors notwendig sind, beschrieben. Im Zuge dessen werden zuerst Grundlagen zur Verarbeitung von Ereignissen beschrieben (Kapitel 2), bevor Grundlagen zur Erstellung eines Editors in Eclipse erläutert werden (Kapitel 3). Im zweiten Teil wird die Umsetzung des Editors und Details dazu angeführt. Zuerst werden Anforderungen an den zu implementierenden Editor definiert (Kapitel 4). Anschließend wird aufgrund dieser Anforderungen evaluiert welches Framework zur Erstellung des BiCEPS-Editors verwendet werden soll (Kapitel 5). Aufbauend auf diesen Erkenntnissen wird das Design des visuellen Editors entwickelt (Kapitel 6). Die Implementierung selbst ist in Kapitel 7 beschrieben. Im dritten und letzten Teil dieser Arbeit wird die Funktionsweise des Editors in Form eines Benutzerhandbuchs dargelegt (Kapitel 8).

Teil I.

Grundlagen

2. Komplexe Eventverarbeitung

Dieses Kapitel dient als Einführung in die komplexe Ereignisverarbeitung und erläutert die Grundlagen des Complex Event Processing (CEP). Anhand von CEP wird das Bitemporal Complex Event Processing System (BiCEPS) erklärt. BiCEPS verarbeitet Ereignisse und ermöglicht Reaktionen auf diese. Zusätzlich ermöglicht es, auf Modifikationen von Ereignissen auch nach deren Verarbeitung zu reagieren und erweitert herkömmliche CEP Systeme um eine zweite Zeitdimension. Es können Aktionen abhängig von Ereignissen und deren Modifikationen definiert werden, dadurch kann auf das zu späte Eintreffen von Ereignissen reagiert werden.

Im Kapitel 2.1 werden die Grundlagen der komplexen Ereignisverarbeitung dargelegt und beschrieben. Auf deren Basis werden die Erweiterung des BiCEPS erläutert (Kapitel 2.2). Die Syntax, die für die Umsetzung eines BiCEPS notwendig ist, wird im Kapitel 2.4 behandelt. Die Beschreibung wird mit einem Beispielszenario untermalt, welches im Kapitel 2.3 dargestellt wird. Abschließend wird der Aufbau des BiCEPS Frameworks in Kapitel 2.5 beschrieben.

Im weiteren Verlauf der Arbeit werden Ereignisse und Events synonym verwendet.

2.1. Grundlagen CEP

Complex Event Processing (CEP) ermöglicht die systematische und automatische Verarbeitung von Ereignissen (Events), die speziell bei ereignisgesteuerten Informationsystemen verwendet wird (Eckert & Bry, 2009). Tritt ein Ereignis ein, wird dieses von einem CEP verarbeitet und gegebenenfalls wird eine Aktion ausgelöst. Unter einem Ereignis versteht man ein Objekt, welches ein Geschehen oder eine Aktivität darstellt. Dieses Objekt spiegelt also eine Aktivität der realen Welt in einem computergestützten System wider. Ein Ereignis tritt zu einem bestimmten Zeitpunkt ein. (Luckham, 2002, S.52)

> **Beispiel 1** Die Einfahrt eines Zuges am Bahnsteig ist ein solches Ereignis. Der Zug kommt genau zum Zeitpunkt 12:00 in Linz am Bahnhof an.

Einzelne Ereignisse enthalten oft nicht den gewünschten Informationsgehalt, der für eine gewisse Aktion notwendig ist. Somit sind mehrere Ereignisse aus unterschiedlichen Quellen sowie deren Zusammenhang notwendig, um Aktionen auszulösen (Luckham, 2002).

Typische CEP Systeme (Demers et al., 2007; Jacobsen et al., 2010; Li & Jacobsen, 2005; Luckham, 1998; Luckham & Vera, 1995; Schultz-Møller et al., 2009; Wu et al., 2006) fassen die Verarbeitung von mehreren Ereignissen aus unterschiedlichen Quellen in einem komplexen Ereignis (Beispiel 2) zusammen. Ein komplexes Ereignis setzt sich aus einer Menge von Ereignissen, die im Zusammenhang miteinander stehen, zusammen.

> **Beispiel 2** Um einen Kollegen rechtzeitig am Bahnhof abholen zu können, ist neben dem Event Zugankunft aus Beispiel 1 noch die Information notwendig, ob dieser auch abgeholt werden möchte. Das heißt, ein zusätzliches Event ist erforderlich. Diese beiden Ereignisse werden zu einem komplexen Ereignis zusammengefasst.

Der CEP Ansatz beschreibt das Konzept und die Prinzipien, die für die Verarbeitung von Ereignisströmen (Event Streams) notwendig sind. Ein Ereignisstrom ist eine Sequenz von Ereignissen, beispielsweise eine Menge von Zugankunftsereignissen. CEP Systeme erkennen vordefinierte komplexe Ereignisse in solchen Ereignisströmen und definieren Aktionen auf komplexe Ereignisse. (Luckham, 2002)

2.2. Grundkonzept Bitemporal Complex Event Processing System (BiCEPS)

Ein BiCEPS erweitert ein konventionelles CEP System in vielerlei Hinsicht: Es integriert Event Detectors, welche Ereignisse aus Quellen extrahieren. Weiters verwendet BiCEPS einen Bitemporal Complex Event Processor, das heißt, einen Eventprozessor, der eine zusätzliche Zeitdimension verwaltet. Diese zwei Zeitdimensionen ermöglichen eine Verwaltung von sich ändernden Ereignissen und die Durchführung von Aktionen aufgrund solcher Änderungen. Auch die Ausführung von Aktionen durch sogenannte Action Executors ist in BiCEPS integriert.

Im Kapitel 2.2.1 werden nun die Komponenten eines BiCEPS beschrieben, ehe ein detaillierter Vergleich zu traditionellen CEP Systemen angestellt wird (Kapitel 2.2.2).

2.2.1. Komponenten

Ein BiCEPS besteht aus drei Komponenten: Event Detector(EDT), Action Executor (AEX) und Bitemporal Complex Event Processor (BiCEP). Mindestens drei unterschiedliche Komponenten müssen in einem BiCEPS enthalten sein. Abbildung 2.1 zeigt eine mögliche Anordnung dieser Komponenten. Ein EDT ist mit einem oder mehreren BiCEPs verbunden. Ein BiCEP kann mit anderen BiCEPs und einem AEX verbunden werden. (Furche et al., 2013a)

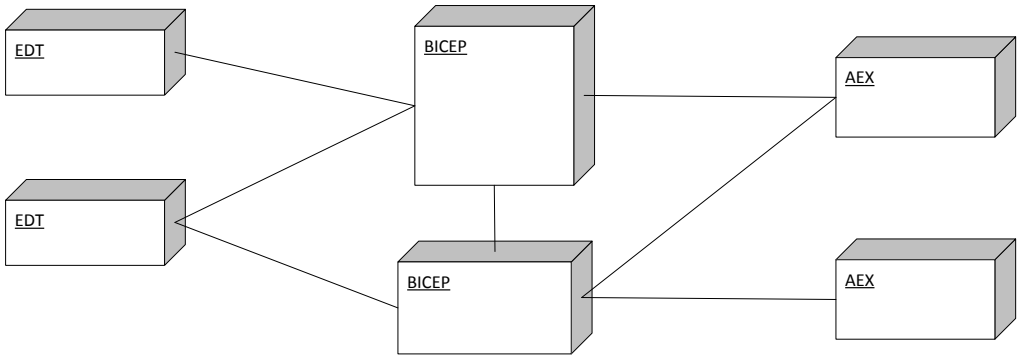

Abbildung 2.1.: BiCEPS-Grundkomponenten

EDT

Ein Event Detector (EDT) dient dazu, Events aus einer Quelle zu extrahieren und aufzubereiten, z.B. die Konvertierung von Zeitformaten. Je Quelle wird ein EDT benötigt, welcher periodisch in einem gegebenen Intervall die Quelle auf Änderungen überprüft und extrahiert, z.B. neue Events oder modifizierte Events. (Furche et al., 2013b)

> **Beispiel 3** Events über Zugankünfte (Uhrzeit, Bahnsteig, etc.) werden von der Webseite einer Zuggesellschaft extrahiert und für die Weiterverarbeitung aufbereitet.

In einem BiCEPS wird für die Extrahierung von Daten aus dem Web das Tool OXPath verwendet. OXPath wurde von der Oxford University entwickelt und ermöglicht eine effiziente und automatische Datenextrahierung. Im Gegensatz zu anderen Web Extraction Tools (Web Harvester, Lixto, etc.) kann OXPath auch Daten von javascript-basierten Webseiten extrahieren. (Furche et al., 2011)

BiCEP

Das Kernstück von BiCEPS ist sein Bitemporaler Complex Event Processor (BiCEP). Er verarbeitet Events, die von EDTs extrahiert wurden, zu komplexen Events und stößt gegebenenfalls Aktionen an. Um auf Modifikationen von erfassten Ereignissen zu reagieren, wird eine zweite Zeitdimension eingeführt. Neben dem Eintrittszeitpunkt (*occurrence-time*) wird in einem BiCEP der Wahrnehmungszeitpunkt (*detection-time*) verwaltet. Der Unterschied dieser Zeitpunkte wird im Kapitel 2.2.2 beschrieben. Die Bedingungen für die auszulösenden Aktionen werden im komplexen Ereignis festgelegt und in BiCEP verarbeitet. Die Events werden durch die Bitemporal Event Processing Language (BiCEPL) definiert. (Furche et al., 2013b) Die Syntax dieser Sprache wird in Kapitel 2.4.2 erläutert.

AEX

Ein Action Executor (AEX) wird von einem oder mehreren BiCEPs angestoßen und löst Aktionen aus. Eine Webaktion kann wiederum mittels OXPath definiert werden (Furche et al., 2013b).

> **Beispiel 4** Mittels OXPath kann beispielsweise eine Aktion definiert werden, welche sich auf einer Webseite anmeldet und ein SMS mit der Nachricht: „Komme zu spät“ versendet.

2.2.2. Vergleich zu traditionellen CEPs

BiCEPS erweitert typische CEP Systeme (Demers et al., 2007; Jacobsen et al., 2010; Li & Jacobsen, 2005; Luckham, 1998; Luckham & Vera, 1995; Schultz-Møller et al., 2009; Wu et al., 2006) um die Komponenten EDT und AEX. Ein EDT extrahiert Ereignisse aus unterschiedlichen Quellen und der AEX löst daraufhin Aktionen aus. Zusätzlich ermöglicht BiCEPS zum

herkömmlichen Complex Event Processor in CEP Systemen die bitemporale Verarbeitung von Events mittels dem BiCEP.

In bisherigen CEP Systemen werden Ereignisse zu ihrem Eintrittszeitpunkt verarbeitet. Diese Events können nicht modifiziert werden, weder ihr Eintrittszeitpunkt noch andere Attribute können sich ändern (Furche et al., 2013b). Typischerweise können diese Systeme ausschließlich auf pünktlich eintreffende Ereignisse reagieren bzw. wird kein Unterschied zwischen pünktlich und zu spät eintreffenden Ereignissen gemacht. Einige wenige Systeme, z.B. (Srivastava & Widom, 2004), können verspätete Ereignisse verarbeiten, indem diese für eine gewisse Zeit gepuffert werden. Innerhalb dieser Zeitspanne werden zu spät eintreffende Ereignisse berücksichtigt und auch Änderungen von Attributen zugelassen.

Neben den bereits erwähnten Unterschieden erweitert ein BiCEP andere CEP um eine zweite Zeitdimension. Es wird zusätzlich zum Eintrittszeitpunkt(*occurrence-time*) ein Wahrnehmungszeitpunkt (*detection-time*) für Ereignisse eingeführt. Unter Eintrittszeitpunkt versteht man den Zeitpunkt, zu dem das Ereignis in der realen Welt eintritt oder eintreten wird. Der Wahrnehmungszeitpunkt beschreibt den Zeitpunkt, wann das Ereignis vom System wahrgenommen wird. Diese beiden Zeitpunkte können voneinander abweichen, wenn beispielsweise ein Ereignis zu spät extrahiert und infolgedessen zu spät wahrgenommen wird. Das heißt, der Wahrnehmungszeitpunkt liegt nach dem tatsächlichen Eintrittszeitpunkt.(Furche et al., 2013b)

> **Beispiel 5** Bezogen auf das Fallbeispiel ist der Eintrittszeitpunkt jener Zeitpunkt, zu welchem der Zug tatsächlich in den Bahnhof einfährt. Der Wahrnehmungszeitpunkt hingegen, wann der Abholer über eine Webapplikation über die Zugankunft informiert wird. Es kann vorkommen, dass der Zug bereits im Bahnhof eingefahren ist und der Abholer noch nicht über dieses Ereignis informiert wurde. In diesem Fall ist der Wahrnehmungszeitpunkt nach dem Eintrittszeitpunkt. Umgekehrt, wenn der Abholer über die zukünftige Zugankunft informiert wird, ist der Wahrnehmungszeitpunkt vor dem Eintrittszeitpunkt.

2.2.3. Event Condition Action Model

BiCEP baut auf dem Event-Condition-Action Model auf und definiert Subscribed Event Classes, Complex Event Classes und Action Classes.

Jedes erkannte und verarbeitet Ereignis gehört genau zu einer Eventklasse. Die Eventklassen (Event Classes) teilen sich in Subscribed Event Classes und Complex Event Classes. Jede Eventklasse hat Attribute, die die Struktur eines Events definieren. Eine Teilmenge dieser Attribute bilden die Schlüsselattribute. Für alle Complex Event Classes wird für Schlüsselattribute eine Key-Subsumption angewendet. Dies bedeutet, dass jeder Primärschlüssel der Teilereignisse als Attribut im komplexen Ereignis enthalten sein muss. Unter Teilereignissen versteht man alle Ereignisse, die das komplexe Ereignis definieren.(Furche et al., 2013b)

Zusätzlich zu den Attributen und den Schlüsselattributen wird in Eventklassen die Aktionsanweisungsbedingung (Condition-Action-Statement) definiert. Jedes Condition-Action-Statement besteht aus einer Bedingung (cond) und einer daraus folgenden Aktionsanweisung (action). Diese Aktion wird angestoßen, wenn die Bedingung zutrifft. Neben diesen Eventklassen existieren Action Classes, welche eine Aktion (action) beschreiben.(Furche et al., 2013b)

Ein EDT extrahiert Subscribed Events, die durch ihre Subscribed Events Classes beschrieben werden. Die Subscribed Events werden in einem Complex Event aggregiert und in einem BiCEP verarbeitet. Ein Complex Event wird durch seine Complex Event Class beschrieben. Ein BiCEP löst eine Aktion (action) aus, die von AEX ausgeführt wird.(Furche et al., 2013b)

2.3. Anwendungsbeispiel für BiCEPS

In diesem Kapitel wird das Fallbeispiel mittels UML modelliert und alle auftretenden Ereignisse und deren Eigenschaften beschrieben.

Wie bereits in Kapitel 1.3 beschrieben, wird das Treffen immer im Vorhinein vereinbart. Der Kollege aus Wien fährt mit dem Zug nach Linz, wo er vom Bahnhof abgeholt wird. Der Abholer hat einen längeren Weg zu bewältigen und muss somit einige Zeit früher den Weg zum Bahnhof antreten, um pünktlich dort zu sein. Er erstellt eine Erinnerung, wann er den Anfahrtsweg antreten muss.

Der Zug kommt um 12:00 in Linz an, und die Errinnerung wird laut geplantem Ablauf um 11:30 aktiviert. Was ist nun, wenn der Zug Verspätung hat oder wenn keine Abholung mehr notwendig ist? Genau an diesem Punkt setzt BiCEPS an. Es kann mittels einer zweiten Zeitdimension auf veränderte Ereignisse reagiert werden. Unter „veränderte“ wird hier eine zeitliche Änderung des ursprünglichen Eintrittszeitpunktes oder eine Änderung anderer Eigenschaften des Events verstanden. Zur Lösung dieser Probleme wird BiCEPS eingesetzt. Das Vorgehen und der Einsatz eines BiCEPS wird in diesem Kapitel erläutert.

In den weiteren Paragraphen werden die Klassen, die im Diagramm abgebildet sind, detailliert beschrieben. Im folgenden werden die Eventklassen und deren Attribute Camelcase und kursiv formatiert. So spiegelt beispielsweise eine Zugankunft das reale Ereignis, die Einfahrt eines Zuges am Bahnsteig, wider. Die *ZugAnkunft* repräsentiert die dazugehörige Eventklasse.

Subscribed Event Classes Um den Anwendungsfall abzubilden, werden drei Subscribed Event Classes benötigt (Abbildung 2.2). Diese sind die *ZugAnkunft*, *ZugBuchung* und das *Meeting*.

ZugAnkunft Für jede Einfahrt eines Zuges am Bahnhof wird ein Ereignis Zugankunft ausgelöst. Dieses wird mit der gleichnamigen Klasse *ZugAnkunft* repräsentiert. Diese Klasse beinhaltet eine Zug-Nummer (*zugNr*) und ein Datum (*datum*). Die Kombination dieser beiden Attribute ermöglicht eine eindeutige Identifikation einer Zugankunft. Weiters beinhaltet eine Zugankuft das Attribut *bahnhof*, welches den Endbahnhof beschreibt.

ZugBuchung Um einen gültigen Fahrschein zu lösen, muss eine Buchung des Zuges erfolgen. Eine Zugbuchung wird für genau ein Meeting gemacht. Dies wird in der Klasse *ZugBuchung* modelliert. Die Klasse definiert einen bestimmten Zug (*ZugNr*) und ein bestimmtes Datum (*datum*). Eine Buchung ist durch eine *buchungsNr* eindeutig identifizierbar. Weiters enthält eine Buchung noch einen Abfahrtsbahnhof (*abfahrtsbahnhof*) sowie einen Ankunftsbahnhof (*ankunftsbahnhof*).

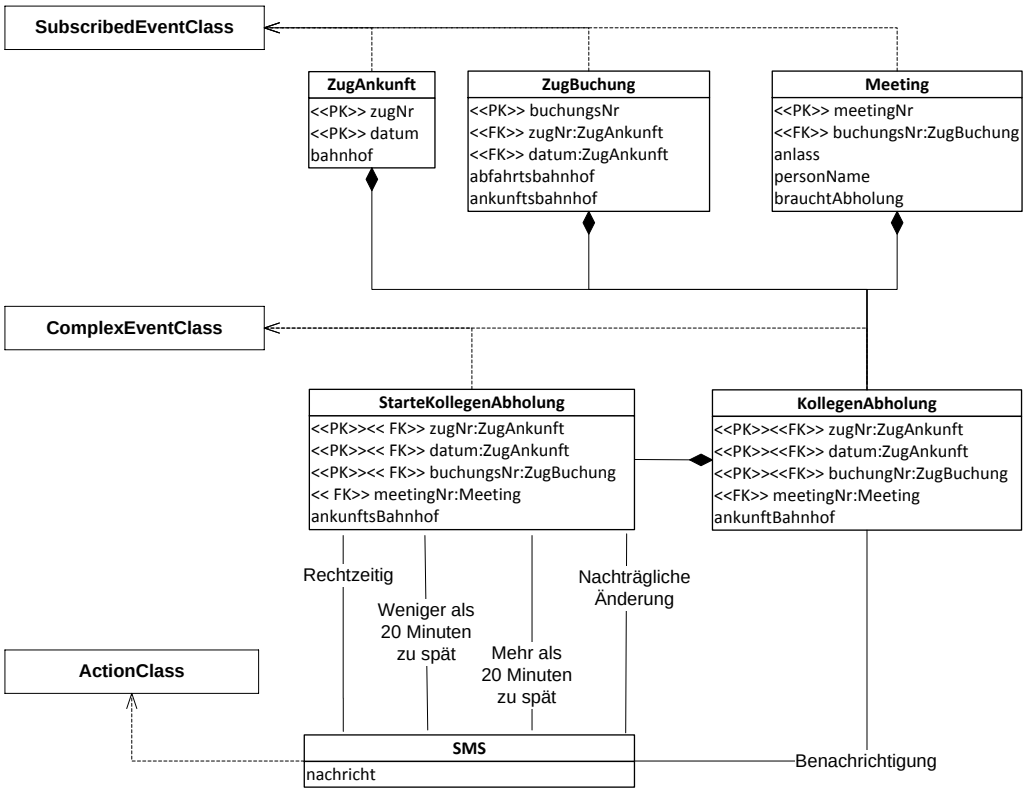

Abbildung 2.2.: BiCEPS Fallbeispiel

Meeting Das Treffen zweier Kollegen stellt ein Ereignis dar, welches in der Klasse *Meeting* abgebildet wird, wobei zu einem Meeting genau eine Zugbuchung existiert. Ein *Meeting* wird durch eine eindeutige *MeetingNr* identifiziert. Ein Treffen hat einen Verweis auf eine *buchungsNr*, da ohne ein gültiges Fahrticket keine Reise zum Treffpunkt möglich wäre. Das Flag *brauchtAbholung* kennzeichnet, ob eine Abholung vom *ankunftsBahnhof* notwendig ist. *brauchtAbholung* kann die Werte JA oder NEIN beinhalten. Zusätzlich werden noch Informationen über die Person (*personName*) und den Anlass (*anlass*) des Treffens im *Meeting* vermerkt.

Complex Event Classes Dieser Anwendungsfall erforderte die Definition von zwei Complex Event Classes: *KollegenAbholung* und *StarteKollegenAbholung*. Deren Eigenschaften werden in Abbildung 2.2 dargestellt. Die *KollegenAbholung* aggregiert die Subscribed Event Classes *ZugAnkunft*, *ZugBuchung* und *Meeting*. *StarteKollegenAbholung* ist relativ zeitlich auf *KollegenAbholung* definiert, konkret passiert ein *StarteKollegenAbholung* Event 30 Minuten vor einem *KollegenAbholung* Event.

KollegenAbholung Die Eventklasse *KollegenAbholung*, die Abholung eines Kollegen, tritt ein, falls ein *Meeting* und die dazugehörige *ZugBuchung* existieren. Weiters muss eine *ZugAnkunft* dazu vorhanden sein. Eine Abholung vom Bahnhof findet nur dann statt, wenn sie auch gewünscht wird.

StarteKollegenAbholung Wie im Beispiel beschrieben, sollte der Kollege vom Bahnhof abgeholt werden. Dazu wird explizit hingewiesen, dass die Anreise zum Bahnhof einige Zeit in Anspruch nimmt. Somit muss das Ereignis *StarteKollegenAbholung* einige Zeit (30 Minuten) vor dem tatsächlichen Eintrittszeitpunkt (geplante *ZugAnkunft* um 12:00) der *KollegenAbholung* eintreten. Die Klasse *StarteKollegenAbholung* hat die gleichen Attribute wie die Klasse *KollegenAbholung*. Es beschreibt das Ereignis, wann sich der Abholer auf den Weg zum Bahnhof macht.

Action Classes In diesem Anwendungsfall gibt es nur eine Aktionsklasse, nämlich das Versenden eines *SMS*, das unterschiedliche Nachrichten beinhalten kann. Das Szenario 1 ist jene Benachrichtigung, welche von der *KollegenAbholdung* eine SMS Aktion auslöst, wann eine Abholung zustande kommt. Die unten angeführten Szenarien(2-5) werden vom Ereignis *StarteKollegenAbholung* angestoßen und informieren den Abholer mittels SMS.

Szenario 1 Kollege will abgeholt werden.

Szenario 2 Ein rechtzeitiges Eintreten der *StarteKollegenAbholung* ist gegeben, wenn der Abholer rechtzeitig über den Status der Zugankunft informiert wird. Somit wird der Abholer auch rechtzeitig vom SMS Service mit der Nachricht „Fahre zum Bahnhof, um deinen Kollegen rechtzeitig abzuholen“ erinnert.

Falls das Event *StarteKollegenAbholung* zu spät ausgelöst wird, ergeben sich die Szenarien 3 und 4. Der Kollege kommt hier zu früh am Bahnhof an, somit ist der Abholer zu spät, da die Anfahrtszeit 30 Minuten beträgt.

Szenario 3 Ist das Event (*KollegenAbholung*) weniger als oder gleich 20 Minuten zu spät, wird der Abholer erinnert, sich schnell auf den Weg zu machen und den Kollegen zu benachrichtigen, dass er zu spät zum Bahnhof kommt.

Szenario 4 Wird das Event mehr als 20 Minuten zu spät ausgelöst, liegt folgender Sachverhalt vor. Der Abholer wird zu spät über die (*KollegenAbholung*) informiert und macht sich erst 20 Minuten nach dem Eintreten des Events auf den Weg. Somit kann er den Kollegen nicht mehr rechtzeitig abholen und wird per SMS verständigt, ein Taxi für den Kollegen zu bestellen.

Szenario 5 Ist der Abholer bereits am Weg zum Bahnhof und der Kollege kommt nicht rechtzeitig an, wird er benachrichtigt, dass der Kollege von Wien sich verspätet.

2.4. BiCEPL: BiCEP's Language

Die Sprache BiCEPL spezifiziert das Condition-Action-Model (Kapitel 2.2.3) für Eventklassen. Es definiert das Schema von Subscribed und Complex Event Classes. Mittels BiCEPL werden die Attribute, Lebenszeiten und Condition-Action-Statements von Eventklassen definiert.

In BiCEPL werden erweiterte SQL Select-Statements verwendet, um komplexe Eventklassen zu definieren. Die Erweiterung ermöglicht den temporalen Vergleich von aggregierten Events in Complex Events. Weiters kann auch auf die zeitlichen Informationen (Eintrittszeitpunkt und Wahrnehmungszeitpunkt) von aggregierten Events zugegriffen werden. (Furche et al., 2013b)

Vorweg werden die verschiedenen zeitlichen Prädikate von Ereignissen (Kapitel 2.4.1), die in solchen Systemen auftreten können, beschrieben. Anschließend wird die Syntax (Kapitel 2.4.2) in der Backus-Naur Form (BNF) veranschaulicht und an einem Beispiel angewendet.

2.4.1. Zeitliche Prädikate

BiCEP ermöglicht es, auf Events und deren Modifikationen zu reagieren. Diese Reaktionen auf beispielsweise verschobene Eintrittszeitpunkte werden in BiCEPL festgelegt. Laut Furche et al. (2013b) werden folgende 10 Szenarien für das Auslösen von Aktionen hinsichtlich zeitlicher Differenz gegeben (Abbildung 2.3). Anhand von zwei Kriterien können die Szenarien unterschieden werden: erstens auf der Basis bereits erfasster Events, zweitens auf Basis des Unterschieds zwischen Eintrittszeitpunkt und Wahrnehmungszeitpunkt. Grundsätzlich werden drei Hauptkategorien von Fällen unterschieden, welche in Abbildung 2.3 mit ANNOUNCEMENT, CANCELLATION und CHANGE dargestellt werden.

Wird ein Event zum ersten Mal erfasst, tritt ANNOUNCEMENT ein. Es werden drei Fälle in Hinblick auf die Differenz zwischen Eintrittszeitpunkt und Wahrnehmungszeitpunkt unterschieden. ONTIME, das Event ist pünktlich, z.B. das Event Zugankunft tritt um 12:00 ein und wird genau zu diesem Zeitpunkt erfasst. Das Event ist LATE, also zu spät und ist bereits eingetreten zum Erfassungszeitpunkt, z.B. der Abholer wird nach 12:00 informiert, dass der Zug pünktlich um 12:00 in Linz angekommen ist. Ein Event kann auch in der Zukunft (FUTURE) auftreten, z.B. der Abholer blickt vor 12:00 auf die Ankunftstafel und sieht, dass der Zug um 12:00 ankommen wird.

Wird ein Event storniert (CANCELLATION), unterscheidet man zwischen einer Absage eines zukünftigen und vergangenen Events. FUTURE repräsentiert, dass ein Event, das in der Zukunft stattfinden sollte, nicht mehr auftreten wird, z.B. der Abholer wird vor der Zugankunft

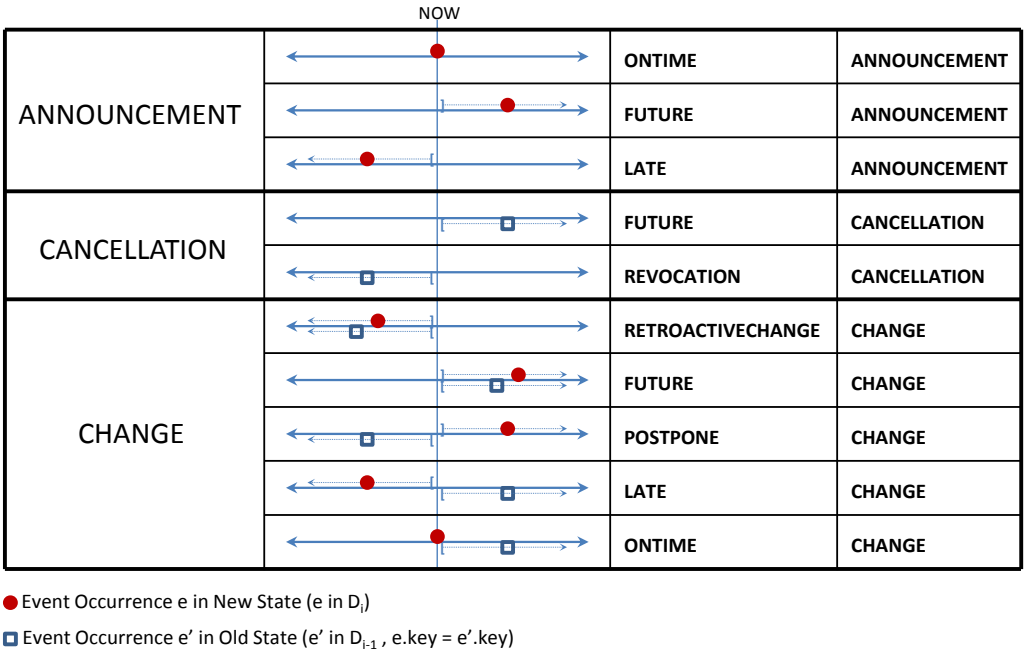

Abbildung 2.3.: Zeitliche Prädikate in BiCEP (Furche et al., 2013b)

informiert, dass der Zug aufgrund eines technischen Gebrechens nicht mehr eintrifft.
REVOCATION bezeichnet den Fall, in welchem ein bereits erfasstes Event aus der Vergangenheit storniert wird, z.B. der Abholer erfährt um 13:00, dass der Zug, welcher um 12:00 ankommen hätte sollen, abgesagt wurde.

Wird ein bereits erfasstes Event modifiziert (CHANGE), unterscheidet man zwischen fünf Fällen: RETROCATIVECHANGE definiert den Fall, dass sich ein vergangenes Event ändert und sein Eintrittszeitpunkt nach wie vor in der Vergangenheit liegt, z.B. es wird angenommen, dass der Zug um 12:00 angekommen ist, jedoch stellt sich um 12:30 heraus, dass er bereits um 11:00 angekommen ist. Liegt jedoch nun sein Eintrittszeitpunkt in der Zukunft statt in der Vergangenheit, tritt der Fall POSTPONE ein, z.B. der Zug wird in Linz um 12:00 erwartet, jedoch stellt sich um 12:30 heraus, dass er wegen einer Verspätung um 13:00 in Linz ankommen wird. Wie bei ANNOUNCEMENT gibt es auch hier die Fälle FUTURE, LATE, ONTIME. Das heißt, für diese Fälle wird nicht unterschieden, ob ein Event bereits vor dem Eintrittszeitpunkt erfasst und möglicherweise modifiziert wurde.

2.4.2. Syntax von BiCEPL

Wie schon erwähnt, ist BiCEPL eine Erweiterung von SQL. Abbildung 2.4 zeigt die Syntax von BiCEPL mittels BNF. Anhand dieser Syntax werden die Events des Anwendungsbeispiels 2.3 definiert.

Ein *program* beinhaltet mehrere Subscribed oder Complex Event Classes, welche mittels *sclass* und *cclass* beschrieben werden.

Eine Subscribed Event Class *sclass* ist änderbar oder nicht, dies wird mit dem Schlüsselwort *mutable* beschrieben. Die Attribute und deren Datentypen werden im *schema* beschrieben. Ein *schema* definiert die Eigenschaften eines Events. Die Datentypen entsprechen den Datentypen des SQL99 Standards. LIFESPAN legt fest, wie lange ein Event nach dem Eintrittszeitpunkt noch existiert. Dieser wird durch einen positiven Integer und in einer Zeiteinheit Tage(d), Stunden(h), Minuten(m) und Sekunden(s) beschrieben.
Die Bedingung, welche das Auslösen eines Events bestimmt, wird über ein Condition-Action-Statement *cond_action* definiert. Innerhalb einer Bedingung kann mit NEW auf den aktuellen und mit OLD auf den zuletzt gültigen Wert eines Eventattributes zugegriffen werden. Weiters wird zu einer Bedingung die auszulösende Aktion bestimmt. Zusätzlich zu den zeitlichen Prädikaten (*predicates*) kann mit FIRED abgefragt werden, ob ein Event bereits eine Aktion ausgelöst hat.

Complex Events Classes *cclass* unterliegen der selben Syntax wie *sclass*. Zusätzlich werden diese noch mit einem SQL-Select Statement (*selection*) erweitert. Dies ermöglicht die Abfrage von Eventattributen und die Referenzierung auf andere Events. Diese SQL Abfrage wird zusätzlich mit einem OCCURRING AT Schlüsselwort erweitert. Mit diesem Schlüsselwort wird die Eintrittszeit des Events definiert. Dazu wird eine Referenz auf ein Attribut angegeben.

In den folgenden Beispielen werden die Eventklassen, die in Kapitel 2.3 beschrieben wurden, in der BICEPL Syntax definiert. Die Attribute der unten angeführten Syntaxbeispiele können dem UML Diagramm (Abbildung 2.2) entnommen werden.

⟨*program*⟩ ::= { ⟨*sclass*⟩ | ⟨*cclass*⟩ }

⟨*sclass*⟩ ::= 'CREATE' ('MUTABLE' | 'IMMUTABLE') 'SUBSCRIBED EVENT CLASS' ⟨*schema*⟩ 'LIFESPAN' ⟨*time_literal*⟩ [⟨*cond_action*⟩ { ',' ⟨*cond_action*⟩ }] ';'

⟨*cclass*⟩ ::= 'CREATE' 'COMPLEX EVENT CLASS' ⟨*schema*⟩ 'LIFESPAN' ⟨*time_literal*⟩ 'AS' ⟨*selection*⟩ [⟨*cond_action*⟩ { ',' ⟨*cond_action*⟩ }] ';'

⟨*schema*⟩ ::= ⟨*name*⟩ '(' ⟨*attributes*⟩ ')' 'ID' '(' ⟨*key*⟩ ')'

⟨*attributes*⟩ ::= ⟨*name*⟩ ⟨*type*⟩ { ', '⟨*name*⟩ ⟨*type*⟩ } ⟨*key*⟩ ::= ⟨*name*⟩ { ',' ⟨*name*⟩ }

⟨*cond_action*⟩ ::= 'ON' ⟨*cond*⟩ 'DO' ⟨*action*⟩

⟨*cond*⟩ ::= ⟨*atom*⟩ | 'NOT' ⟨*cond*⟩ | ⟨*cond*⟩ 'AND' ⟨*cond*⟩ | ⟨*cond*⟩ 'OR' ⟨*cond*⟩

⟨*atom*⟩ ::= ⟨*value*⟩ ⟨*predicate*⟩ ⟨*value*⟩
| 'ANNOUNCEMENT' | 'CANCELLATION' | 'CHANGE'
| 'ONTIME' | 'LATE' | 'LATE(' ⟨*min_delay*⟩ ',' ⟨*max_delay*⟩ ')'
| 'FUTURE' | 'RETROACTIVECHANGE' | 'REVOCATION' | 'POSTPONE'
| 'FIRED'

⟨*value*⟩ ::= ('OLD.' | 'NEW.') ⟨*name*⟩ | ⟨*literal*⟩ | 'NOW'

⟨*action*⟩ ::= ⟨*name*⟩ '(' ⟨*value*⟩ { ',' ⟨*value*⟩ }')'

⟨*selection*⟩ ::= 'SELECT' ⟨*select_clauses*⟩ 'OCCURRING AT' ⟨*time*⟩

⟨*time*⟩ ::= ⟨*table_ref*⟩ '.' ⟨*name*⟩ | ⟨*time*⟩ [('+' | '-') ⟨*time_literal*⟩]
| ('MAX' | 'MIN') '(' ⟨*time*⟩ {',' ⟨*time*⟩ } ')'

⟨*time_literal*⟩ ::= ⟨*integer*⟩ ('s' | 'm' | 'h' | 'd')

Abbildung 2.4.: BiCEPL BNF Definition (Furche et al., 2013b)

Beispiel 6 Die Subscribed Event Class *ZugAnkunft* (Quellcode: 2.1) ist änderbar (mutable), da sich der Zeitpunkt der Zugankunft ändern kann. Als Key wird die *zugNr* und das *datum* definiert. Das Event hat eine Lebenszeit *lifespan* von zwei Tagen.

```
1 CREATE MUTABLE SUBSCRIBED EVENT CLASS
2 ZugAnkunft(zugNr TEXT, datum TEXT, bahnhof TEXT)
3 ID (zugNr, datum)
4 LIFESPAN (2d);
```

Quellcode 2.1: Subscribed Event Class: ZugAnkunft

Beispiel 7 Die *ZugBuchung* (Quellcode 2.2) ist die Subscribed Event Class für die Buchung eines Zugtickets und ist änderbar (mutable). Das Event wird durch die *buchungsNr* eindeutig identifiziert und hat ebenfalls eine Lebenszeit von 2 Tagen.

```
1 CREATE MUTABLE SUBSCRIBED EVENT CLASS
2 ZugBuchung(buchungsNr TEXT,zugNR TEXT, datum TEXT,
     abfahrsbahnhof TEXT,ankunftsbahnhof TEXT)
3 ID (buchungsNr)
4 LIFESPAN (2d);
```

Quellcode 2.2: Subscribed Event Class: ZugBuchung

Beispiel 8 Die Subscribed Event Class Meeting (Quellcode: 2.3) stellt das Treffen dar. Ein Treffen kann sich ändern (*mutable*), beispielsweise kann der Termin geändert werden. Eine *meetingNr* stellt das Schlüsselattribut dar. Ebenfalls wird dieses Event nach 2 Tagen des Eintretens gelöscht.

```
1 CREATE MUTABLE SUBSCRIBED EVENT CLASS
2 Meeting(meetingNr TEXT,buchungsNr TEXT, anlass TEXT,
     personNamen TEXT, braucheAbholung TEXT)
3 ID (meetingNr)
4 LIFESPAN (2d);
```

Quellcode 2.3: Subscribed Event Class: Meeting

Beispiel 9 Die Complex Event Class *KollegenAbholung* (Quellcode: 2.4) findet statt, wenn der Zug ankommt, der Kollege eine Zugbuchung ausgeführt hat und ein Meeting mit einem Abholwunsch existiert. Diese Bedingung wird durch das *statement* mit dem SQL-Statement (Zeilen 4-6) geprüft. Das Event wird ausgelöst, wenn der Zug eintrifft (*OCCURING AT an.occ*) bzw. die Bedingung (statement) erfüllt ist. Mittels *ANNOUNCEMENT* wird beim ersten Eintreten des Events eine SMS Nachricht gesendet.

```
1 CREATE COMPLEX EVENT CLASS KollegenAbholung (zugNr TEXT,datum
     TEXT,buchungsNr TEXT, meetingNr TEXT,ankunftsbahnhof TEXT)
2 ID (zugNr,datum,buchungsNr)
3 LIFESPAN(2d)
4 AS SELECT an.zugNr,an.datum,zb.buchungsNr, me.meetingNr, an.
     ankunftsbahnhof
5 FROM ZugAnkunft an,ZugBuchung zb, Meeting me
6 WHERE an.zugNr=zb.zugNr AND an.datum=zb.datum AND zb.
     buchungsNr=me.buchungsNr AND me.braucheAbholung like 'JA'
7 OCURRING AT an.occ
8 ON ANNOUCEMENT DO SMS('Notiere Termin Kollegenabholdung');
```

Quellcode 2.4: Complex Event Class: KollegenAbholung

Beispiel 10 Die Complex Event Class *StarteKollegenAbholung* (Quellcode: 2.5) hat die gleichen Attribute (Zeile 1) wie *KollegenAbholung*. Dieses Event tritt 30 Minuten vor der *KollegenAbholung* ein (Zeile 5). Der Abholer wird über ein SMS Service verständigt, daraus ergeben sich vier Szenarien:

– ON TIME: Eine rechtzeitige Abholung findet statt (siehe Szenario 2).

- ON LATE: Der Abholer ist weniger als 20 Minuten zu spät (siehe Szenario 3).
- ON LATE: Der Abholer ist mehr als 20 Minuten zu spät (siehe Szenario 4).
- ON RETROACTIVECHANGE: Der Abholer wird über eine Verspätung während der Fahrt zum Bahnhof informiert (siehe Szenario 5).

```
1 CREATE COMPLEX EVENT CLASS StarteKollegenAbholung (zugNr TEXT,
     datum TEXT,buchungsNr TEXT, meetingNr TEXT,ankunftsbahnhof
     TEXT)
2 ID (zugNr,datum,buchungsNr) LIFESPAN(2d)
3 AS SELECT *
4 FROM KollegenAbholung fa
5 OCCURING AT fa-30m
6 ON ONTIME DO SMS('Fahre zum Bahnhof')
7 ON LATE(20m,10h) DO ('Fahre nicht mehr weg, bestelle ein Taxi'
     ),
8 ON LATE(0s,20m) DO SMS('Fahre zum Bahnhof, Benachrichtige
     Kollege'),
9 ON RETROACTIVECHANGE DO SMS('Kollege kommt um %s an',NEW.occ);
```

Quellcode 2.5: Complex Event: StarteKollegenAbhohlung

2.5. BiCEPS Framework

Diese Masterarbeit baut auf einem bestehenden BiCEPS Framework auf, welches in Java implementiert wurde. In Kapitel 2.5.1 wird die Kommunikation zwischen EDT, BiCEP und AEX vorgestellt. Anschließend werden die Zusammenhänge zwischen den verschiedenen Elementen und deren Eigenschaften im BiCEPS Framework beschrieben (Kapitel 2.5.2).

2.5.1. Buffer

Die Interaktion zwischen den Komponenten (EDT, BiCEP, AEX), welche bereits im Kapitel (2.2.1) beschrieben wurden, wird durch jeweilige Buffer realisiert. Ein Buffer dient als Zwischenspeicher für Events und als Bindeglied für die Komponenten. Eine Komponente kann Events in ihrem Buffer ablegen und kann Events von einem Buffer auslesen. Somit können unterschiedliche Komponenten über den Buffer Events ein- und auslesen, es entsteht einen Kommunikation. Diese Kommunikation zwischen den Komponenten wird asynchron durchgeführt. Dadurch werden Wartezeiten, die durch Abhängigkeiten zwischen Komponenten entstehen können, vermieden. Es muss nicht auf die vollständige Abarbeitung der Bearbeitungskette von vorgelagerten Elementen gewartet werden.

2.5.2. Aufbau des Frameworks

Das UML Diagramm in Abbildung 2.5 zeigt das konzeptionelle Modell des BiCEPS. Hiermit werden die notwendigen Elemente und Attribute, die für die Implementierung des BiCEPS Frameworks verwendet wurden, beschrieben.

Das zentrale Stück des Modells ist das *BiCEPS-Model*, welches alle *BiCEPS-Elemente* beinhaltet. Ein *BiCEPS-Element* ist abstrakt und ist Basisklasse für die Komponenten *EDT, BiCEP* und *AEX*, welche einen eindeutigen Namen (*name*) haben. Die Kommunikation zwischen den *BiCEPS-Elementen* wird über Buffer realisiert. Diese Buffer werden in der abstrakten Klasse *BiCEPS-Buffer* definiert, welche als Basisklasse für *EDTBuffer, BiCEPBuffer* und *AEXBuffer* fungiert. Als Attribute werden hier *name* und *purge* vererbt. Letzteres definiert die Zeit, wann ein Event aus dem Buffer gelöscht wird. Diese Buffer (*EDTBuffer*, *AEXBuffer*, *BiCEPBuffer*) enthalten Events. Eine *EventClass* enthält einen Namen (*name*) und eine Lebenszeit (*lifespan*). Die Definition der Event Class wird in BiCEPL Syntax (Kapitel 2.4.2) im Attribut *definition* festgelegt.

Eine Subscribed Event Class wird durch die Klasse *SubscribedEventClass* dargestellt. Eine Complex Event Class wird in der Klasse *ComplexEventClass* beschrieben. Die Action Classes werden in der Implementierung nicht direkt definiert. Es werden spezielle Events zum Anstoßen von Aktionen eingeführt, diese werden in der Klasse *PublishedEventClass* beschrieben. Diese Änderung ist erforderlich, um auch hier eine sequenzielle Abarbeitung in Buffern zu realisieren.

Eine *ComplexEventClass* kann mehrere *EPStatements* (Event Publication Statements) beinhalten. Ein *EPStatement* gehört genau zu einer *ComplexEventClass*. Die Klasse *EPStatement* beinhaltet eine Bedingung (*condition*) und kann mehrere *PublishedEventClasses* definieren. Die *condition* definiert mit Hilfe von zeitlichen Prädikaten (z.B. LATE, ONTIME) die Bedingung, die erfüllt sein muss, damit eine Instanz einer *PublishedEventClass* erzeugt und veröffentlicht wird.

Ein *EDT* beinhaltet das Attribut *interval*. Dieses Zeitintervall wird in Sekunden angegeben und beschreibt das Intervall, in welchem der *EDT* ausgeführt wird. Ein *EDT* kann durch Parameter genauer spezifiziert werden. Ein Parameter wird durch einen Schlüssel (*name*) und einen Wert (*value*) in der Klasse *Parameter* definiert. Ein *EDT* hat genau einen *EDTBuffer*, welcher als Zwischenspeicher für die asynchrone Kommunikation verwendet wird. Ein *EDTBuffer* ist von seinem *EDT* existenzabhängig. Ein *EDT* erzeugt Ereignisse die einer *SubscribedEventClass* zugehörig sind und kann diese Ereignisse an mehrere *BiCEPs* weiterleiten.

Ein *BiCEP* hat das Attribut *chronon*, welches das Intervall der Verarbeitungszyklen definiert. Ein *BiCEP* kann mehrere *Writebuffer* besitzen. Ein *Writebuffer* gehört genau zu einem *BiCEP* und ist ein existenzabhängiger Teil dessen. Ein *BiCEP* beinhaltet mindestens einen *Readbuffer*, wobei dieser von einem oder mehreren *BiCEP*s gelesen wird. *Writebuffer* und *Readbuffer* sind Rollen des *BiCEPBuffer*, welcher für die asynchrone Kommunikation verwendet wird. Ein *BiCEPBuffer* kann Ereignisse unterschiedlicher PublishedEventClasses beinhalten. Die Kommunikation zwischen *BiCEP*s wird durch *Writebuffer*, in welche Events der Klasse *PublishedEventClass* geschrieben werden, und dem *Readbuffern*, aus welchem die Events der Klasse *PublishedEventClass* gelesen werden, realisiert. Ein *BiCEP* definiert mindestens eine *ComplexEventClass*.

Ein *AEX* enthält, wie der *EDT*, eine Menge von Parametern (*Parameter*) und genau einen *AEXBuffer*, welcher für die Kommunikation zwischen *BiCEP* und *AEX* verwendet wird. Ein *AEX*

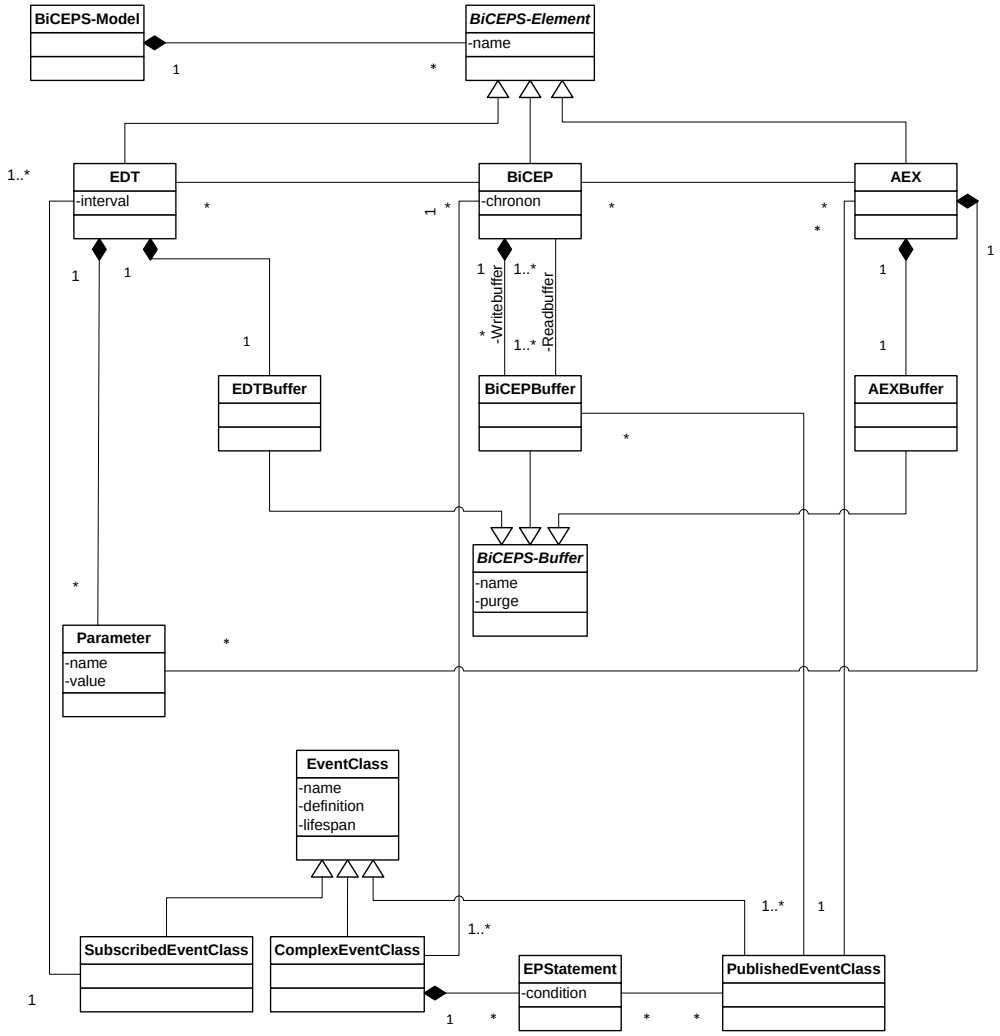

Abbildung 2.5.: Konzeptionelles Modell BiCEPS

verarbeitet Ereignisse genau einer *PublishedEventClass*; hingegen kann eine *PublishedEventClass* von mehreren *AEX* verarbeitet werden.

3. Eclipse Plattform

Ziel dieser Masterarbeit ist es, einen visuellen Editor zur Erstellung eins BiCEPS (Kapitel 2) zu implementieren. Um diesen Editor zu realisieren, wird ein Eclipse Rich Client Plattform (Eclipse RCP) Plug-in entwickelt. Eclipse RPC ist ein mächtiges Rahmenwerk zum Entwickeln von Eclipse Anwendungen und Erweiterungen. Diese Erweiterungen werden Plug-in genannt und können zusammen gekoppelt werden.

Der Editor wird auf Basis bereits bestehender Eclipse Plug-ins aufgebaut, die im Rahmen des Eclipse Project entwickelt wurden. Das Eclipse Project ist jenes Projekt, das für die Entwicklung von Eclipse verantwortlich ist und stellt bereits entwickelte Plug-ins öffentlich zur Verfügung. Dazu wird das bestehende BiCEPS Framework (Kapitel 2.5) mittels Eclipse Modeling Framework (EMF) in ein Datenmodell übertragen. Mit Hilfe dieses Frameworks kann auf Basis des Modells automatisch Java Quellcode generiert werden. Auf diesen generierten Quellcode können bestehende grafische Frameworks (Eclipse Project Tools) zugreifen und gegebenenfalls grafische Funktionen individuell erweitern werden.

Das Kapitel 3.1 gibt einen Überblick des Aufbaus von Eclipse und dessen Inhalt. Weiters werden zwei Teile des Eclipse Projects, die für die Umsetzung relevant sind, detailliert beschrieben. Einerseits wird das EMF (Kapitel 3.2) für das Datenmodell des Editors verwendet, andererseits werden unterschiedliche Frameworks des Tool Project für die visuelle Darstellung vorgestellt (Kapitel 3.3).

3.1. Einführung in Eclipse Projects

Eclipse ist eines der wohl mächtigsten und bekanntesten Entwicklungsumgebungen unserer Zeit. Vor allem für Java Entwickler trifft diese Aussage zu. Neben der großen Community und dem Open Source Gedanken ist der Aufbau der Entwicklungsumgebung eine Besonderheit (Kapitel 3.1.1). Die Entwicklung der Eclipse Plattform teilt sich in vier Projekte auf: Eclipse Project, Model Project, Tools Project und Technology Project. Diese vier Projekte spiegeln die Kernkompentenz der Eclipse Plattform wider.

Das Eclipse Project unterstützt Eclipse Entwickler bei der Erweiterung der Plattform selbst. Dazu stellt es die notwendige Entwicklungsumgebung für die Weiterentwicklung von Eclipse zur Verfügung. Für eine modelgetriebene oder modellunterstützte Entwicklung wurde das Model Project von der Community ins Leben gerufen. Das Tool Project beinhaltet Erweiterungen für Eclipse und inkludiert beispielsweise Anwendungen, um Eclipse als Entwicklungsumgebung für andere Programmiersprachen zu verwenden. Ein großer Teil des Projekts beschäftigt sich mit visuellen Erweiterungen von Eclipse. Das Technology Project ist das „Auffangbecken“ für neue Entwicklungen, die Eclipse für andere Anwendungsbereiche vorbereiten sollen. (Steinberg et al., 2008, S.4-5)

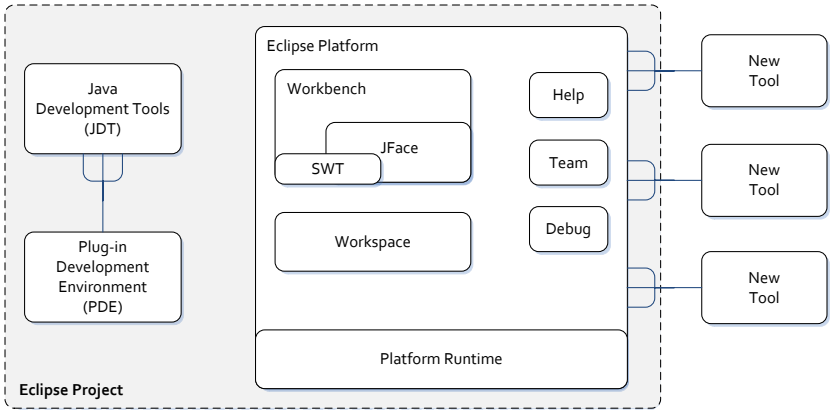

Abbildung 3.1.: Eclipse Architektur (Eclipse Plattform, 2012, S.1)

3.1.1. Aufbau von Eclipse

Eclipse besteht seit Version 3.0 aus einem Java Kern namens Equinox, welcher die sogenannten OSGi[1] -Kernspezifikationen implementiert. Aufbauend auf diesem Kern können verschiedene Erweiterungen (Kapitel 3.1.2) zur Laufzeit gestartet werden. Der OSGi Standard ermöglicht eine standardisierte Erweiterung der Eclipse Plattform (Eclipse Plattform, 2012).

Abbildung 3.1 zeigt den Aufbau der Eclipse Plattform. Eclipse besteht somit aus drei größeren Teilen: Eclipse Plattform, Java Development Tools (JDT) und Plug-in Developer Environment (PDE). Die Plattform beinhaltet die Grundfunktionalität und dient als Koordinationsstelle für Plug-ins. Der Workspace ist jener Speicherplatz, in welchem die Daten eines Projektes in Eclipse abgelegt werden. Auch der Workbench, welcher die Perspektiven, Ansichten und Fenster vorgibt, ist hier beinhaltet. Eclipse besteht weiters noch aus dem JDT und dem PDE, welche für die Entwicklung von Erweiterungen der Plattform vorgesehen sind. Dies sind einerseits JDT, welche eine vollständige Entwicklungsumgebung für Java beschreiben, auf der anderen Seite stellt PDE eine Entwicklungsumgebung für die Implementierung von Erweiterungen den Entwicklern zur Verfügung. (Eclipse Plattform, 2012)

Graphische Komponenten

Der Workbench ist für die grafische Darstellung in Eclipse zuständig, hierbei wird auf Java Frameworks zurückgegriffen. Bei der Entwicklung der Benutzerschnittstelle von Eclipse wird nicht das Standard Windowing Toolkit (Swing) eingesetzt, welches für Java Programme meist verwendet wird, sondern es kann zwischen zwei auf Java basierenden grafischen Benutzerschnittstellen, SWT und JFACE, gewählt werden. Es ist anzumerken, dass diese nur als Überblick in dieser Arbeit erwähnt werden. (Rivieres & Wiegand, 2004, S.340ff)

[1] Open Service Gateway Initiative (OSGi) ist ein Java Konsortium, das sich um die Standardisierung für Java-Entwicklungsumgebungen einsetzt.

Standard Widget Toolkit (SWT) SWT ist eine plattformunabhängige Schnittstelle für die betriebssystemnahe Darstellung von graphischen Elementen. Im Gegensatz zu anderen graphischen API werden native Elemente des Betriebssystems verwendet. Somit ist es möglich, dass Eingabefelder auf einem Linux-Betriebssystem anders aussehen als unter Windows. Dies ermöglicht eine Portierung des Quellcodes von Windows auf Linux und umgekehrt und erleichtert die Bedienbarkeit für Benutzer. (SWT, 2014)

JFace Speziell auf Windows basierenden Plattformen ist bekannt, dass SWT ineffizient arbeitet. Als Abhilfe oder Weiterentwicklung sollte JFace als plattformunabhänige Erweiterung eingesetzt werden. JFace erweitert den Ansatz von SWT. Es erleichtert die Einbindungen von komplexeren Widgets, die aus mehreren SWT-Komponenten bestehen. Weiters wurde eine Abstraktionsschicht, der Viewer, eingezogen, um die Integration und die Kommunikation mit der Datenschicht zu erleichtern. (Eclipse Foundation JFace, 2014)

3.1.2. Einführung Eclipse Plug-in

Eine Erweiterung in Eclipse hinsichtlich Funktionalität wird Plug-in genannt. Die Erweiterungen oder das Zusammenspiel von Erweiterungen werden als kleinste Einheit der Eclipse Plattform definiert. Laut Eclipse Plattform (2012) ist ein Plug-in ein Modul eines Gesamtsystems, welches über definierte Schnittstellen in einer definierten Art und Weise kommuniziert. Diese Plug-ins werden in der Programmiersprache Java realisiert und mittels einer Manifest-Datei beschrieben. Diese Datei spezifiziert die Integration von Java Klassen und die Abhängigkeiten zu anderen Plug-ins. Es können auch Erweiterungspunkte definiert werden, um Funktionen des Plug-ins für andere Plug-ins öffentlich zu machen. Diese Details werden in einer XML Datei (*plugin.xml*) definiert. Diese Architektur ermöglicht das Lazy-Loading in Eclipse, es werden tatsächlich nur die notwendigen Plug-ins beim Starten von Eclipse geladen. Zusätzliche Plug-ins können zur Laufzeit nachgeladen werden (Eclipse Plattform, 2012).

Ein Plug-in besteht aus einem Verzeichnis, welches die Manifest Datei enthält. Dieses Verzeichnis kann auch Ressourcen (Symbole, Icons) und gepackten Java Quellcode als Java Archiv Datei (JAR) enthalten. Diese Struktur muss nicht manuell erstellt werden, es wird mittels PDE ein Assistent zur Erstellung von Eclipse bereitgestellt. Weiters gibt es spezielle Editoren für die Anpassung der Eclipse Manifest Datei (Eclipse Plattform, 2012).

3.2. Eclipse Modeling Project

Das Eclipse Modeling Project, kurz EMF, wurde von der Object Management Group (OMG) ins Leben gerufen und gehört zur Model-Driven-Architecture (MDA) der Eclipse Plattform (Object Management Group, 2014). Grundidee von MDA in Eclipse ist, eine ganzheitliche Lösung für den Entwicklungszyklus einer Software zu schaffen, dessen Fokus auf ein Modell und nicht auf das programmiertechnische Detail gelegt wird (Steinberg et al., 2008). Dieses Modell wird in einem Meta-Modell selbst beschrieben, welches ECore genannt wird. Aus diesem zentralen Modell wird schlussendlich durch Vorlagen, definiert im Generatormodell (GenModel), ein Quellcode erzeugt.

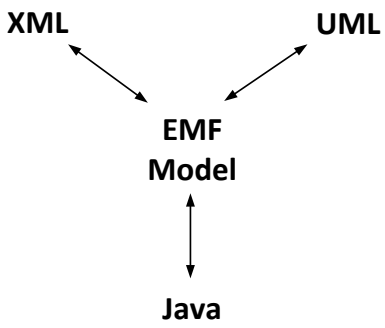

Abbildung 3.2.: Eclipse Modeling Framework (Steinberg et al., 2008, S.14)

Dieser Abschnitt erläutert die Grundlagen von EMF (Kapitel 3.2.1) und zeigt dessen Aufbau. Anschließend wird auf die oben erwähnten Modelle ECore und GenModel und deren Zusammenspiel in Kapitel 3.2.3 und 3.2.4 eingegangen.

3.2.1. Grundlagen

EMF wird auf der Eclipse Foundation Seite (Eclipse EMF Documentation, 2013) mit folgenden Worten definiert:

> The EMF project is a modeling framework and code generation facility for building tools and other applications based on a structured data model. From a model specification described in XMI, EMF provides tools and runtime support to produce a set of Java classes for the model, along with a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor.

Laut dieser Definition dient EMF als Werkzeug und Grundlage für die modellgetriebene Softwareentwicklung. Es stellt die Infrastruktur für die Realisierung solcher „modellzentrierten“ Entwicklungen zur Verfügung. Das Modell selbst wird mit einem Meta-Modell beschrieben und durch den Interchange Standard (XMI) zugänglich gemacht. Mit Hilfe von Mappings, welche durch die Verwendung von XMI möglich sind, kann ein lauffähiger Quellcode erzeugt werden. Somit kann ein reales System auf Basis eines Modells abgebildet werden.

Abbildung 3.2 verdeutlicht die Rolle von EMF und die Vereinigung der drei Technologien Java, UML und XML. EMF steht somit zwischen den Technologien und koordiniert deren Kommunikation (Moore et al., 2004, S.4ff). Beispielsweise kann mittels EMF ein Datenmodell in UML-Notation modelliert werden. Dieses kann anschließend direkt in ein javabasiertes Klassenmodell transformiert werden. XML kann hier für die Persistierung bzw. Datensicherung eingesetzt werden. Die Datenintegrität kann mittels einem XML-Schema (XSD) garantiert werden.

3.2.2. EMF Architektur

Dieses Kapitel gibt einen Überblick der Architektur und des Zusammenspiels der beteiligten Komponenten in EMF (Abbildung 3.3). Das Framework besteht aus zwei getrennten Schichten, diese sind EMF-Runtime und EMF-Tools.

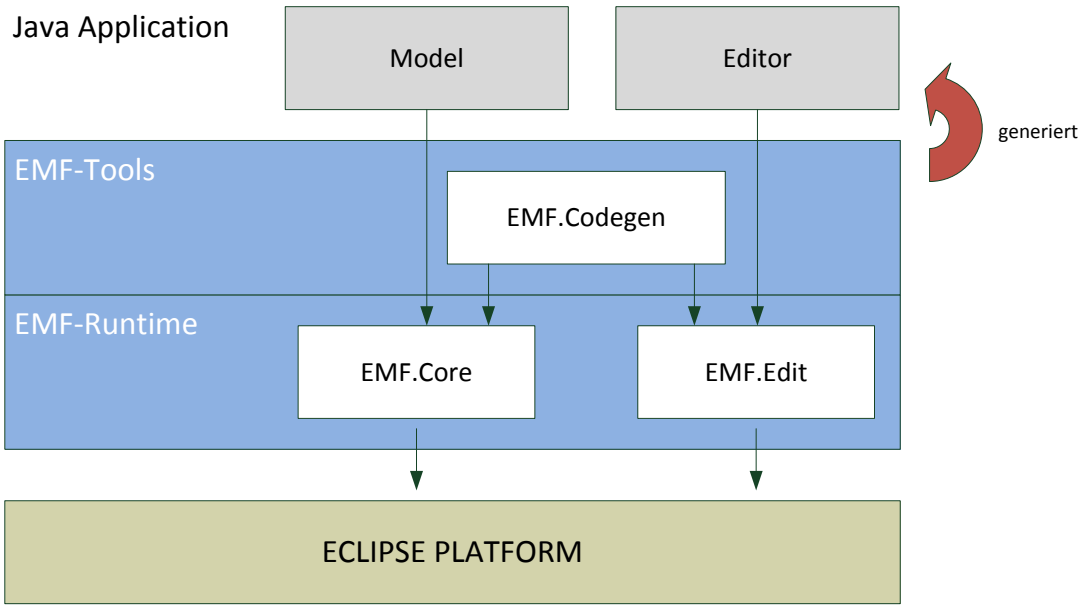

Abbildung 3.3.: EMF Architektur (Steinberg et al., 2008, S.24)

Als Kern des Frameworks gilt die EMF-Runtime, welche die Hauptfunktionen von EMF wider spiegelt. Diese Komponente teilt sich wiederum in zwei Teile auf: EMF.Core und EMF.Edit. Im EMF.Core wird das Meta-Modell von ECore (Kapitel 3.2.3) abgelegt. Damit können die konkreten Modelle beschrieben werden. Support Funktionen für diesen EMF.Core werden mittels dem EMF.Edit ermöglicht. Solche Funktionen sind das Bearbeiten von bestehenden Instanzen, Benachrichtigen von Änderungen bei Abhängigkeiten zwischen Instanzen und die Persistenz der konkreten Modell-Objekte. Die in EMF-Tools beinhaltete Code-Generierungs-Einheit (Kapitel 3.2.4) ist für die Generierung des Quellcodes verantwortlich. Der Generator ist in der Lage, einen vollständigen Editor für die Instanzierung und Bearbeitung des definierten Modells zu erstellen. Dieser mächtige Codegenerator kann über eine Benutzerschnittstelle konfiguriert werden. Die Implementierung des EMF.Codegen baut auf den Java Development Tools (JDT) auf (Abbildung 3.1). (Dave, 2008)

3.2.3. ECore Modell - Datenmodell

Ein EMF Modell wird mittels dem EMF Meta-Modell ECore beschrieben, welches in einer Datei mit der Endung (.ecore) definiert wird. Als Syntax wird der OMG Standard XML Metadata

Interchange (XMI) verwendet. (Steinberg et al., 2008, S.17ff)

In diesem Abschnitt wird zuerst auf die Möglichkeiten der Erstellung solcher Modelle eingegangen. Des weiteren wird das Meta-Modell von Ecore und dessen Elemente beschrieben. Dies wird anhand eines Beispiels verdeutlicht. Es ist anzumerken, dass in der Masterarbeit nur die Grundfunktionalitäten von ECore beschrieben werden.

Modellerstellung

Für die Erstellung solcher Modelle gibt es drei verschiedene Ansätze:

- **Direkte Erstellung:**
  ECore Dateien können manuell über jeden Texteditor definiert werden. EMF enthält einen einfachen baumbasierenden Editor, welcher eine direkte Erstellung ermöglicht. Drittanbieter wie TOP-Cased (Top Cased, Website, 2013) bieten Editoren für ECore Modelle an. Für die Masterarbeit wurde der ECore Tools Graphical Editor verwendet, welcher standardmäßig bei der EMF Installation in Eclipse mitgeliefert wird. (Steinberg et al., 2008)
- **Importfunktionen über Standards:**
  EMF ermöglicht über einen Assistenten den Import bzw. Export über UML. Hierbei kann ein Mapping zwischen den Modellen über einen Editor erstellt werden. Als Standard für den Austausch wird hier Rational Rose® mit der Dateiendung *.mdl* unterstützt. Als zweiter Standard für den Import von bestehenden Modellen wird XML Schema (XSD) unterstützt (Eclipse Modeling Framework, 2013).
- **Java Annotation:**
  Auch die Erstellung aus bestehenden Java-Klassen ist möglich. Ähnlich wie beim UML Ansatz (siehe letzte Aufzählung) wird durch eine Importfunktion eine Modellerstellung ermöglicht. Hierzu müssen jedoch die Interfaces und Klassen in Java mit speziellen Annotationen definiert werden. (Eclipse Modeling Framework, 2013; Steinberg et al., 2008)

ECore Meta-Modell

Das Meta-Modell von ECore definiert die Struktur und die Beziehungen zwischen den Objekten. Es dient dazu, andere Modelle zu beschreiben und zu repräsentieren. Die Datentypen, Beziehungen und deren Struktur sind stark an Java angelehnt (Dave, 2008, S.10).

Die angeführten verwendeten Entitäten gehören zu den Kernelemente von ECore (Abbildung 3.4) und werden nach der Aufzählung noch in den folgenden Paragraphen näher erklärt. Als Quelle dazu wurde (Steinberg et al., 2008, S.105) verwendet.

- EClass repräsentiert eine modellierte Klasse und besteht aus einem Namen, keinem oder mehreren Attributen und keiner oder mehreren Referenzen.
- EAttribute stellt ein Attribut (Eigenschaft) dar und hat einen Namen und einen Datentyp.
- EReference realisiert eine Assoziation zwischen Klassen. Eine Referenz hat einen Namen und einen Zieldatentyp der referenzierten Klasse.

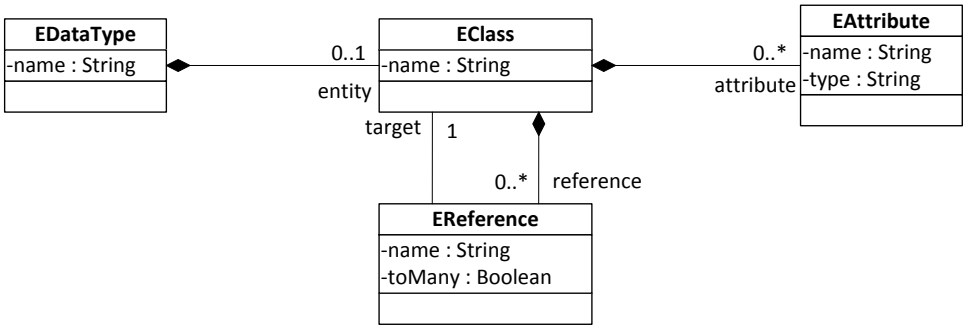

Abbildung 3.4.: ECore Meta-Modell (Steinberg et al., 2008)

– EDataType bezeichnet einen Datentyp eines Attributs.

EClass - Metaklassen Meta-Klassen bestehen aus Attributen, Referenzen und Operationen. Attribute beschreiben die Eigenschaften einer Klasse und können mit einem Namen und einem Datentyp spezifiziert werden. Es kann kein oder mehrere Attribute für eine Klassen definiert werden. Referenzen stellen die Beziehung zu anderen Klassen her und werden im weiteren Verlauf detaillierter beschrieben. Zusätzlich können noch Operationen, die eine Funktionalität einer Klassen beschreiben, hinterlegen werden.

EAttribute - Eigenschaften Ein Attribut bezeichnet eine Eigenschaft einer Klasse. Diese wird mit einem Namen und einem Datentyp festgelegt. Attribute können, müssen aber nicht in einer Klasse enthalten sein. Zusätzlich können noch strukturelle Eigenschaften über folgende Parameter definiert werden (Steinberg et al., 2008, S.107-108).

– Changeable: Das Attribut lässt sich extern setzen.
– Derived: Das Attribut ist aus mehreren Werten zusammengesetzt.
– Transient: Das Attribut wird von der Serialisierung ausgenommen.
– Unique: Das Attribut ist eindeutig.
– Unsettable: Das Attribut kann nicht mit einem Wert initialisiert werden.
– Volatile: Das Attribut besitzt kein Feld, es ist nur durch eine Methode zugreifbar.
– Bounds: Bestimmt die Ober- und Untergrenz (Kardinalität).
– DefaultValue: Standardwert des Attributs.

(Steinberg et al., 2008, S.108)

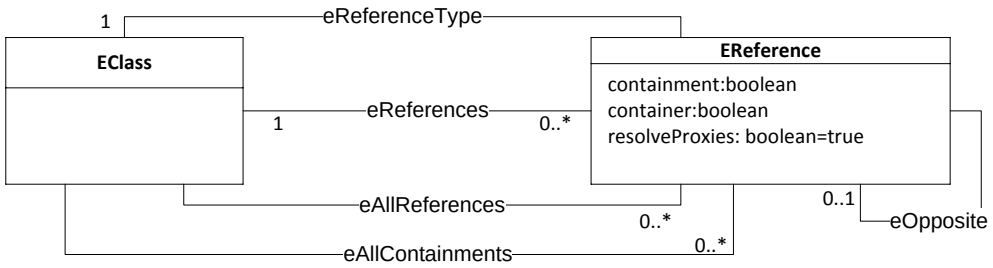

Abbildung 3.5.: Meta-Modell Referenzen (Steinberg et al., 2008, S.111)

EReference - Beziehungen Eine EReference ermöglicht die Abbildung von strukturellen Eigenschaften in EMF Modellen und beschreibt die Beziehung zwischen den Klassen. Die Parametrisierung dieser Referenzen wird anhand der Abbildung 3.5 beschrieben.
Zusätzlich zum Namen und Datentyp der in Beziehung stehenden Klassen können drei boolschen- Variablen *containment*, *container* und *resolveProxies* sowie die Kardinalität der Klassen gesetzt werden. Der *containment*-Flag ist mit einer Kompositionen (A hat B) in UML vergleichbar. Durch das Setzen des *container*-Flags kann die bidirektionale Kommunikation zwischen A und B definiert werden. Hingegen wird der Wert von *containment*-Flag nicht gesetzt, wird eine Assoziation (A kennt B) realisiert. Weiters können mit dem *eOpposite* noch bidirektionale Beziehungen definiert werden. Die Verwendung von *resolveProxies* (Platzhalter) ermöglicht, ein Objekt und seine referenzierten Objekte teilweise zu laden. (Moore et al., 2004, S.23ff)

Die Eigenschaften Container und Containment spielen in EMF eine wichtige Rolle. EMF sieht vor, dass es auf oberster Ebene einen Container gibt, in dem alle Instanzen von *EClass* enthalten sind. Somit geben diese zwei Flags vor, ob eine Instanz in einem anderen Container enthalten sein darf oder ob auf eine Referenz verwiesen wird. (Uwe Kloos, 2012, S.37)

EDatatype - Datenstruktur Ein EDatatype repräsentiert einen Datentyp eines Attributs (EAttribute) oder einer Klasse (EClass) (Moore et al., 2004, S.32ff). Neben den individuellen Klassentypen, welche durch die Definition von Klassen (EClass) entstehen, werden auch standardmäßig die Datentypen in der Tabelle 3.1 im EMF Modell unterstützt.

Zusammenhang der Elemente Ein ECore Modell fasst alle Klassen und Datentypen eines EMF Modells in einer Ressource zusammen, diese werden als ERessource bezeichnet. Über diese Ressourcen können auch mehrere ECore verbunden werden. Eine ERessource beinhaltet einen Mechanismus für die Serialisierung jener Instanzen, die durch die beinhalteten Factories (ERessourceFactories) erstellt werden. Die ERessourcen können mittels URI eindeutig identifiziert und angesprochen werden. (Moore et al., 2004)

ECore Data Type	Java Primitive Type or Class	Serializable
EBoolean	boolean	true
EByte	byte	true
EChar	char	true
EDouble	double	true
EFloat	float	true
EInt	int	true
ELong	long	true
EShort	short	true
EByteArray	byte[]	true
EBooleanObject	java.lang.Boolean	true
EByteObject	java.lang.Byte	true
ECharacterObject	java.lang.Character	true
EDoubleObject	java.lang.Double	true
EFloatObject	java.lang.Float	true
EIntegerObject	java.lang.Integer	true
ELongObject	java.lang.Long	true
EShortObject	java.lang.Short	true
EBigDecimal	java.math.BigDecimal	true
EBigInteger	java.math.BigInteger	true
EDate	java.util.Date	true
EJavaObject	java.lang.Object	false
EJavaClass	java.lang.Class	true
EString	java.lang.String	true

Tabelle 3.1.: EMF Datentypen (Steinberg et al., 2008, S.124)

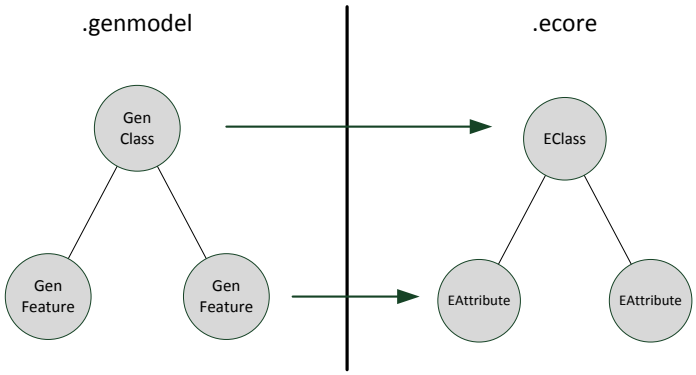

Abbildung 3.6.: Generatormodell (Steinberg et al., 2008, S.29)

3.2.4. GenModel: Generatormodell

Die Codegenerierung ist eines des mächtigsten Werkzeuge, die EMF zur Verfügung stellt. Dieser Generator erstellt auf Basis eines ECore Modells (Kapitel 3.2.3) automatisch einen Java Quellcode. Um diese Erstellung zu ermöglichen, wird ein Generatormodell erstellt, darauf aufbauend wird der Quellcode generiert. Der automatisch generierte Quellcode kann manuell erweitert werden.

In diesem Abschnitt wird zuerst die Erstellung des Generatormodells beschrieben. Anschließend wird die Generierung des Quellcodes verdeutlicht. Abschließend werden die Möglichkeiten der manuellen Erweiterung des generierten Quellcodes erklärt.

Generatormodell

Der Codegenerator erstellt auf Basis der im ECore vordefinierten Klassen, Attribute und Referenzen ein unabhängiges Modell. Dieses Generatormodell mit der Endung *.genmodel* reicht zur Quellcode Erstellung noch nicht aus. Es müssen zusätzliche Informationen wie Präfixe für Dateinamen, Bezeichnungen für Java-Packages und Speicherorte in dieser Datei vermerkt werden. Dieses Generatormodell beinhaltet somit alle ECore Informationen und zusätzliche Informationen, die zur Erstellung des Quellcodes nötig sind. Dieses Modell wird ebenfalls in ECore Notation abgespeichert. Anstatt der *EClass*, *EAttributes* werden die Elemente mit dem Präfix `Gen` (z.B. GenClass, GenAttribute) ersetzt. Das Generatormodell dient somit als Wrapper für das jeweilige ECore Modell. Der Vorteil der strikten Trennung beider Modelle ist, dass es verschiedene Generatoren für dasselbe ECore Modell geben kann, jedoch werden alle Informationen redundant gespeichert. Um hier einen inkonsistenten Zustand zu vermeiden, übernimmt EMF die Synchronisation selbst (Dave, 2008; Eclipse Modeling Framework, 2013).

Quellcode-Generierung

Nach der Erstellung eines Generatormodells (.genmodel) kann der Java Quellcode automatisch auf Knopfdruck erstellt werden. Hierbei wird für jede *EClass* ein eigenes Interface (Quellcode: 3.1) angelegt. Dieses Interface wird durch eine Implementierungsklasse (Quellcode 3.2) mit dem Postfix *Impl* implementiert . Diese Implementierungsklassen werden durch vordefinierte Templates erweitert. Diese Templates beinhalteten die notwendige Logik, um folgende Mechanismen zu garantieren. Es wird ermöglicht, zur Laufzeit dynamische Klassen zu laden (Reflection). Es wird mittels Notifications auf Änderungen von gekoppelten oder referenzierten Objekten reagiert. Die Datensicherung wird über Persistence-Methoden ermöglicht. Somit können diese Objekte serialisiert gespeichert und wieder ausgelesen werden.

Dieser Vorgang kann auch durch das Definieren des *Instance Type Name*-Attributes vermieden werden. Dieses Attribut legt fest, ob ein bereits existierendes externes Java Interface verwendet und keine Java Klasse für die EClass generiert wird. Das bestehende Interface wird somit implementiert. (Steinberg et al., 2008, S.243)

Zusätzlich zu den definierten EClasses werden noch RessourceFactories erstellt, die die Instanzierung der Java Klassen ermöglichen.

```
1 package biceps.interfaces
2 public interface IEDT...
```

Quellcode 3.1: Generiertes-Interface

```
1 package biceps.impl
2 public
3   class EDTImpl implements IEDT...
```

Quellcode 3.2: Generierte Implementation-Klasse

Quellcode-Erweiterung

Wird dieser automatisch erstellte Quellcode händisch erweitert, stellt der Codegenerator eine Merge Funktion zur Verfügung. Somit wird der manuell hinzugefügte Quellcode nicht beim erneuten Generieren des Quellcodes überschrieben (Steinberg et al., 2008, S.247). Bei der manuell erweiterten Codestelle (Methodenkopf, Variabledeklaration) muss die Annotation `@generated` gelöscht werden (siehe Quellcode 3.3). Bei Konflikten wird die manuell erweiterte Stelle behalten und der generierte Code mit dem Postfix `Gen` erweitert.

```
1 //custom method not generated
2 public int CustomToString() ...
3
4 //generated method
5
6 /** * @generated */
7 public String customToStringGen()  ...
```

Quellcode 3.3: CodeGeneration-Erweiterung

Im Quellcode 3.3 wird in der *customToString* Methode der Rückgabewert von String auf Integer geändert. Durch die Entfernung der Annotation wird bei der nächsten Codegenerierung diese Methoden nicht überschrieben. Wird eine Änderung am Methodenrumpf (Rückgabewert geändert) vorgenommen, kann ein Konflikt auftreten. In diesem Fall erstellt der Generator sicherheitshalber dieselbe Methode mit dem Namen *customToStringGen*.

Der Generator generiert leichtgewichteten Quellcode. Laut Steinberg et al. (2008) ähnelt dieser dem manuell erstellten Quellcode und ist in jedem Fall korrekt.

3.2.5. Serialisierung und Persistenz

Um die erzeugten Instanzen des ECore langfristig zu sichern und die Datensicherung zu gewährleisten, unterstützt EMF auch die Serialisierung von Objekten. Es werden zwei verschiedene Mechanismen implementiert: XML und XMI.

```
1  xsi:type="ecore:EClass"name="PurchaseOrder">
2
3    <eStructuralFeatures xsi:type="ecore:EReference"
4    name="items" eType="\#//Item"
5    upperBound="-1" containment="true"/>
6
7    <eStructuralFeatures xsi:type="ecore:EAttribute
8    "name="shipTo" eType="ecore:EDataType
9    http:...Ecore\#//EString"/> ...
10 </eClassifiers>
```

Quellcode 3.4: Beispiel Serialisierung XMI

Diese Implementierungen können beim Generieren der Modelle definiert werden. Die Durchführung der Speicherung wird über die EResources, welche alle Elemente eines Modells beinhalten, realisiert. Quellcodeauszug 3.4 zeigt die Darstellung eines serialisierten Objekts (PurchaseOrder) in der XMI Notation.

Die Eclipse Plattform bietet auch noch weitere Implementationen für die Erweiterungen der Persistenz von EMF Modellen an. Unter anderem ermöglicht Teneo die Speicherung eines Modells in einer relationalen Datenbank (Teneo, 2013).

3.3. Eclipse Tool Project

Neben dem Modeling Project (Kapitel 3.2) existiert noch das Tool Projekt, das Eclipse eine Reihe von Erweiterungen zusätzlicher Funktionen bereitstellt. Neben der Unterstützung anderer Programmiersprachen in Eclipse (C/C++, PHP, COBOLD) unterstützen diese Erweiterungen auch grafische Komponenten, die Entwickler bei der Implementierung visueller Editoren benötigen.

Dieser Abschnitt beschäftigt sich hauptsächlich mit den grafischen Frameworks, die im Tool Project enthalten sind. Es werden drei Frameworks, die eine Implementierung eines Editors unterstützen, vorgestellt. Diese sind: Graphical Editing Framework (Kapitel 3.3.1), Graphical

Modeling Framework (Kapitel 3.3.2) und Graphiti (Kapitel 3.3.3).

3.3.1. Graphical Editing Framework

Graphical Editing Framework (GEF) ermöglicht eine schnelle Implementierung eines grafischen Editors, welcher auf ein bestehendes Modell aufbaut. GEF ist Teil der Project Tools und somit in Eclipse integriert. Mit Hilfe des Draw2D Frameworks wird die Visualisierung des Modells ermöglicht. Draw2D ist ebenfalls in Eclipse integriert und gehört zu den SWT Standards. Dieses Framework ermöglicht die visuelle Darstellung von Elementen. Durch GEF ist es möglich, bestehende Datenmodelle und deren Instanzen über eine Benutzerschnittstelle zu modifizieren. Es können Eigenschaften der dargestellten Elemente verändert werden, auch Operationen können auf diesen Elementen ausgeführt werden. Auch standardisierte Eigenschaften von grafischen Editoren wie Drag and Drop, Copy and Past, Context Menu und Toolbars werden durch das Framework ermöglicht. (Moore et al., 2004, S.86-93)

Im Gegensatz zu individuellen Java Anwendungen, die solche grafischen Editoren mittels SWT realisieren, können GEF Editoren durch die Einhaltung von Standards leichter erweitert und gewartet werden.

Draw2D

Draw2D ermöglicht die Darstellung und Zeichnung von grafischen Elementen. Die Kernelemente des Draw2D Frameworks nennen sich Figures, die grafische Repräsentanten sind. Figures können wiederum aus mehreren Figures bestehen. Ein Figure repräsentiert eine beliebige geometrische Form. Um die Positionierung der kaskadierten Figures kümmert sich der LayoutManger (Moore et al., 2004, S.93-95).

Das Konzept von Draw2D basiert auf einem sogenannten LightWeightSystem (LWS), welches in der Abbildung 3.7 dargestellt wird. Dieses LWS dient als Verbindungsstück zwischen SWT und den Figures. Somit wird die schwergewichtete SWT-Komponente von dem Draw2D abgrenzt und dient sozusagen als Vermittler zwischen Figures und SWT. Dieses LWS besteht aus einer Root-Figure, welche als Container für alle Figures dient. Es beinhaltet ein SWT Canvas, in welchem Hintergrund und Schriftarten abgelegt sind. Ein weiteres Element des LWS ist der EventDispatcher. Dessen Aufgabe ist es, die SWT Events aufzufangen und auf die entsprechenden Figures weiterzuleiten. Der Dispatcher hört die Benutzereingaben ab, beispielsweise reagiert er auf Focuswechsel von Elementen, Tooltips und Mauseingaben. Der UpdateManager ist hinsichtlich der Darstellung der Hauptakteur. Dieser koordiniert das Zeichnen und die Kommunikation zwischen SWT und den zu zeichnenden Figures. (Moore et al., 2004)

Abbildung 3.8 zeigt das Zusammenspiel und den zeitlichen Verlauf der oben beschriebenen Elemente in Draw2D detailliert. Der Ablauf wird durch eine User-Interaction (1) angestoßen, dies kann beispielsweise eine Drag & Drop Aktionen auf ein Element sein. Jede Veränderung von GUI Elementen wird über einen Listener (2) wahrgenommen. Das LWS leitet dieses Event an den jeweiligen SWT Event Dispatcher (3) weiter. Durch diese Entkoppelung müssen nicht alle SWT Elemente informiert werden, es wird nur das betroffene Element vom SWT Event Dispatcher über die Änderung (4) benachrichtigt. Diese Figure benachrichtigt den Update Manager und fordert einen Neuzeichnung an (5a). Gegebenenfalls benachrichtigt der Updatemanager

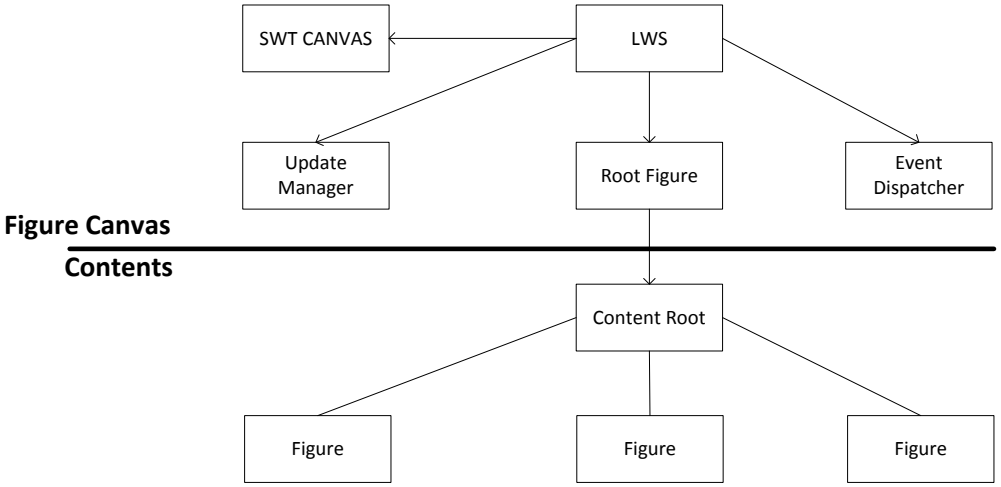

Abbildung 3.7.: Draw2D Architektur (Gronback, 2009, S.319)

die Containerfigures über eine Neuzeichnung (5b) und informiert das UI über die erfolgreiche Neuzeichnung (6). (Gronback, 2009)

Draw2D ist für die Darstellung in GEF verantwortlich, es kann jedoch auch als Standalone Version genutzt werden. Für einen Quellcodeauszug wird auf (Moore et al., 2004, S.94) verwiesen.

Architektur von GEF

Die Darstellung und Zeichnung von Objekten wird in GEF von Draw2D übernommen. Draw2D verwendet dazu SWT Komponenten. Diese Darstellungsobjekte heißen Figures und entsprechen einer grafischen Darstellung, wie beispielsweise ein Rechteck mit einem Text, das keine Logik beinhaltet. GEF ermöglicht die Bearbeitung von Instanzen des Datenmodells über grafische Elemente, welche über Draw2D visuell dargestellt werden. GEF hält sich strikt an das MVC Paradigma und trennt Model und View Komponenten (Gronback, 2009, S.325ff).

Um Änderungen an Objekten des Datenmodells über einen Benutzerschnittstelle vorzunehmen benötigt GEF ein eigenes Modell (Abbildung 3.9). Aufgabe des Modell ist es, die Kommunikation zwischen Figures und dem textuell beschriebenen Datenmodell herzustellen. Somit wird eine Interaktion zwischen Benutzereingaben und dem darunter liegenden Datenmodell, welches die Logik und die Informationen enthält, ermöglicht. Das GEF Modell besteht aus sogenannten EditParts, welche das Datenobjekt wie auch den visuellen Repräsentanten (Figure) enthalten. Dabei entspricht das GEF Modell derselben Hierarchie wie das Datenmodell. GEF unterscheidet zwischen zwei Typen von EditParts, eines für die Verbindungen der Elemente und eines für die Elemente an sich. Die Verwaltung und Erstellung dieser EditParts wird durch die EditPart-Factory übernommen. (Bilnoski, 2010)

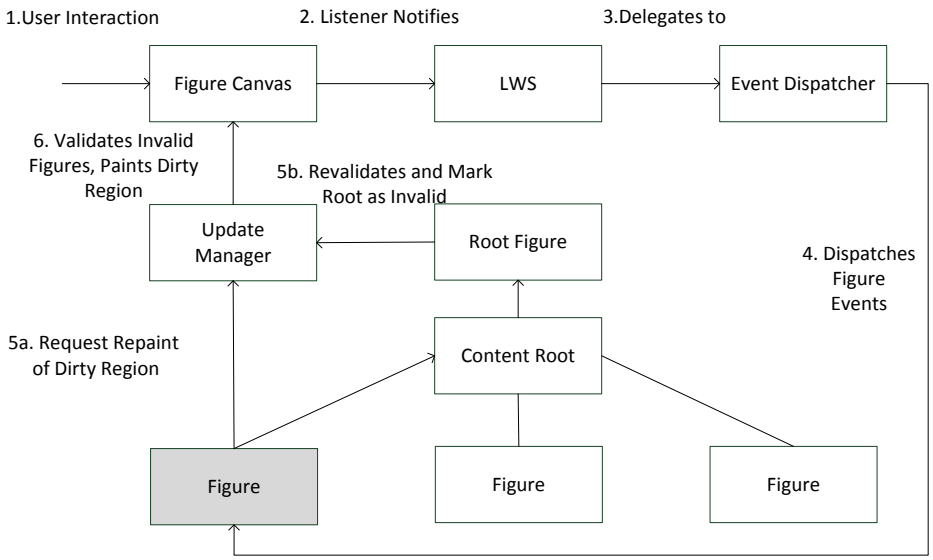

Abbildung 3.8.: Draw2D Figures (Gronback, 2009, S.320)

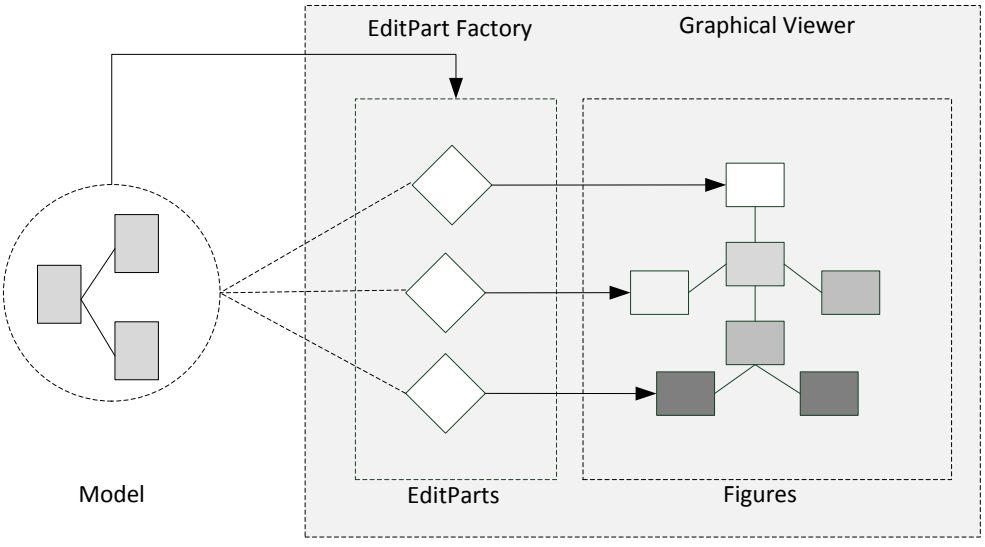

Abbildung 3.9.: GEF Modell (Gronback, 2009, S.327)

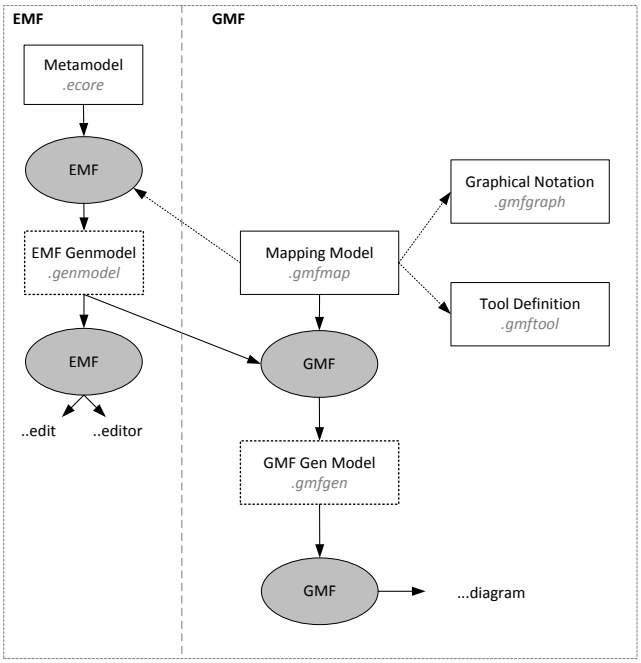

Abbildung 3.10.: GMF Workflow (GMF, 2014, S.8)

3.3.2. GMF

Graphical Modeling Framework (GMF) ist ein weiteres Framework, das eine schnelle standardisierte Implementierung eines grafischen Editors ermöglicht. Dabei greift GMF auf zwei andere Frameworks zurück, diese sind GEF und EMF. Wie schon im Kapitel 3.3.1 beschrieben, wird GEF für die Erstellung grafischer Elemente im Eclipse Workbench eingesetzt. EMF (Kapitel 3.2) hingegen wird für die Erstellung und Verwaltung von Datenmodellen verwendet. GMF kombiniert beide Ansätze und schafft auf Basis eines bestehenden EMF Modells eine grafische Schnittstelle, mit welcher dieses Modell modifiziert werden kann (GMF, 2014).

Architektur

GMF vereint GEF und EMF und erzeugt aus dem bestehenden EMF Modell mit Hilfe von GEF einen grafischen Editor. Dieser ermöglicht die Modifizierung des Modells über eine visuelle Schnittstelle (Benutzerschnittstelle). Bei dieser Erstellung werden weitere Zwischenmodelle generiert, es gibt keinen direkten Weg von EMF nach GEF (Gronback, 2009, S.60).

GMF stellt Generatoren zur Verfügung, um aus dem bestehenden Modell ein Zwischenmodell zu generieren, dieser Workflow und die zu benötigten Artefakten werden in der Abbildung 3.10 verdeutlicht. Als erster Schritt muss aus dem vorgelegten EMF Modell ein Generator Modell (siehe Kapitel 3.2.4) erstellt werden. Als Resultat wird javabasierter Quellcode erzeugt. Diese beiden

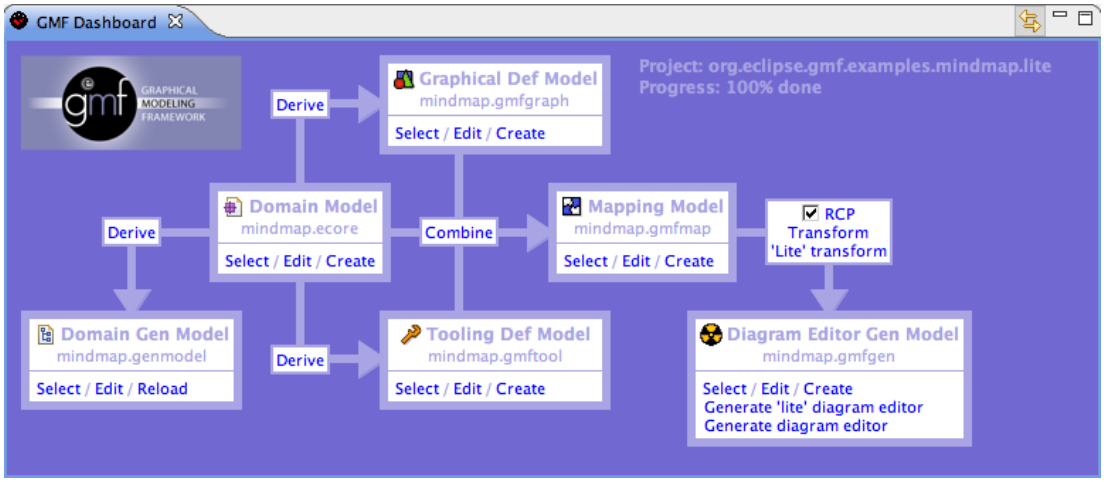

Abbildung 3.11.: GMF Dashboard (GMF, 2014, S.1)

Modelle dienen als Grundlage für das Graphical-Notation-Model. Dieses Graphical-Notation-Model beschreibt die Figuren (siehe Kapitel 3.3.1-Figures), die im Diagramm Editor zu sehen sein werden. Weiters werden neben den rein grafischen Elementen auch die Verbindungselemente definiert. (Gronback, 2009, S.61-62)

Das Tool-Definition-Model beschreibt Informationen über die Steuerelemente (ToolTips, Icons), welche beispielsweise in einer Palette angezeigt werden. Hier werden also die Werkzeuge, die ein Editor enthält, beschrieben (Gronback, 2009, S.63).

Diese beiden Modelle (Graphical-Notation-Model, Tool-Definition-Model) werden in Dateien mit den Endungen *.gmfgraph* und *.gmftool* abgelegt. Das Mapping Model hat die Aufgabe, beide Modelle zu verknüpfen. Somit spezifiziert dieser Mapper die Beziehung zwischen den grafischen Modellen (.gmffraph,. gmftool) und dem Datenmodell (ECore). Als Resultat wird das GMF Modell erzeugt (Gronback, 2009, S.63). Ähnlich dem Vorgang wie bei ECore wird aus diesem GMF Modell ein Generatormodell erzeugt, welches den Quellcode für den Editor erzeugt. Dieser strikt vorgegebenen Vorgang zur Erstellung eines Editors wird von GMF mittels dem GMF Dashboard (Abbildung 3.11) erleichtert, welches die Zwischenmodelle generiert. (Gronback, 2009, S.66ff)

3.3.3. Graphiti

Die Entwicklung eines graphischen Editors auf Basis eines Datenmodells mit den bisherigen Frameworks kann schnell sehr komplex werden. Einen alternativen Ansatz stellte 2010 die SAP AG mit ihrem *Framework Graphical Tooling Infrastructure* vor und stellt es der Eclipse Community frei zur Verfügung (Brand et al., 2011).

Der Grundgedanke von Graphiti ist, die Komplexität von Draw2D und GEF hinter einer Java API zu verstecken. Somit kann die Einarbeitungszeit in Draw2D dem Entwickler erspart werden. Im Gegensatz zu GMF (Kapitel 3.3.2) arbeitet Graphiti mit Java Schnittstellen. Hingegen entsteht bei GMF ein Editor aus einem Modell, das wiederum aus einem Datenmodell erzeugt

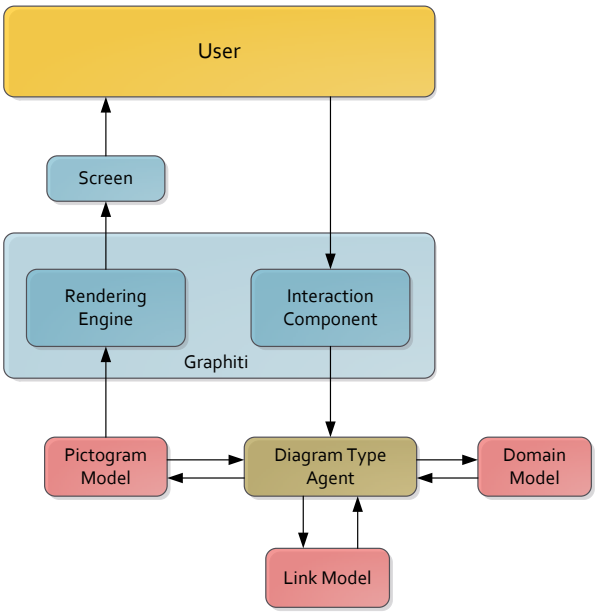

Abbildung 3.12.: Graphiti Architektur (Brand et al., 2011, S.4)

wird. Die Funktionalität des Editors wird in Graphiti über ein Featurekonzept dargestellt, welches im weiteren Verlauf noch detaillierter beschrieben wird. (Brand et al., 2011)

Architektur

Das Graphiti Framework baut ebenso, wie die anderen bereits vorgestellten Frameworks, auf dem MVC-Paradigma auf, dies veranschaulicht die Abbildung 3.12. Die Interaktion mit dem Benutzer wird in Richtung des Frameworks durch die Interaction Component ermöglicht. Deren Aufgabe ist es, Benutzereingaben über Maus oder Tastatur an den Diagram Type Agent weiterzuleiten. Die Darstellung, die für den Benutzer ersichtlich ist, wird über die Rendering Engine ermöglicht. Die Komponenten Interaction Component und die Rendering Engine basieren auf dem GEF Framework und bilden die Graphiti Laufzeitumgebung ab (Brand et al., 2011; Wenz, 2012).

Im Gegensatz zu den beiden Komponenten (Interaction Component, Rendering Engine) müssen die darunter liegenden Graphiti-Komponenten vom Entwickler selbst definiert werden. Der Diagram Type Agent ist das Kernstück und dient als Schnittstelle zwischen Pictogram Model, Domain Model und Link Model. Das Domain Model beschreibt die Daten, welche durch den Editor visualisiert werden sollen. Dies wird meist durch ein EMF Modell realisiert, jedoch unterstützt Graphiti Plain Old Java Object (POJO) Modell ebenfalls. Das Gegenstück zum Domain Model nennt sich Pictogram Model. Es beinhaltet alle Informationen über die Darstellung der einzelnen grafischen Elemente. Dadurch, dass dieselben Daten im Domain Model als auch im Pictogram Model abgelegt sind, wird durch Update Funktionen eine automatische Synchronisation von Graphitit bereitgestellt. Um diese beiden Modelle nun zu verbinden, dient das Link Model.

Somit wird die grafische Darstellung und das domainspezifische Modell verknüpft. (Graphiti, 2014)

Featurekonzept

Wie schon in der Einleitung von Graphiti erwähnt, basiert dieses Framework auf dem Featurekonzept. Die Funktionalitäten, welche vom Editor bereitgestellt werden, werden in sogenannten Features beschrieben, welche im Feature Provider festgelegt werden. Dieser ist wiederum im Diagram Type Agent (Abbildung 3.12) registriert.

Die Erstellung solcher Features ist vom Entwickler frei wählbar. Die Auswahl, auf welches Element eine Feature angewandt wird, wird über den Kontext entschieden. Dieser wird vom Interaction-Component geliefert, z.B.: Auswahl eines Elements im Zeichenfeld (Brand et al., 2011).

Graphiti unterscheidet folgende Basisfunktionalitäten, die vom Entwickler beliebig erweitert werden können.

- Create Feature: Erstellung eines neues Objekts im Domain Modell.
- Add Feature: Erstellung eines Pictogram-Objekts.
- Delete Feature : Löschung eines Domain-Objekts als auch eines Pictogram-Objekts.
- Layout Feature: Spezifiziert Verhalten, falls ein Pictogram-Objekt vergrößtert wird.
- Custom Feature: Erstellt einen Eintrag im Kontextmenü eines Pictorgram-Objekts.

Teil II.

Design und Implementierung

4. Anforderungen

In diesem Kapitel werden die Implementierungsanforderungen an den visuellen Editor, die durch die Aufgabenstellung festgelegt und Basis für die Umsetzung sind, beschrieben. Laut Balzert (2009, S.456) werden in der Regel Anforderungen eines Softwaresystems in nicht-funktionale und funktionale Anforderungen unterteilt. Funktionale Anforderungen legen fest, welche Funktionen ein Softwaresystem leisten muss. Hingegen beschreiben die nicht-funktionalen Anforderungen die qualitative Sicht auf die Funktionen, sie beschreiben somit, wie ein System funktionieren muss (Balzert, 2009, S.456ff). Dieses Referenzmodell wird für die Definition der Anforderungen an den visuellen Editor verwendet.

Im Kapitel 4.1 werden die funktionalen Anforderungen beschrieben, die nicht-funktionalen Anforderungen des Editors werden im Kapitel 4.2 angeführt. Abschließend wird ein Prototyp für die Benutzerschnittstelle (Kapitel 4.3) vorgestellt.

4.1. Funktionale Anforderungen

In diesem Abschnitt werden alle funktionalen Anforderungen vorgestellt, welche an die Implementierung eines visuellen Editors für BiCEPS gestellt werden. Unter funktionalen Anforderungen versteht man jene Aspekte, die ein System aus funktioneller Sicht leisten muss (Balzert, 2009). Es werden somit die Funktionen beschrieben, welche der visuelle Editor beinhaltet.

4.1.1. Visualisierung der BiCEPS Komponenten

Es wird eine visuelle Benutzerschnittstelle für ein BiCEPS erstellt. Dazu müssen die Komponenten (EDT, AEX, BiCEP) grafisch dargestellt werden. Diese Anforderung wird in Standardfunktionen, Einfügen von Komponenten, Verbinden von Komponenten, Anpassung der Komponenten und Sicherung der modellierten Komponenten unterteilt.

Standardfunktionen

Der visuelle Editor soll die standardisierten Funktionen eines visuellen Editors wie Drag and Drop von Komponenten, das Verbinden von Komponenten, Zoom-Funktion der Zeichenfläche, Tooltips für Komponenten und eine Palette zur Erstellung von Komponenten bereitstellen.

Einfügen von Komponenten

Das Einfügen von neuen Komponenten in die Zeichenfläche muss ermöglicht werden und soll über eine Palette, welche alle verfügbaren Komponenten des BiCEPS beinhaltet, realisiert werden. Die Darstellung der Komponenten (EDT, BiCEP, AEX) muss einfach zu unterscheiden sein und wird im Kapitel 4.3 näher beschrieben.

Verbindung der Komponenten

Die Verbindung der Komponenten ist über Linien zu realisieren, welche die jeweiligen Buffer (EDTBuffer, AEXBuffer, BiCEPBuffer) repräsentieren. Aufbauend auf dem konzeptionellen BiCEPS Framework Modell (Abbildung 2.5) sind die vorgegebenen Restriktionen zwischen Komponenten und Buffern einzuhalten. Daraus ergeben sich folgende Fälle für Verbindungen zwischen den Elementen. Es ist anzumerken, dass diese gerichteten Verbindungen entsprechen.

- Ein EDT kann ausschließlich mit BiCEPs verbunden werden.
- Ein BiCEP kann mit einem AEXs oder BiCEPs verbunden werden.
- Ein AEX kann mit keinem nachfolgenden Element verbunden werden.

Editierbarkeit der Eigenschaften von Komponenten

Die eingefügten Elemente (Komponenten, Linien) müssen parametrisierbar sein. Das heißt, die Eigenschaften der Komponenten (EDT, BiCEP und AEX), welche bereits in Abbildung 2.5 beschrieben wurden, müssen auch über die Benutzerschnittstelle editierbar sein. Zusätzlich zu den Komponenten müssen auch die Eventklassen (SubscribedEventClass, ComplexEventClass und PublishedEventClass) spezifiziert werden.

Eventfluss

Im visuellen Editor muss auch der Eventfluss zwischen den Komponenten ersichtlich sein. Es müssen zusätzlich zu den Komponenten und deren Verbindungen auch die jeweiligen Eventnamen im Diagramm ersichtlich sein. Dazu soll der Eventname über die Linie, welche zwei Komponenten verbindet, gesetzt werden. Existieren mehrere Events (z.B. zwischen zwei BiCEPs), sind die Eventnamen mit einem Beistrich zu trennen und oberhalb der Linie zu platzieren.

4.1.2. Syntaxprüfung von BiCEPL

Es muss im visuellen Editor dem Benutzer ermöglicht werden die in BiCEPL definierten Events einer Syntaxprüfung zu unterziehen. Diese Validierung ist mittels externem Parser, welcher vom BiCEPS Framework zur Verfügung gestellt wird, durchzuführen. Dabei ist die Definition eines Events zu überprüfen und gegebenenfalls ist eine Syntax-Fehlermeldung auszugeben.

4.1.3. Fehlerbehandlung

Wie bereits in den Kapiteln 4.1.1 und 4.1.2 erwähnt, müssen die eingegebenen Daten validiert werden. Es ist zwischen der Validierung des BiCEPS Frameworks (Kapitel 2.5.2) und Eventdefinitionen in BiCEPL (Kapitel 2.4.2) zu unterscheiden. Die Verbindung zwischen Komponenten (Kapitel 4.1.1) sind anhand des konzeptionellen Modells des BiCEPS zum Zeitpunkt ihrer Erstellung zu validieren. Eine Verbindung zwischen nicht kompatiblen Komponenten ist bereits zum Erstellungszeitpunkt zu verhindern. Die Validierung der Eventdefinitionen (Kapitel 4.1.2) muss spätestens während der Übersetzungszeit vom visuellen Modell zum lauffähigen BiCEPS passieren. Diese Vorgang muss vom Benutzer bewusst (z.B. Validierungsbutton) gestartet werden.

4.1.4. Konfiguration der Komponenten

Die Elemente des visuellen Editors, welche die Komponenten (EDT, BiCEP und AEX) widerspiegeln, müssen von außen parametrisierbar sein. Dies bedeutet, die Eigenschaften (Name, Parameter) der Komponenten können von außen modifiziert werden. Zusätzlich muss auch die Logik, welche die Funktionalität einer Komponente beschreibt, austauschbar sein. Somit wird ermöglicht, dass die Elemente, welche in der Palette angezeigt werden, verändert werden können, ohne den Quellcode des Editors zu verändern. Die Änderungen müssen von außen und ohne Expertenwissen möglich sein.

4.1.5. Sicherung der Zeichenfläche

Die Zeichenfläche, welche die modellierten Komponenten und ihre Beziehung zueinander enthält, muss serialisierbar sein. Hier ist sicherzustellen, dass diese Daten nicht nur für den visuellen Editor lesbar sind, sondern die Elemente und deren Eigenschaften auch für andere Anwendungen verwendet werden können.

4.1.6. Ausführbarkeit des BiCEPS

Das Modell, welches im visuellen Editor erstellt wird, muss ausführbar sein. Dazu muss dieses visuelle Modell so konvertiert werden, dass die Eigenschaften und Beziehungen der Elemente in das BiCEPS Framework (Kapitel 2.5) übernommen werden können. Dort können diese dann ausgeführt werden.

4.2. Nicht-funktionale Anforderungen

Die nicht-funktionalen Anforderungen sind jene, unter welchen Umständen die Funktionalitäten des visuellen Editors zu erbringen sind. Diese Anforderungen beschreiben die qualitativen Eigenschaften, welche an Softwaresystem gestellt werden. Somit wird beschrieben, wie die funktionalen Anforderungen umgesetzt werden.(Balzert, 2009, S.463)

4.2.1. Implementierungsprache Java

Das bereits bestehende Framework (Kapitel 2.5), auf welchem der Editor aufbaut, wurde in Java implementiert. Dies hat zur Folge, dass auch der Editor auf Java basieren soll.

4.2.2. Einsatz von Eclipse

Bei der Erstellung des Editors soll auf bestehende Frameworks zurückgegriffen werden. Die Wiederverwendbarkeit und die Erweiterbarkeit spielen dabei eine große Rolle. Der Einsatz von Eclipse ist weit verbreitet und somit ist der Umgang mit dem Tool vielen Anwendern bekannt. Um unnötige Einarbeitungszeit durch eine neue unbekannte Umgebung für den Benutzer zu vermeiden, soll der Editor als Erweiterung (Plug-in) von Eclipse implementiert werden.

4.2.3. Standards für Datenaustausch

Die Informationen, welche mit dem visuellen Editor erstellt werden, sollen für andere Anwendungen einfach nutzbar sein. Dazu wird für den Datenaustausch und die Sicherung der Informationen des Editors der XMI Standard verwendet.

4.3. UI Prototyp

Der vorliegenden Prototyp der Benutzerschnittstelle vereint die Elemente der grafischen Benutzerschnittstelle, die hinsichtlich der Realisierung der funktionalen Anforderungen relevant sind. Abbildung 4.1 zeigt einen Prototyp, der die Benutzerschnittstelle des visuellen Editors für BiCEPS abbildet. Auch die Darstellung der Elemente wird hier beschrieben.

Die Palette beinhaltet alle verfügbaren Elemente, welche über Drag and Drop auf die Zeichenfläche verschoben werden können. Unter verfügbaren Elementen versteht man die visuellen Repräsentanten der Komponenten (EDT, BiCEP und AEX) in BiCEPS. Alle Elemente werden durch Rechtecke dargestellt. Ein EDT hat einen eckigen Einschnitt am linken Rand. Ein AEX hat eine Spitze, die nach rechts zeigt. Ein BiCEP hat keine zusätzlichen visuellen Merkmale. Diese Elemente können über Linien verbunden werden, welche die Buffer des BiCEPS Frameworks repräsentieren. Die Eventnamen der jeweiligen Buffer werden als Linienüberschrift angezeigt. Sind mehrere vorhanden, werden diese mit Beistrichen getrennt.

Die Eigenschaften des ausgewählten Elements können über ein separates Fenster, über Eingabefelder, verändert werden. Über den Button „Check Syntax“ kann eine Eventdefinition mittels BiCEPS Parser validiert werden. Das Modell kann über den „Run Model“-Button ausgeführt werden.

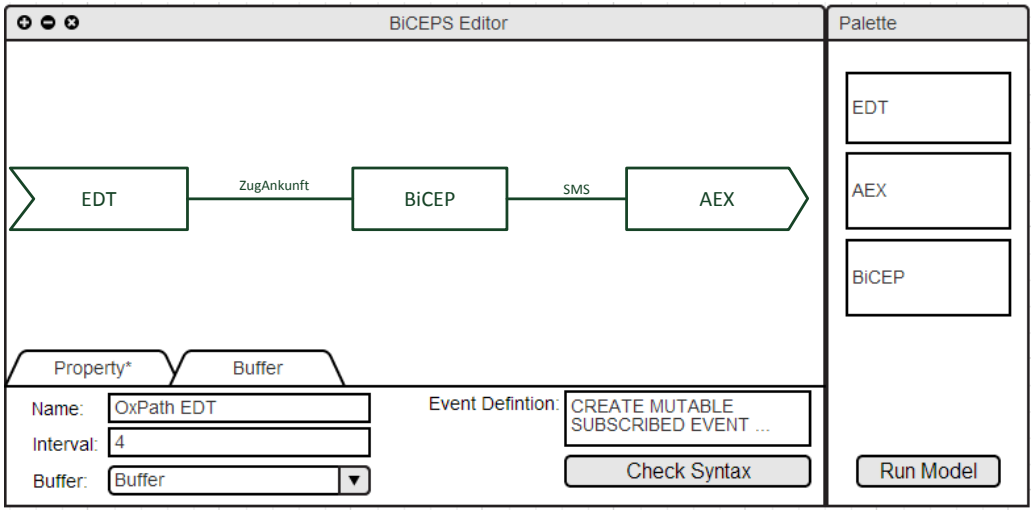

Abbildung 4.1.: UI Prototyp

5. Evaluierung der Frameworks

Dieses Kapitel evaluiert Frameworks zur Erstellung von Editoren in Eclipse (GEF, GMF, Graphiti), beschrieben in den Kapiteln 3.3.1 bis 3.3.3, auf ihre Eignung zur Implementierung eines BiCEPS-Editors. Dafür werden, gemäß den Anforderungen, definiert in Kapitel 4, Kriterien festgelegt, und anschließend ausgewertet. Als Grundlage der Evaluierung wurde für jedes der drei Frameworks ein eigener (unvollständiger) Prototyp, welcher das Erstellen und Verbinden von BiCEPS-Elementen ermöglicht, entwickelt. Am Ende dieses Kapitels wird das Ergebnis dieser Evaluierung zusammengefasst.

5.1. Kriterien

In Anlehnung an Refsdal (2011), welcher eine Studie über den Vergleich von GMF und Graphiti publiziert hat, wurden folgenden Kriterien ausgewählt:

- Funktionalität und Umfang:
  Beschreibt die Mächtigkeit und somit die zur Verfügung stehenden Werkzeuge der Frameworks die eine Implementierung des BiCEPS-Editors unterstützen.
- Erweiterbarkeit und Anpassung:
  Aufgrund von individuellen Anpassungen und Erweiterungen soll eine einfache und unkomplizierte Erweiterung durch zusätzlichen externen Java Code möglich sein.
- Erlernbarkeit:
  Aufgrund des Umfangs und der Anpassung des BiCEPS Modells soll sich der Umfang für die Einarbeitungszeit in Grenzen halten. Weiters soll auf bereits existierende Technologien aufgebaut werden.
- Wiederverwendbarkeit:
  Für spätere Weiterentwicklungen oder Erweiterungen soll es möglich sein, Teile des Editors in anderen Szenarien wieder zu verwenden. Somit soll hier eine Technologie ausgewählt werden, welche mit andern langfristig kompatibel ist.

5.2. Evaluierung der Kriterien

In diesem Abschnitt werden die Technologien (GEF, GMF, Graphiti) anhand der im vorherigen Kapitel festgelegten Kriterien verglichen und beschrieben.

Funktionalität und Umfang: Alle drei Frameworks entsprechen den visuellen Anforderungen (Kapitel 4.1.1) und bieten somit alle Funktionalitäten, die für die Erstellung eines BiCEPS Editors notwendig sind. Die Frameworks sind somit in der Lage, einfache visuelle Objekte zu zeichnen und zu verbinden. Auch Funktionen wie Drag & Drop, Zoomfunktion des Zeichenbereichs und eine Palette werden von allen drei Technologien zur Verfügung gestellt. GEF ermöglicht durch Draw2D jede beliebige Darstellung von visuellen Elementen. Die Frameworks GMF und Graphiti bauen auf dem GEF Framework auf und können somit auch alle Funktionalitäten nützen, die GEF zur Verfügung stellt. Durch eine eingezogene Abstraktionsschicht, welche für den Prozess der Modellerstellung benötigt wird, ermöglicht GMF im Vergleich zu GEF eine erleichterte, durch den Assitenten gestützte Erstellung eines Editors. Durch diese Zwischenschicht wird die Komplexität von Draw2D versteckt, jedoch ist auch kein direkter Zugriff auf die Figures möglich. Diese Schnittstelle zu dem detaillierten Draw2D Framework kann bei komplexen Editoren oft von Vorteil sein. Das Graphiti Framework verfolgt die Idee, die darunterliegende Schicht vollkommen zu verdecken. Zwar bietet Graphiti dieselbe Grundfunktionalität wie die anderen beiden Frameworks an, jedoch kann diese in speziellen Fällen durch individuelle Anforderungen nur über Umwege realisiert werden. Grund dafür ist die Verschleierung aller darunterliegenden komplexeren Zeichenschichten.

Erweiterbarkeit und Anpassung: Um die Anforderung der Syntaxprüfung (Kapitel 4.1.2) zu realisieren, muss auf externen Quellcode zur Laufzeit zugegriffen werden. Dazu muss das visuelle Framework mit individuellem Quellcode erweitert oder angepasst werden. Bei GMF handelt es sich um ein generatives Framework, dies bedeutet, dass der Quellcode für den Editor automatisch generiert wird. Zwar kann hier mittels „EMF-Merge“ (Kapitel 3.2.4) ein individueller Code integriert werden, jedoch ist die Wartung und Migration von verschiedenen Versionen deutlich aufwändiger als eine zentrale Version zu verwalten. Dies führt auch zu erhöhter Fehleranfälligkeit und die Konsistenz kann nicht eingehalten werden. Die Integration von externen Quellcodes ist bei GEF und Graphiti deutlich leichter. Dazu ist zu erwähnen, dass es in GEF hierzu keine vorgesehenen Mechanismen gibt, es ist eine manuelle Einbettung notwendig. Laut Refsdal (2011) ist die Erweiterbarkeit und Wartung zwischen den einzelnen Modellen bzw. die Integration von anderen Quelltexten durch die einfache Architektur von Graphiti gegeben.

Erlernbarkeit: Hier liegt die große Schwäche des mächtigen GEF Frameworks. Die Einarbeitungszeit ist enorm, da auf keine zusammengestellten Elemente (z.B. UML Klassen) zurückgegriffen werden kann. Auch die Kommunikation mit einem Datenmodell muss individuell implementiert werden. Refsdal (2011) unterscheidet bei der Erlernbarkeit hinsichtlich des GEF -Editors zwei Kriterien. Als erstes werden die „non-customized“ Editoren beschrieben, welche leicht und schnell durch das Dashboard realisiert werden können. Das Dashboard dient dem Entwickler als Richtlinie und führt durch die Erstellung des Editors. Das zweite Kriterium sind die „customized“ Editoren, welche ohne Einblick in die API nicht realisiert werden können. Die Erstellung eines einfachen Editors unter Graphiti dauert deutlich länger als die Erstellung eines „non-customized“ Editors mit GMF. Grund dafür ist, dass für jede Funktion und jedes Element eine Klasse manuell angelegt werden muss. Jedoch ist die Individualisierung in Graphiti durch die Java API deutlich leichter und flexibler. Hingegen gibt der Prozess der Modellerstellung eine einheitliche standardisierte Lösung vor. Weiters weist Wenz (2012) auf die hohe Prototypenfähigkeit des Graphiti Frameworks hin.

Wiederverwendbarkeit: Mit einer strikt vorgegebenen Architektur, die leicht verständlich ist, bietet das Graphiti Framework alle Möglichkeiten, um Elemente wiederzuverwenden. Das Fehlen von vordefinierten Schnittstellen in GEF und GMF erschwert die Erfüllung des Kriteriums.

5.3. Evaluierungsergebnis

Auf Basis der obigen Evaluierung hat sich das Graphiti Framework als am besten geeignet für die Implementierung des BiCEPS-Editors herausgestellt. Im Folgenden wird diese Entscheidung begründet:

Die visuellen Anforderungen (Kapitel 4.1.1) stellen keinen außerordentlichen Anforderungen an die grafische Benutzerschnittstelle. Graphiti deckt alle geforderten Funktionen hinsichtlich visueller Darstellung und Funktionen zur Bearbeitung von Elementen am besten ab. Die Sicherung der Elemente (Kapitel 4.1.5) und die Anpassung des Quellcodes (Kapitel 4.1.2) sind in Graphiti einfach, da durch die Java API schnell individueller Quellcode eingepflegt werden kann. Auch externe Zugriffe können über diese leicht getätigt werden. Auch in puncto Erlernbarkeit kann Graphiti gegenüber GEF und GMF punkten. Weiters erlaubt das Featurekonzept eine agile Entwicklung des Editors. Da es noch kein vollständiges BiCEPS während der Umsetzung des Editors gab, ist dies von Vorteil.

6. Design

Ziel der Umsetzung ist die Erstellung eines grafischen Editors auf Basis des konzeptionellen Modells von BiCEPS (Abbildung 2.5), welcher sich über eine standardisierte Schnittstelle parametrisieren lässt. Somit können individuelle Komponenten eines BiCEPS implementiert und mit dem Editor in ein BiCEPS Programm eingebunden werden. Durch den BiCEPL Compiler können die definierten Events validiert werden. Durch eine Schnittstelle kann die visuelle Darstellung eines BiCEPS Modell mit einem Mapper in ein lauffähiges BiCEPS umgesetzt werden. Für die Umsetzung der visuellen Darstellung wird das Graphiti Framework (Kapitel 3.3.3) verwendet. Die Verwaltung der zugrunde liegenden Datenstruktur übernimmt EMF (Kapitel 3.2). Alle weiteren Komponenten werden in Java entwickelt.

Dieser Abschnitt beschreibt mit Hilfe der Abbildung 6.1 den Aufbau des Editors und das Zusammenspiel der Komponenten (BICEPS-RUNTIME, BICEPL-COMPILER, BICEPS-MAPPER, PERSISTENCE-PROVIDER, BICEPS-METAMODEL, BICEPS-MODEL, sowie PALETTE und BICEPS-EDITOR). Es werden auch die verwendeten Technologien für die jeweiligen Komponenten beschrieben.

6.1. BiCEPS Framework

Der Editor baut auf dem bestehenden BiCEPS Framework auf. Dieses Framework kann in zwei Komponenten, die in Abbildung 6.1 rot markiert sind, unterteilt werden. Zum einen gibt es die BiCEPS-RUNTIME, die in einer Java Applikation die Laufzeitumgebung für ein BiCEPS bereitstellt, zum anderen gibt es den BiCEPL-COMPILER. Dieser validiert die in BiCEPL definierten Eventdefinitionen und setzt diese so um, dass sie von der BiCEPS-RUNTIME verwendet werden können. Das entspricht den Anforderungen, die in Kapitel 4.1.2 und 4.1.6 beschrieben werden. Diese beiden Komponenten sind vom Editor getrennt und unabhängig. Um einen Zugriff zwischen diesen externen Komponenten und dem visuellen Editor zu ermöglichen, sind Schnittstellen erforderlich. Diese werden in Kapitel 6.6 angeführt.

6.2. BiCEPS-Editor

Das Kernstück, der BiCEPS-EDITOR, ist in Abbildung 6.1 blau dargestellt, wird in Graphiti umgesetzt und ist für die visuelle Darstellung der BiCEPS Komponenten (EDT, AEX und BiCEP) und deren Buffer zuständig. Auch die Benutzerschnittstelle für die Eventdefinitionen wird hier bereitgestellt. Der BiCEPS-Editor dient dazu, die visuellen Anforderungen (Kapitel 4.1.1) zu erfüllen.

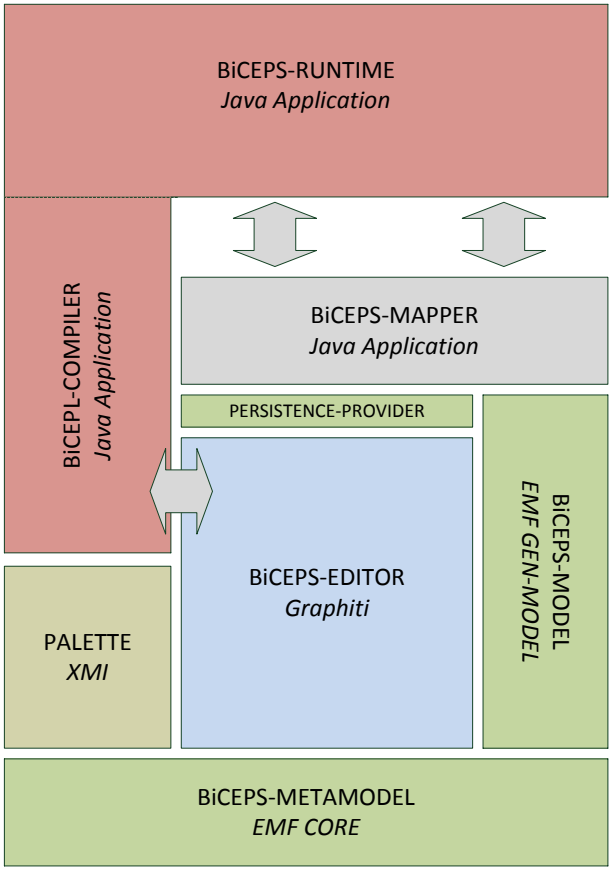

Abbildung 6.1.: Umsetzungsarchitektur

6.3. Datenmodell

Das Datenmodell setzt sich aus dem BICEPS-METAMODEL, dem BICEPS-MODEL und dem PERSISTENCE-PROVIDER zusammen, diese Komponenten sind grün gekennzeichnet. Das Datenmodell für ein BiCEPS, welches das konzeptionelle Modell widerspiegelt, nennt sich BICEPS-METAMODEL und wird in EMF umgesetzt. Es beschreibt die Struktur des Datenmodells, welches vom Editor für die Verwaltung der Daten genutzt wird. Dieses Datenmodell wird in Form eines ECore bereitgestellt. Auf diesem ECore wird mit dem Generatormodell (Kapitel 3.2.4) von EMF ein Java Datenmodell erstellt. Dieses BICEPS-MODEL ermöglicht den Zugriff des Editors auf die Datenobjekte mit Java. Die Instanzen des Datenmodells können mit Hilfe von EMF in Form einer XMI Datei gesichert werden. Dazu müssen die Instanzen des Modells mit dem PERSISTENCE-PROVIDER serialisiert und persistiert werden. Diese Anforderung wurde in Kapitel 4.1.5 beschrieben.

6.4. Palette

Um eine Konfiguration des Editors von außen zu ermöglichen, muss dieser parametrisierbar sein. Damit ist die Erfüllung der Anforderungen der externen Konfiguration, die in Kapitel 4.1.1 beschrieben sind, gewährleistet. Die Komponente PALETTE dient als Schnittstelle für diese Konfiguration. Die Informationen über die Komponenten (EDT, BiCEP und AEX) werden in einer XMI Datei abgelegt. Anhand des BICEPS-METAMODEL und des BICEPS-MODEL kann der Editor die Informationen auslesen und die konfigurierten Elemente bereitstellen. Diese Konfiguration beinhaltet den Namen der Komponente und deren Funktionalität.

6.5. BiCEPS-Mapper

Die Umsetzung des visuellen Modells in ein ausführbares Programm wird durch den BICEPS-MAPPER realisiert. Dieser hat die Aufgabe, die visuell dargestellten Komponenten in BiCEPS Komponenten (EDT, BiCEP und AEX) zu transformieren, um diese in der Laufzeitumgebung ausführen zu können (siehe Anforderung Kapitel 4.1.6). Dieser Mapper wird in Java implementiert und kann auch als Standalone Anwendung ausgeführt werden. Der Mapper greift entweder auf die serialisierten und persistierten Dateien des PERSISTENCE-PROVIDER oder direkt über EMF auf die Instanzen des BICEPS-MODEL zu.

6.6. Schnittstellen

Die Schnittstelle zwischen dem Editor und BICEPS wird in Abbildung 6.1 durch graue Pfeile dargestellt. Der BICEPL- COMPILER wird vom BICEPS-EDITOR direkt über eine vordefinierte Schnittstelle in Java aufgerufen. Die Umsetzung des visuellen Modells in ein ausführbares Programm wird durch den BICEPS-MAPPER realisiert. Diese kann einerseits über XMI auf die abgespeicherten Instanzen des BICEPS-MODEL zugreifen, andererseits kann direkt zur Laufzeit auf die Elemente des BICEPS-EDITOR zugegriffen werden.

7. Implementierung des Editors

Dieses Kapitel beschreibt die Implementierung des visuellen Editors und die Verwendung der Technologien für jene Komponenten, welche bereits in Kapitel 6 beschrieben wurden. Die Implementierung des visuellen Editors ist in drei Eclipse Plug-ins (BiCEPSModel, BiCEPSEditor, BiCEPSMapper) unterteilt. Das BiCEPSModel ist mit EMF implementiert und enthält das Datenmodell und Funktionalitäten, die für den visuellen Editor benötigt werden. Der Editor selbst ist im BiCEPSEditor Plug-in enthalten und ist mit dem Graphiti Framework implementiert. Die Schnittstelle zwischen BiCEPSEditor und BiCEPS Framework ist der BiCEPSMapper. Alle Plug-ins zusammen ergeben den visuellen Editor, der den Anforderungen, die in Kapitel 4 beschrieben sind, entspricht.

Das bestehende BiCEPS Framework und dessen Aufbau wurde bereits in Kapitel 2.5.2 thematisiert. Diese Komponente wurde in Java umgesetzt und die verwendeten Zugriffe und deren Eigenschaften werden in Kapitel 7.1 beschrieben. Anschließend wird das Datenmodell (BiCEPS-METAMODEL, BiCEPS-MODEL) und der PERSISTENCE-PROVIDER, welche im Biceps-Model Plug-in enthalten sind, im Kapitel 7.2 beschrieben. Der visuelle Editor (BiCEPSEditor) selbst wird im Kapitel 7.3 beschrieben, es werden die Graphiti spezifischen Implementierungsdetails erörtert. Die Palette, welche für die Konfiguration des Editors verantwortlich ist und deren Umsetzung in XMI erfolgt, wird im Kapitel 7.3.2 behandelt. Abschließend wird in Kapitel 7.4 der Mapper und dessen Schnittstelle thematisiert.

7.1. Implementierung BiCEPS Framework - BICEPS

Der Aufbau und das zugrundeliegende konzeptionelle Modell wurde bereits in Kapitel 2.5.2 beschrieben. Der BiCEPL Compiler wird zur Laufzeit vom Editor angesprochen (Kapitel 4.1.2), dadurch ergibt sich hier die einzige direkte Abhängigkeit zum BiCEPS Framework. Die Klasse des BiCEPS Compiler wird dynamisch zur Laufzeit nachgeladen und ist somit zur Laufzeit des Editors verfügbar. Alle anderen Komponenten des BiCEPS Frameworks sind vom Editor unabhängig.

7.2. Implementierung Datenmodell - BiCEPSModel

Das Datenmodell besteht aus BiCEPS-METAMODEL, dem BiCEPS-MODEL und dem PERSISTENCE-PROVIDER. Die Implementierung der Komponenten befindet sich im BiCEPSModel Plug-in und wird mit Hilfe von EMF umgesetzt. Die Sicherung dieses Datenmodells wird durch den PERISTENCE-PROVIDER ermöglicht, welcher ebenfalls mit Hilfe von EMF umgesetzt ist.

In diesem Kapitel wird die Umsetzung des Datenmodells beschrieben, im Kapitel 7.2.1 das logische Modell des Editors, genannt BiCEPS-METAMODEL. Dieses logische Modell wird in Ecore spezifiziert und ist die Basis für das Javamodell, bezeichnet als BiCEPS-MODEL, (Kapitel

7.2.2), welches mit dem EMF Generatormodell realisiert wird. Die Speicherung der Daten wird in Kapitel 7.2.3 beschrieben.

7.2.1. BiCEPS-METAMODEL

Für die Umsetzung des visuellen Editors ist das konzeptionelle Modell des BiCEPS (Abbildung 2.5) so erweitert worden, dass die Anforderungen (Kapitel 4.1) des visuellen Editors erfüllt werden. Das logische Modell ist in EMF umgesetzt und wird im ECore abgelegt. Diese ECore Datei wird in XMI Syntax definiert und ist schwer lesbar, deswegen wird das logische Modell in UML Notation anhand Abbildung 7.1 dargestellt und beschrieben. Die gerichteten Beziehungen (Pfeile) zeigen die Richtung des möglichen Zugriffs auf Attribute an. Beziehungen ohne gerichteten Pfeil sind bidirektional umgesetzt, somit ist eine eOpposite-Beziehung (Kapitel 3.2.3) implementiert. Alle weiteren Beziehungsimplementierungen werden in den folgenden Absätzen bei den jeweiligen Beziehungen detailliert beschrieben.

In den abstrakten Basisklassen BiCEPS-ELEMENT und BiCEPS-BUFFER sind die Attribute name und classPath definiert. Der Name eines Elements wird im Attribut *name* festgelegt. Der *classPath* beinhaltet den Pfad jener Klasse, welche die Funktionalität der jeweiligen Komponenten in Java implementiert. Die Kopiermethoden (*copy*, *copyBuffer*) ermöglichen das Vervielfältigen von Komponenten (Kapitel 7.3.3). Zusätzlich wird *purge* mit einem Long-Wert im BiCEPS-BUFFER vermerkt. Die Klasse *EventClass* hat die Attribute *name*, *definition*. Das Attribut *definition* beinhaltet die Eventdefintion in BiCEPL und repräsentiert diese als String; *lifespan* wird in einem Integerwert abgelegt. Jede Eventklasse ist mit einem Namen (*name*) beschrieben.

Das BiCEPS-MODEL ist um die Attribute *dbPath*, *modelPath* und *picoPath* erweitert. Diese drei Attribute werden als Zeichenkette (EString) repräsentiert. Der *dbPath* spezifiziert jenen absoluten Pfad einer Datenbank, in welchem die verarbeiteten Events eines BiCEPS gespeichert werden. Der *modelPath* beschreibt den Pfad, in welchem das zu verwendende BiCEPS FRAMEWORK (JAR File) abgelegt ist, der *picoPath* jenen Pfad, wo die Palettendefinition (PALETTE) abgelegt ist. Ein BiCEPS-MODEL hat keine oder mehrere EventClasses, welche in einer Liste (*eventList*) gespeichert werden. Das BiCEPS-MODEL kann keinen oder auch mehrere BiCEPS-BUFFER beinhalten, diese werden in der Liste *bufferList* definiert. Die BiCEPS-ELEMENT (EDT, BiCEP und AEX) sind existenzabhängig vom BiCEPS-MODEL und werden in der elementList vermerkt. Die drei Listen (*elementList, bufferList und eventList*) werden als containment Beziehungen (Kapitel 3.2.3) implementiert, somit sind diese vom BiCEPS-MODEL existenzabhängig.

Ein EDT definiert sein Intervall (*intervall*) mit einem Double-Wert (EDouble). Sein SubscribedEvent wird im *subscribedEvent* Attribute abgelegt. Dieses wird mit der EMF Eigenschaft containment definiert. Der EDT hat einen oder mehrere Parameter, die in einer Liste (*parameterList*) beschrieben werden. Die Parameter sind existenzabhängig (containment) vom EDT und werden mit einem Namen (*name*) und einem Wert (*value*) als String definiert. Die Beziehung zwischen EDT und Parameter ist gerichtet, es kann nur seitens des EDT auf die Parameter zugegriffen werden. Ein EDT beinhaltet eine oder keine Referenz zu einem BiCEP (subscribedBiCEP). Der EDTBuffer ist im *detector* definiert.

Ein BICEP definiert das Intervall zwischen Verarbeitungszyklen (chronon) als Long-Wert (ELongObject). Die Attribute *classPathMapper* und *classPathTaskmanager* definieren den abso-

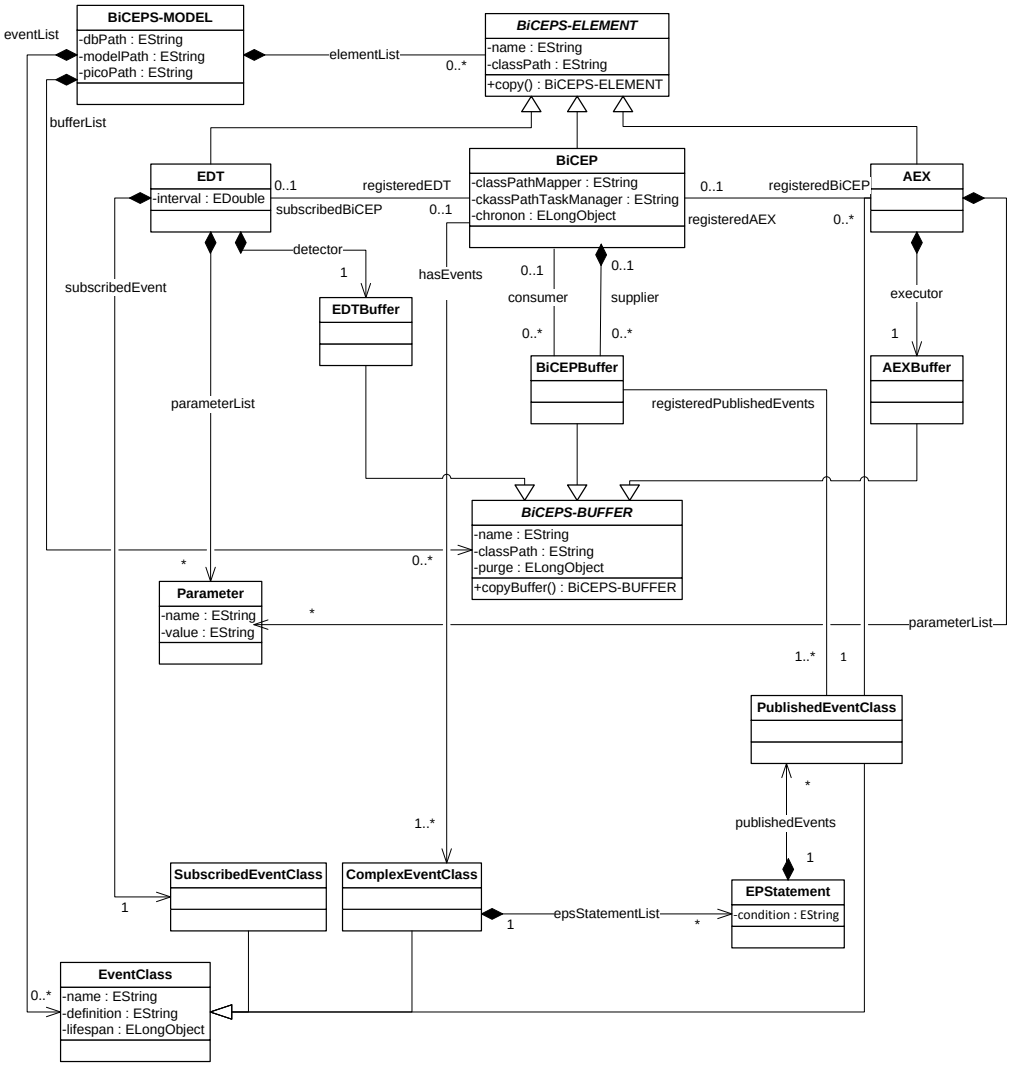

Abbildung 7.1.: Logisches Modell BiCEPS

luten Pfad als String der jeweiligen Java Klassen. Die in Kapitel 2.5.2 beschriebenen Readbuffer und Writebuffer werden in der Implementierung als *consumer* und *supplier* definiert und bilden die Referenz zu einem oder mehreren BiCEPBuffer. Die Beziehung von einem *supplier* zu einem BiCEPS ist mittels containment, also ist der BiCEPBuffer vom BICEP existienzabhängig, realisiert. Ein BiCEPS hat eine oder keine Referenz zu seinem EDT (*registeredEDT*) und eine Referenz zu seinem AEX (*registeredAEX*). Eine oder mehrere ComplexEventClasses werden in der Liste *hasEvents* vermerkt. Der BiCEPBuffer speichert und legt ein oder mehrere PublishedEvents in der Liste *registeredPublishedEvents* ab.

Ein AEX hat eine oder keine Referenz auf einen BiCEP (*registeredBiCEP*). Sein AEXBuffer wird in einer containment Beziehung *executor* beschrieben. Die Parameter werden wie bei einem EDT in einer *parameterList* definiert. Ein AEX hat genau eine PublishedEventClass, welche im Attribut *publishedEvent* beschrieben ist.

Die EventClasses (SubscribedEventClass, ComplexEventClass, PublishedEventClass) und deren Beziehungen wurde bereits in Kapitel 2.5.2 beschrieben. Hierzu ist nur anzumerken, dass die *condition* (Eventdefinition) eines EPStatement als String implementiert ist.

7.2.2. BiCEPS-MODEL

Um den Zugriff vom Graphiti Framework, das in Java implementiert ist, zum Datenmodell zu ermöglichen, wird das Ecore Modell in ein Javamodell transferiert. Diese Aufgabe wird vom EMF GenModel (Kapitel 3.2.4) übernommen. Der generierte Java Quellcode bildet das logische Modell (Abbildung 7.1) vollkommen ab. Für jede EMF Klasse (Abbildung 7.1) erstellt der EMF Generator ein Interface (z.B. EDT) mit dem gleichen Namen und eine Klasse mit dem Postfix Impl (z.B. EDTImpl), die das jeweilige Interface implementiert. Die Impl-Klassen beinhalten den generierten Quellcode. Dieser Quellcode beinhaltet Getter und Setter Methoden für die Membervariablen und die Logik, welche für die Verwaltung der Objektstruktur (Notifier, Adapter) benötigt wird. Die Methoden `copy` und *copyBuffer* zum Vervielfältigen der BiCEPS Komponenten (EDT, BiCEP, AEX, EDTBuffer, BiCEPBuffer und AEXBuffer) müssen jedoch manuell erweitert werden, da es sich hier um leere Methoden handelt und diese keine Funktionalität bieten. Dazu wird die Methode *copy* in den Klassen EDTImpl, AEXImpl und BiCEPImpl manuelle erweitert. Auch in den Buffern EDTBuffer, BiCEPBuffer und AEXBuffer wird die Methode `copyBuffer` manuell erweitert.

Um den generierten Quellcode manuell zu erweitern, wird in den Methodenköpfen die Annotation `@generated` gelöscht. Somit wird der manuell erweiterte Quellcode bei der nächsten Generierung nicht mehr überschrieben. Der Quellcodeauzug 7.1 zeigt die Erweiterung der Kopierfunktion (`copy`) des EDT (`EDTImpl`) und dient als Beispiel. Es wird mittels der Factory (BiCEPSFactory)) eine neue Instanz eines EDT erstellt. Diese wird mit dem Klassenpfad (`classPath`), Namen (`name`), den Parametern (`parameterList`) und dem EDTBuffer (`detector`) mittels der Getter und Setter befüllt.

```
1 @Override
2   public BiCEPS_ELEMENT copy() {
3     EDT copy = BiCEPSFactory.eINSTANCE.createEDT();
4     copy.setClassPath(this.classPath);
5     copy.setName(this.getName());
6     copy.getParameterList().addAll(this.getParameterList());
```

```
7       copy.setDetector(this.getDetector());
8       return copy;
9     }
```

Quellcode 7.1: Manuelle Erweiterung EDT Copy-Methoden

7.2.3. PERSISTENCE-PROVIDER

Auch die Speicherung der Objekte des Datenmodells ist in EMF umgesetzt. Dazu wird das in EMF eingebaute Persistence Framework verwendet. Das Framework serialisiert und persistiert die Javaobjekte. Diese Serialisierung wird mittels RessourceSets von EMF (Kapitel 3.2.3), welche als Container für EMF Instanzen und deren Verwaltung dienen, ermöglicht. Hierbei wird der XMI Standard verwendet (Anhang C.5). Die vom Editor verwendeten Javaobjekte (EMF Instanzen) werden mittels dem RessourceSet serialisiert und persistiert. Die Objekte des Editors werden in einer .bicep Datei (Anhang C.5) abgelegt. Die dazu benötigen Klassen (Adapter, Factory) werden vom GenModel beim Generieren des Quellcodes erstellt. Diese Funktionalitäten, welche für das Serialisieren und Persistieren von Objekten in EMF benötigt werden, sind in Adaptern (RessourceAdapter) und Factories (RessourceFactory) implementiert.

7.3. Implementierung Editor - BICEPSEditor

Der BICEPSEditor dient als Benutzerschnittstelle für ein BiCEPS. Dieser wird mit dem Graphiti Framework (Kapitel 3.3.3) umgesetzt. Die Datenverwaltung der modellierten Elemente und deren Eigenschaften wird vom EMF übernommen. Der Editor besteht aus einem Assistenten (Wizard), der das Anlegen eines visuellen BiCEPS unterstützt. Der Zugriff auf EMF wird durch ein EMF-Service, das die Schnittstelle zwischen Graphiti und EMF bildete, realisiert. Die visuelle Darstellung und die Modifikation der Elemente (EDT, BiCEP, AEX, EDTBuffer, BiCEPBuffer, AEXBuffer, SubscribedEventClass, ComplexEventClass und PublishedEventClass) selbst ist mit dem Graphiti Diagram implementiert. Diese Komponenten können über die PALETTE parametrisiert werden. Der Eclipse Plug-in BicepsEditor teilt sich somit in vier Aufgabenbereiche auf: Wizard, EMF Service, Graphiti Diagram und die Palette.

Dieser Abschnitt beschreibt die Implementierung des visuellen Editors. Dazu wird vorerst der visuelle Aufbau des Eclipse Plug-ins (Kapitel 7.3.1) beschrieben. Anschließend wird in Kapitel 7.3.2 die Umsetzung der Palette vorgestellt. Abschließend werden in Kapitel 7.3.3 der Assistent, die EMF Schnittstelle und die Umsetzung der Komponenten für die visuelle Darstellung des Editors beschrieben.

7.3.1. Benutzeroberfläche des Editors

Abbildung 7.2 zeigt die Benutzeroberfläche des visuellen Editors. Dieser setzt sich aus *Diagram*, *Palette* und *Properties* zusammen. Die *Palette* zeigt verfügbare Komponenten, welche von außen parametrisierbar sind, an. Das *Diagram* ist die Zeichenfläche, in welchem die Komponente (EDT, BiCEP und AEX) von der *Palette* mittels Drag & Drop platziert werden. Beim Auswählen einer Komponente im Zeichenbereich kann diese mittels dem Pfeilwerkzeug mit anderen Komponenten verbunden werden. Weiters sind die Eigenschaften der Komponenten über die *Properties*

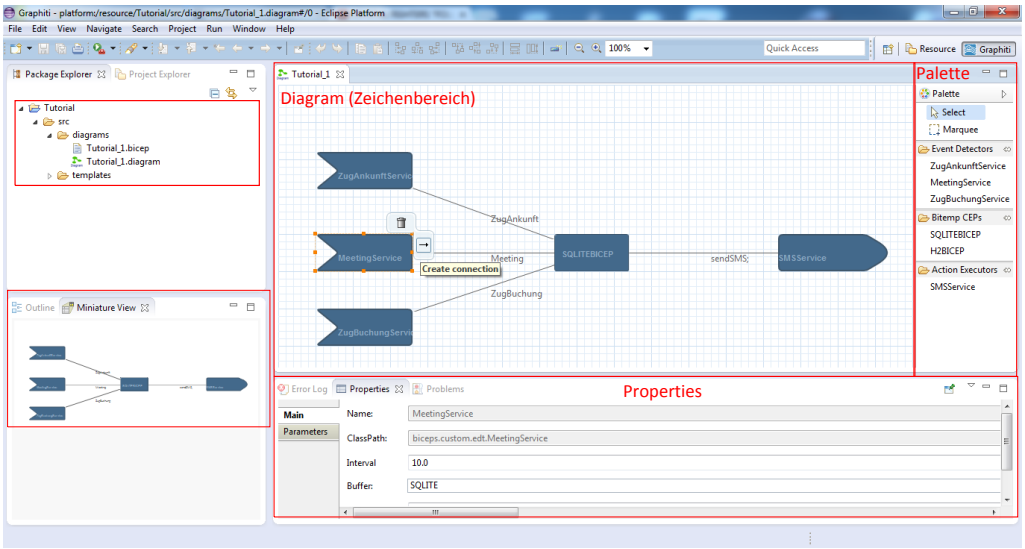

Abbildung 7.2.: Benutzeroberfläche BiCEPSEditor

editierbar. Die Umsetzung der beschriebenen grafischen Objekte werden in den nachfolgenden Kapiteln beschrieben.

7.3.2. Palette

Die Palette (Abbildung 4.3) des visuellen Editors definiert, welche Komponenten im visuellen Editor verwendet werden können. Diese Palette ist parametrisierbar und wird außerhalb des Editors definiert. Für jedes Diagramm, welches mit dem Diagramwizard (7.3.3) erstellt, wird können unterschiedliche Paletten definiert werden. Die Palette wird in einer XMI Datei spezifiziert, welche beim Erstellen des Diagramms angegeben wird.

Die XMI Datei entspricht dem Schema des BICEPS-METAMODEL, welches als XSD Datei hinterlegt ist. Durch dieses XML-Schema wird die Datenintegrität der XMI Datei gewährleistet. Die Informationen der Palette werden vom PERSISTENCE-PROVIDER ausgelesen und in Javaobjekte umgewandelt. Somit ist ein direkter Zugriff für das Graphiti Framework für die visuelle Darstellung möglich.

Der Aufbau der Datei spiegelt das logisches Modell (Abbildung 7.1) des Datenmodells wider. Alle möglichen Elemente werden im `InitialModel` spezifiziert. Darin sind die Komponenten eines BiCEPS (Kapitel 2.2.1) über den TAG `possibleElement` definiert. Dazu muss als `xsi:code` der jeweilige Type der Komponente (EDT, BiCEP und AEX) definiert werden. Weiters ist der Name (`name`) und der dazugehörige `classpath`, welcher die Klasse, die die Funktionalität der Komponente beschreibt, zu definieren. Die Parameternamen für einen EDT und einen AEX sind in der `paramList` mittels `key` spezifizierbar. Die Buffer (EDTBuffer, BiCEPBuffer, AEXBuffer) werden über die TAGS `possibleEDTBuffer`, `possibleBiCEPBuffer` und `possibleAEXBuffer` definiert. Jeder TAG trägt einen Namen (`name`) und definiert den `classpath` der Klasse.

Der Quellcode 7.2 zeigt die Definition der EDTs ZugAnkunftService (Zeilen 3-5) und MeetingService (Zeilen 7-8). Das ZugAnkunftService hat einen Parameter mit dem Namen zugNr (`key`). Weiters ist ein EDTBuffer mit dem Namen SQLITE (Zeile 10) in dieser Beispielkonfiguration der Palette definiert. Die Klassenpfade (`classpath`) spezifizieren jene Klassen, welche die Funktionalität der Komponenten implementiert und schlussendlich im BiCEPS Framework ausgeführt werden.

```
1  <biceps:InitialModel xmlns:xsi="http://www.w3.org/2001/
       XMLSchema-instance" xmlns:biceps="http://www.mod4j.org/
       biceps">
2
3   <possibleElements xsi:type="biceps:EDT" name="
       ZugAnkunftService" classPath="biceps.custom.edt">
4         <parameterList key="zugNr"/>
5    </possibleElements>
6
7    <possibleElements xsi:type="biceps:EDT" name="MeetingService
       " classPath="biceps.custom.edt.MeetingService">
8    </possibleElements>
9
10   <possibleEDTBuffer name="SQLITE" classpath="biceps.
       event_buffer.edtBuffer.EDTSQLiteBuffer"/>
11
12 </biceps:InitialModel>
```

Quellcode 7.2: Palettenkonfiguration

7.3.3. Komponenten des Editors

Die Funktionsbereiche des Editors - der Wizard, das EMF Service und das Graphiti Diagram - spiegelt sich im Aufbau der Packagestruktur des Editors wider. Die drei Komponenten werden in den Packages `bicepseditor.wizard` (Kapitel 7.3.3 - Wizard), `bicepseditor.diagram` (Kapitel 7.3.3 - Graphiti Diagram) und `bicepseditor.service` (Kapitel 7.3.3 - EMF Service) in Java implementiert.

Wizard

Der visuelle Assistent (`bicepseditor.wizard`) unterteilt sich in den Projektwizard und den Diagramwizard. Der Projektwizard legt die Projektstruktur im jeweiligen Workspace für das Eclipse Projekt an. Weiters werden hier Beispielkonfigurationen für einen möglichen Anwendungsfall (siehe Tutorial: 8.4) hinterlegt. Der Diagramwizard erstellt das Diagramm selbst. Dieser dient als Benutzerschnittstelle für die verwendeten Ressourcen (Palette, BiCEPS RunTime). Es ist der Pfad für die XMI Datei der Palette, die JAR Datei der BICEPS Runtime und die Datenbank für den BiCEP anzugeben. Die beiden Assistenten sind mit der Wizardklasse von JFace in Java implementiert. Die Eingabe der Attribute des BiCEPS-MODEL (picopath, dbPath, modelPath) ist mit JFace Textfeldern umgesetzt. Abbildung 8.1 und 8.5 veranschaulichen die Benutzeroberfläche der Wizards.

EMF - Service

Das EMF Service (`bicepseditor.services`) ist die Schnittstelle zwischen Java und EMF. Es ermöglicht den Zugriff vom BiCEPSEditor zum BiCEPSModel. Somit ist ein Zugriff vom javabasierten Graphiti Framwork zum Datenmodell (BiCEPS-MODEL) möglich.

Dieses EMF Service unterteilt sich in `BiCEPSInitService` und in `BiCEPSModelService`. Das `BiCEPSInitService` initialisiert die Palette und liest die Palettenkonfiguration aus. Dadurch kann der Editor auf die möglichen BiCEPS Elemente und deren Buffer zugreifen. Die Klasse liest das XMI File ein und transferiert diese Informationen mittels EMF in das Datenmodell (Kapitel 7.2). Das `BiCEPSModelService` verwaltet den Zugriff auf das Datenmodell. Es können somit die Datenobjekte gelesen und verändert werden. Weiters ist das `BiCEPSModelService` die Schnittstelle zum PERISTENCE-PROVIDER (Kapitel 7.2.3), welcher die Sicherung der Datenobjekte übernimmt. Beide Services sind in Java implementiert und greifen auf das EMF zu.

Graphiti Diagram

Das Graphiti Diagram setzt sich aus dem BicepsEditorToolBehaviour, BicepsEditorFeatureProvider und BicepsEditorDiagramTypeProvider zusammen. Der `BicepsEditorToolBehaviour` ist für die visuelle Darstellung der Palette zuständig. Das Einfügen und Verändern von Elementen im Zeichenbereich wird mit dem `FeatureProvider` ermöglicht. Der `BicepsEditorDiagramTypeProvider` ist die Schnittstelle zwischen den beiden oben erwähnten Providern und ist somit für die Verwaltung und die Kommunikation zuständig. Zusätzlich wird die Benutzeroberfläche für die Eigenschaften (z.B. Intervall eines EDT) der Komponenten (EDT, BiCEP und AEX) mit Hilfe der Eclipse Properties View realisiert.

BicepsEditorDiagramTypeProvider Die Klasse `BicepsEditorDiagramTypeProvider` erweitert den DiagramTypeProvider von Graphiti (Kapitel 3.3.3) und ist der zentrale Kern des BiCEPSEditor. Dieser initialisiert alle Ressourcen (Klassen, Konfigurationen), welche vom Plugin benötigt werden. Weiters übernimmt er die Koordination zwischen Graphiti Diagram und allen anderen Komponenten des Editors (EMF Services, Wizard).

BicepsEditorToolBehaviour Die vom `BiCEPSInitService` (Kapitel 7.3.3) geladenen Komponenten der Palettendefinition werden mit dem `BicepsEditorToolBehaviour` visuell dargestellt. Das `BicepsEditorToolBehaviour` erstellt auch das Contextmenü, welches die Umsetzung der grafischen BiCEPS Komponenten in ein lauffähiges Programm auslöst (siehe Tutorial 8.8). Die Komponente `BicepsEditorToolBehaviour` wird nach der Erstellung des Diagramms durch den `DiagramWizard` aufgerufen.

BicepsEditorFeatureProvider Wie bereits im Kapitel 3.3.3 erwähnt, baut das Graphiti Framework auf einem Featurekonzept auf. Die Verwaltung dieser Features wird vom `BicepsEditorFeatureProvider` im BiCEPSEditor implementiert.

Ein Feature spiegelt eine Funktionalität, welche auf ein Element angewendet werden kann, wider. Unter Funktionalität wird das Erstellen von Elementen und Verbinden von Element verstanden. Solch ein Element kann ein BiCEPS-ELEMENT oder ein BiCEPS-BUFFER sein. Ein Featureobjekt beinhaltet solch ein Element und eine Vorlage (Template Objekt), welche die vorgegebenen Eigenschaften der Komponenten (EDT, BiCEP und AEX) beinhaltet. Die Vorlagen entsprechen den Repräsentanten, welche in der Palette vom `BicepsEditorToolBehaviour` angezeigt werden. Beim Erstellen eines neuen Elements (Drag & Drop von der Palette in den Zeichenbereich) werden die Eigenschaften der Template Objekte mittels der Kopiermethoden, welches bereits in Kapitel 7.2 für BICEPS-ELEMENT und BICEPS-BUFFER beschrieben worden sind, übernommen.

Die Features werden vom `BicepsEditorFeatureProvider` verwaltet. Es wird zwischen drei Kategorien von Features unterschieden. Diese sind je Komponenten (EDT, BiCEP und AEX) ein CREATE-, ADD- und ein UPDATE-Feature. Das CREATE-Feature erstellt die Objekte im EMF und löst das Kopieren der Eigenschaften von den vordefinierten Palettenkomponenten aus. Das ADD-Feature ist für die Darstellung der Komponenten, welche im Kapitel 4.3 definiert sind, zuständig und erstellt diese Repräsentanten mit Draw2D Figures (Kapitel 3.3.1). Das UPDATE-Feature ermöglicht, auf Änderungen im Datenmodell zu reagieren und aktualisiert die grafische Darstellung im Falle eines inkonsistenten Zustandes zwischen visueller Darstellung und dem Datenmodell. Diese Features werden vom `BicepsEditorFeatureProvider` auf die ausgewählten Komponenten im Zeichenbereich angewendet.

Eclipse Properties Alle Eingaben für ein BiCEPS-ELEMENT werden mittels Eclipse Properties View realisiert (Eclipse Documentation, 2013). Solch ein Property beinhaltet ein Repräsentationsobjekt (z.B.: TextView für Namen) und das dazugehörige Datenobjekt (z.B. AEX). Auf Basis eines Filters wird entschieden, welche Properties ein Objekt hat. Diese Kombination zwischen Filter und Properties wird in der Datei *plugin.xml* (Anhang:C.2) dargestellt.

Der Codeauszug 7.3 zeigt einen Ausschnitt aus der *plugin.xml* Datei, welche für alle BiCEPS-ELEMENTE die Property-Attribute *name* und *classPath* lädt. Property-Attribute werden in einer Benutzermaske angezeigt und können bearbeitet werden. Eine Property wird in einer `propertySection` definiert. Dabei wird die Klasse definiert, welche die Anzeige der Elemente übernimmt (Zeilen 3 & 5). Im diesem Fall sind das die Properties `GENERALNameSectionProperty` und `GERNALClassPATHSectionProperty`, welche ein Eingabefeld für die Membervariablen `namen` und `classPath` des jeweiligen BiCEPS-ELEMENT repräsentieren. Der Filter (Zeilen 3 & 5) definiert die Klasse, die überprüft, ob das ausgewählte visuelle Element die Properties unterstützt. Die Filterklasse `GeneralProperiesFilter` prüft, ob das fokussierte Elemente eine Instanz von BiCEPS-ELEMENT ist.

```
1 <!-- GENERAL ELEMENTS -->
2   <propertySections contributorId="BicepsEditor.
       PropertyContributor">
3       <propertySection tab="graphiti.main.tab" class="
          bicepseditor.diagram.properties.
          GENERALNameSectionProperty"  filter="bicepseditor.
          diagram.properties.filter.GeneralPropertiesFilter" id=
          "graphiti.main.tab.main.name"/>
4
```

```
5       <propertySection tab="graphiti.main.tab" class="
            bicepseditor.diagram.properties.
            GENERALClassPathSectionProperty"  filter="bicepseditor
            .diagram.properties.filter.GeneralPropertiesFilter" id
            ="graphiti.main.tab.main.classpath"  afterSection="
            graphiti.main.tab.main.name"/>
6
7       ...
8   </propertySections>
```

Quellcode 7.3: Beispiel Filter-Properties

Das oben beschriebe Konzept für die Properties wird für jeden AEX, EDT,BICEP und die jeweiligen Buffer angewendet und befindet sich im Package `bicepseditor.features.properties`. Die graphische Darstellung der Eingabenelemente wird hauptsächlich über JFace Elemente (Kapitel 3) realisiert.

7.4. Implementierung Schnittstelle - BiCEPSMapper

Der Mapper ist jene Schnittstelle, die die grafischen BiCEPS Komponenten in tatsächliche BiCEPS Komponenten, welche im Framework hinterlegt sind, transferiert. Dies ermöglicht die Ausführung des graphischen Modells in der BiCEPS Laufzeitsumgebung. Die Klassen, die für die Umsetzung des Mappers notwendig sind, werden in Abbildung 7.3 dargestellt.

Der Mapper ist so konzipiert, dass er auch ohne Editor ausgeführt werden kann. Wird der Mapper als Standalonelösung ausgeführt, kann er über den `BiCEPEMFReader` ein persistiertes Modell über eine XMI Schnittstelle auslesen. Andererseits kann ihm zur Laufzeit ein EMF Modell (ECore) übergeben werden, welches anschließend konvertiert wird.

Für die Konvertierung des Modells ist die `MapperFacade` zuständig. Diese erhält das visuelle EMF Modell (`editorModel`) und konvertiert es in ein lauffähiges BiCEPS (`bicepsModel`). Mit Hilfe der `MapperFactory` werden die zusätzlich benötigten Instanzen des BiCEPS erstellt, die für die Ausführung notwendig sind (z.B TaskManager).

Zuerst wird das `KnowledgeModel` mit den Eventdefinitionen des visuellen Modells initialisiert. Anschließend werden alle visuellen Elemente (EDT, BiCEP, AEX, EDTBuffer, BiCEPBuffer und AEXBuffer) und deren Eigenschaften über die `create` Methoden konvertiert. Um hier unnötige Konvertierungen zu verhindern, werden alle Elemente in einer Hashmap (`java.util`) gespeichert. Der `key` entspricht den graphischen Elementen des BiCEPS Modells und der `value` den tatsächlichen BiCEPS Elementen. Mittels einem oder mehreren eingehängten Listener (`callback`) wird der Status (Statusänderung, Fehler) der Konvertierung publiziert.

Für das Laden der Klassen anhand ihres Namens wird die Klasse `ClassLoaderUtils` verwendet. Es wird über Reflections die jeweilige spezifizierte Klasse (`classPath`) anhand ihres Namens geladen. Die Objekte werden mit dem Konstruktor der Basisklasse, welche im BiCEPS Framework definiert ist, instanziert.

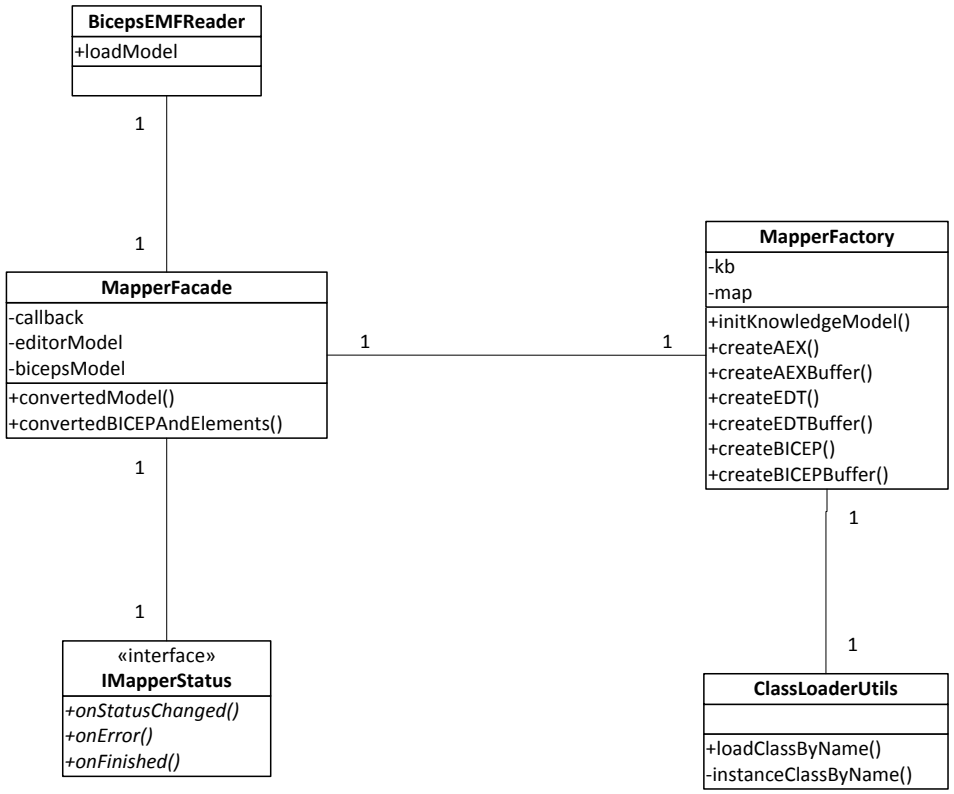

Abbildung 7.3.: Mapper Klassendiagramm

Teil III.

Benutzerhandbuch

8. Tutorial: Erstellung eines BiCEPS mit Eclipse

Dieses Tutorial gibt einen Überblick über die Funktionalität des BiCEPS-Editors. Dafür wird das Fallbeispiel, welches im Kapitel 1.3 angeführt und im Kapitel 2.3 detaillierter beschrieben wurde, verwendet. Es werden EDTs zum Extrahieren von Ereignissen (ZugAnkunft, ZugBuchung, Meeting), ein BiCEP zum Verarbeiten dieser Ereignisse zu komplexen Ereignissen (StarteKollegenAbholung, KollegenAbholung) und ein AEX zum Auslösen von Aktion (sende SMS) erstellt. Alle benötigten Dateien befinden sich im Anhang und werden in den jeweiligen Unterkapiteln referenziert.

Ziel ist es, die Anwendungen und die Funktionalitäten des visuellen BiCEPS-Editors zu veranschaulichen. Dazu wird das Fallbeispiel mit dem Editor modelliert, um es anschließend im BiCEPS Framework auszuführen. Die Eventdaten werden anhand simulierter EDT und AEX Klassen generiert. Als Ergebnis wird ein SMS versendet, wobei diese Aktion mit einer Konsolenausgabe simuliert wird. Um dieses Ziel zu erreichen, sind folgenden Schritte notwendig: Zuerst wird ein Projekt (Kapitel 8.1), in welchem spätere mehrere Diagramme (Kapitel 8.4) für verschiedenen Anwendungsfälle erstellt werden können, angelegt. Die im Diagramm verwendeten Komponenten können vor der Erstellung des Diagramms individuell erweitert werden (Kapitel 8.2) und müssen für jedes Diagramm in der Palette registriert werden (Kapitel 8.3). Sind diese Vorkonfigurationen abgeschlossen, wird das Diagramm erstellt und die Komponenten können von der Palette in den Zeichenbereich (Kapitel 8.5 - 8.7) gezogen und je nach Anwendungsfall konfiguriert werden. Abschließend kann das visuelle Modell über ein Kontextmenü (Kapitel 8.8) ausgeführt werden.

8.1. Erstellen eines Projekts

Um ein BiCEPS zu modellieren, muss zuerst ein entsprechendes BiCEPS-Editor Projekt erstellt werden (Abbildung 8.1). Der Projektwizard kann mittels File→New→Other aufgerufen werden. Weiters muss ein eindeutiger Name für das Projekt vergeben werden (Abbildung 8.2).

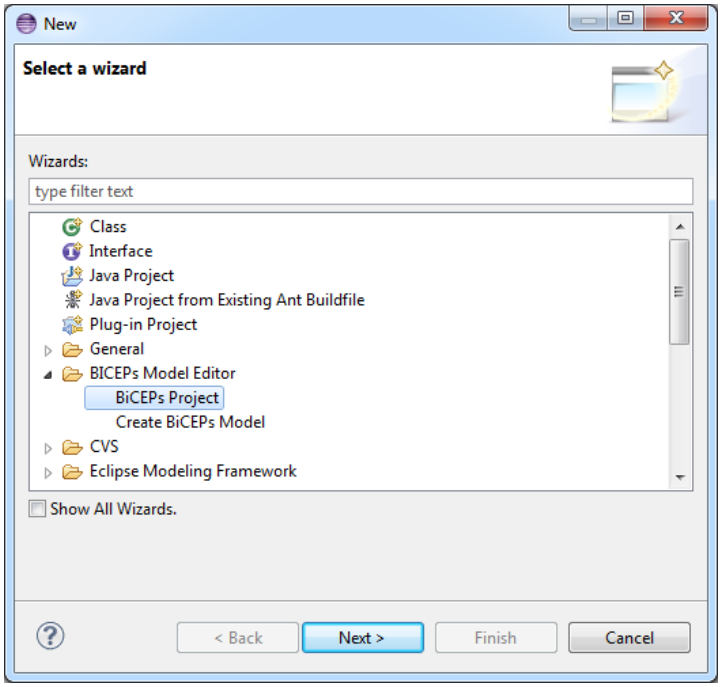

Abbildung 8.1.: Auswahl Projektassistent

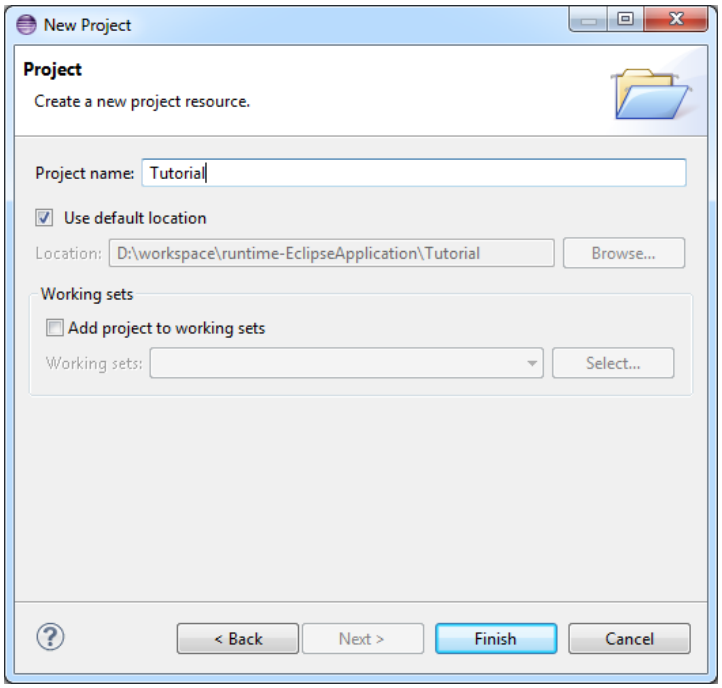

Abbildung 8.2.: Projektnamen auswählen

Die Abbildung 8.3 zeigt die erstellte Projektstruktur. Zukünftig werden alle erstellten Diagramme im Ordner *diagrams* abgelegt. Der *templates* Ordner dient als Referenz für die Erweiterung der Palette.

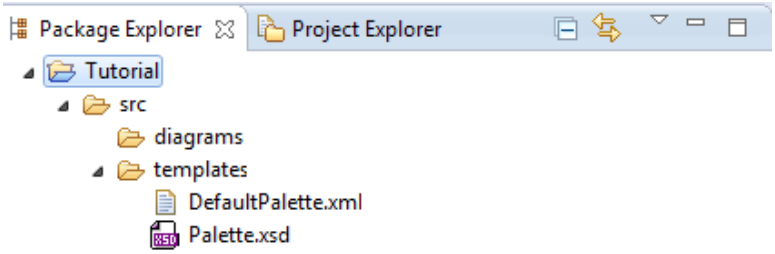

Abbildung 8.3.: Projektstruktur

8.2. Erweitern der BiCEPS Klassen

Im Fallbeispiel (Kapitel 2.3) werden die Eventdaten einer Zugankunft, eines Meetings und einer Zugbuchung von einem Service extrahiert. Diese drei Services werden im Tutorial mit Zufallsgeneratoren simuliert. Für das Tutorial wurden dazu die Klassen `MeetingService`, `ZugAnkunftsService` und `ZugBuchungsService` erstellt. Diese generieren Events sind mit Zufallswerten gefüllt. Auch das SMS Service muss erstellt werden, dazu wird beim Ausführen der Aktion kein SMS gesendet, sondern eine Nachricht auf der Konsole ausgegeben. Die Implementierungen der Klassen befinden sich im Anhang C.3.

8.3. Erweitern des BiCEPS Editor

Die in Kapitel 8.2 erstellten Klassen müssen in der Konfigurationdatei eingetragen werden, um in der Palette des Editors angezeigt zu werden (Kapitel 7.3.2). Dazu müssen die Pfade und Namen der Klassen im XMI File definiert werden (Quellcode 8.1, Zeilen 2-3). Zeilen 5-7 zeigen einen BiCEP, welcher den Namen H2BICEP trägt und `biceps.bicep.BiCEPHandler` verwendet. Bei einem BiCEP müssen zusätzlichen noch der *Taskmanager* und der *Mapper* angegeben werden.
Eine vollständige Abbildung der Palette befindet sich im Anhang unter C.4.

```
1 <?xml version="1.0" encoding="ASCII"?>
2   <possibleElements xsi:type="biceps:EDT" name="
      ZugBuchungService"
3   classPath="biceps.custom.edt.ZugBuchungService">
4
5   <possibleElements xsi:type="biceps:BICEP" name="H2BICEP"
6   classPath="biceps.bicep.BiCEPHandler"
7   classPathTaskManager="biceps.bicep.execution_model.
      H2TaskManager"
8   classPathMapper="biceps.bicep.knowledge_mapper.h2.H2Mapper"
      />
```

Quellcode 8.1: Palette.xml

8.4. Erstellen eines Diagramms

Nach der Konfiguration können diese Informationen (Diagram-Name, DB-Path, Domain-Model, XMI-Palette) beim Erstellen eines neuen Diagramms eingetragen werden. Die Abbildungen 8.4 und 8.5 zeigen die dazu notwendigen Schritte. Der Diagramm Assistent wird über File→New→Other aufgerufen. Anschließend muss ein eindeutiger Name (Diagram-Name) für ein Diagramm gewählt werden. Weiters muss ein Datenbankpfad (DB-Path) für den BiCEP gewählt werden. Wird nur ein Name für die Datenbank vergeben, so wird diese im Projekt im obersten Verzeichnis erstellt. Für die Verwendung externer Komponenten muss die JAR-Datei (Domain-Model) gewählt werden, welche die Klassen mit deren Implementierung enthält. Abschließend muss jene Datei gewählt werden, welche die Palette (XMI-Palette) spezifiziert. Falls das Domain-Model Feld leer gelassen wird, wird die Standardimplementierung von BiCEPS, die im Plugin integriert ist, verwendet.

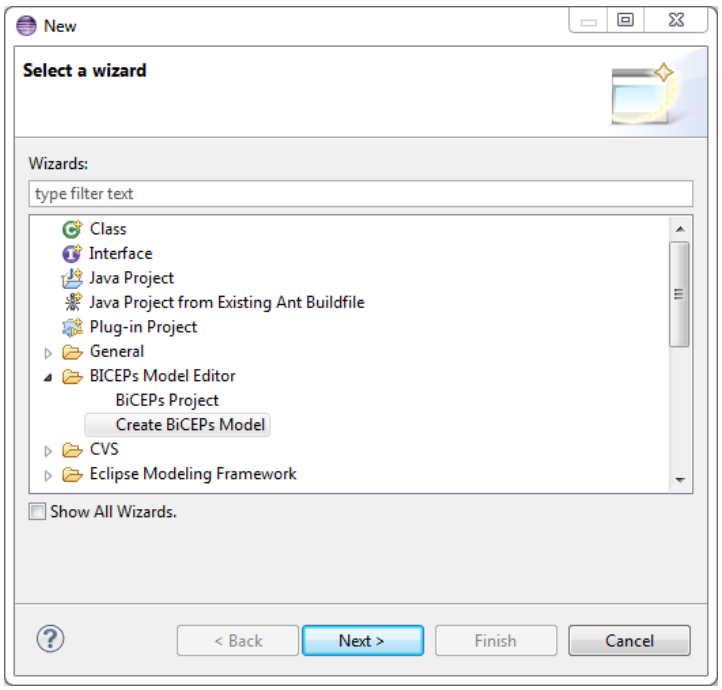

Abbildung 8.4.: Diagramm Assistent

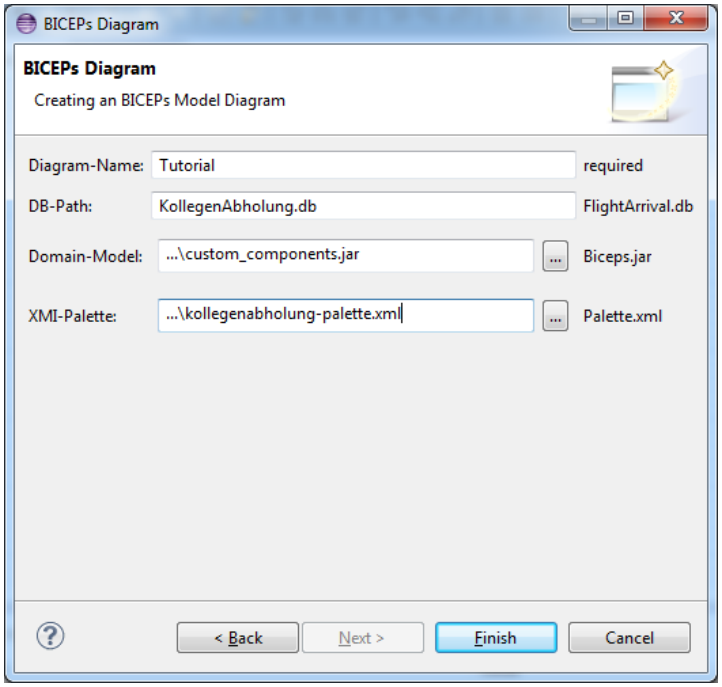

Abbildung 8.5.: Diagramm Assistent Formular

8.5. Erstellen eines EDT

Um einen EDT zu erstellen, wird dieser per Drag & Drop von der Palette in den Zeichenbereich gezogen. Die Parameter, die schon im Kapitel 2.5 beschrieben wurden, können über die Property-Sicht (Abbildung 8.6) eingegeben werden. Über die „Parse“-Schaltfläche wird mittels BiCEPL die eingetragene Eventdefinition validiert.

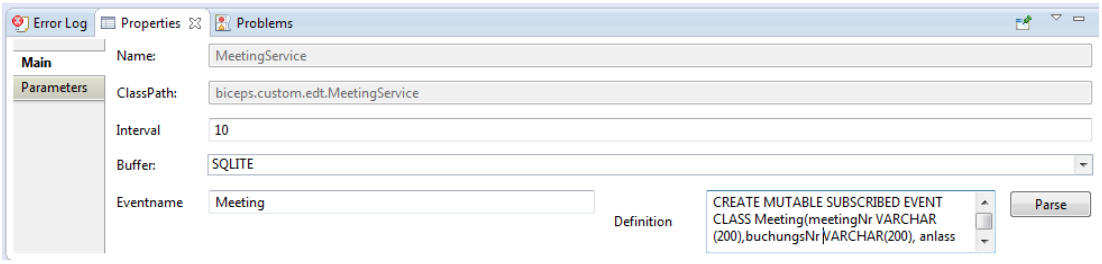

Abbildung 8.6.: Properties EDT

8.6. Erstellen eines AEX

Ein AEX kann wie ein EDT mittels Drag & Drop erstellt werden. Für einen AEX muss ein Buffer ausgewählt werden, in diesem Fall (Abbildung 8.7) ist der *Memory Buffer* in der *Palette.xml* als

Option für einen Buffer vordefiniert (Anhang C.4). Der *Memory Buffer* ist in der Implementation des BiCEPS Frameworks inkludiert und implementiert einen Buffer, welcher seine Informationen im Cache ablegt.

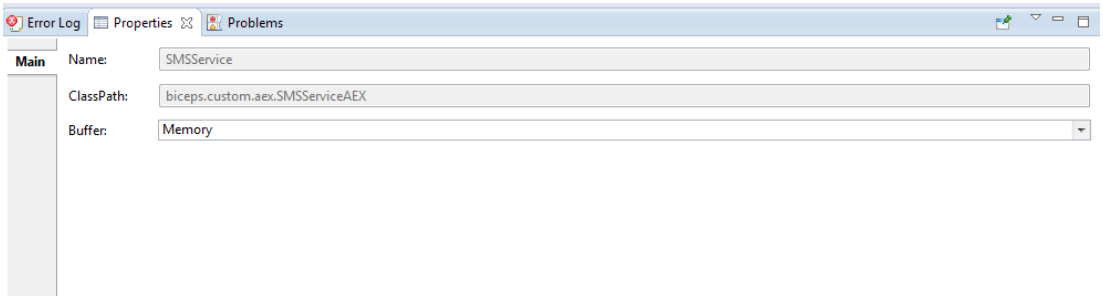

Abbildung 8.7.: Properties AEX

8.7. Erstellen eines BiCEP

Nach der Erstellung eines BiCEP müssen die komplexen Ereignisse definiert werden. Dies zeigt Abbildung 8.8, welche den ComplexEvents Tab in der Property Ansicht zeigt.

Im Beispiel wird ein Complex Event, welches eine FreundAbholung widerspiegelt, eingefügt. Dieses wird beim ersten Aufruf des Events ein Published Event *sendSMS* auslösen. Die Definition kann wieder mittels der „Parse Events“ Schaltfläche überprüft werden.

Die komplexen Ereignisse (Complex Event) werden in einer Tabelle (Table) dargestellt. Jedes Complex Event wird in einer Zeile mit einem Namen und seiner Definition angezeigt. Die EPS Statement und deren Published Events können mittels Drop-down Button angezeigt werden. Über die Schaltflächen *Add ComplexEvent*, *Add EPS in selected CP* und *Add PublishedEvent* können die jeweiligen Zeilen je nach Markierung in der Tabelle eingefügt werden. Als Beispiel kann ein neues Published Event zum Complex Event FreundAbholung hinzugefügt werden, indem das EPS Statement markiert und der *Add PublishedEvent* getätigt wird. Mit dem *Delete* Button können die markierten Zeilen gelöscht werden.

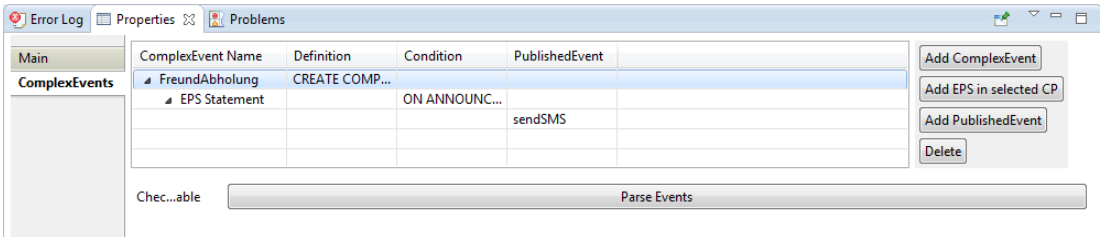

Abbildung 8.8.: Properties BiCEP

Anschließend müssen die Elemente miteinander verbunden werden. Der Verbindungpfeil muss vom ausgehenden Element zum Zielelement gezogen werden. Bei der Verbindung von einem EDT zu einem BiCEP wird automatisch der Eventname über der Verbindungslinie angezeigt.

Die Events, die in einem BiCEP Buffer geschrieben werden, müssen nach der Erstellung der Linie selbst festgelegt werden. Abbildung 8.9 kann mittels Klick auf die Verbindungslinie geöffnet werden. In unserem Fallbeispiel wird nur ein Published Event erstellt.

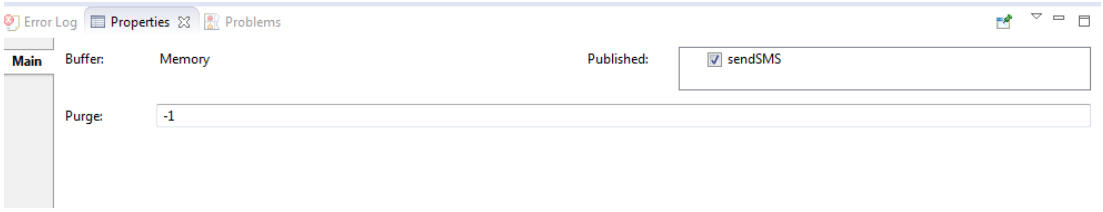

Abbildung 8.9.: Property BiCEP-Buffer

8.8. Ausführung des BiCEPS

Das Ergebnis der oben beschriebenen Schritte entspricht Abbildung 8.10. Weiters wurde eine .bicep Datei erstellt, welche der serialisierten Form der EMF Instanzen entspricht.

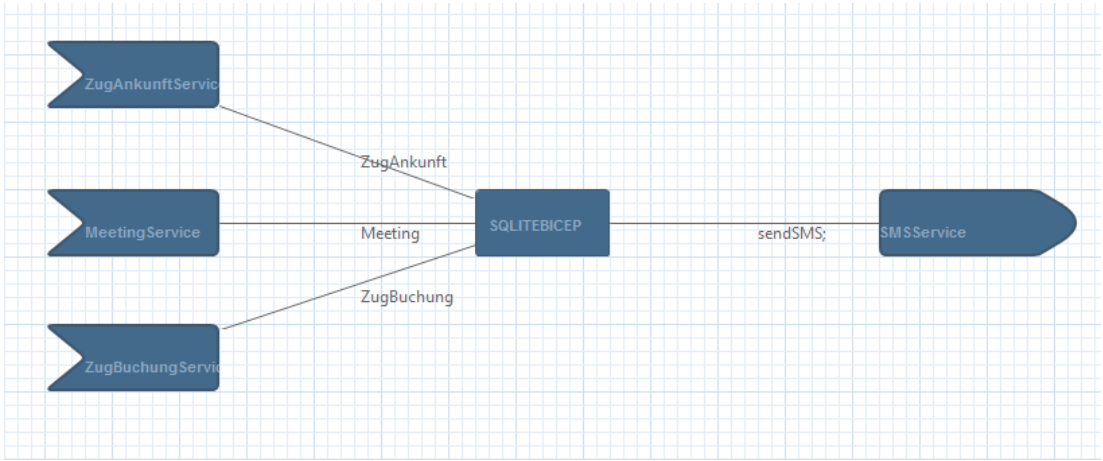

Abbildung 8.10.: Vollständiges BiCEPS Modell

Das erstellte BiCEPS kann über das Kontextmenü mittels *Custom* → *Execute Model* ausgeführt werden. Als Ergebnis (Quellcode: 8.2) wird in der Kommandozeile das Event ausgegeben. Weiters wurde eine Datenbank für jeden Buffer zur Zwischenspeicherung von Ergebnissen angelegt.

```
1 ------------------------------------------------
2 SMS Service was called: {buchungsNr=1, detTime=1395130379,
    ankunftsbahnhof=Linz, occTime=1395130376, meetingNr=0,
    datum=12.03.2014, zugNr=2}
3 ------------------------------------------------
```

Quellcode 8.2: Konsolenausgabe

9. Zusammenfassung

Diese Arbeit beschäftigt sich mit der Entwicklung eines visuellen Editors zur Erstellung eines Bitemporal Event Processing System (BiCEPS) in Eclipse, welcher es ermöglicht, ein solches System in einer Drag & Drop-Umgebung zu modellieren und anschließend auszuführen. Ein BiCEPS integriert das Extrahieren von Ereignissen (mittels Event Detectors), die Verarbeitung dieser zu komplexen Ereignissen (mittels Bitemporal Complex Event Processors), und das Ausführen von Aktionen (mittels Action Executors). Im Gegensatz zu Ereignissen in herkömmlichen CEP Systemen, weisen Ereignisse in BiCEPS zwei Zeitdimensionen auf, neben dem Eintrittszeitpunkt eines Events wird sein Wahrnehmungszeitpunkt definiert. Des weiteren unterstützt BiCEPS die Verarbeitung von veränderlichen Ereignissen und das Ausführen von Aktionen wenn Ereignisse passieren oder wenn diese sich ändern. Zudem wird in dieser Arbeit die Eclipse Platform erläutert und es werden Frameworks (GEF, GMF und Graphiti) zur Erstellung eines visuellen Editors in Eclipse vorgestellt. Nach diesen Grundlagen wird basierend auf den Anforderungen des BiCEPS-Editors eine Evaluierung der Frameworks, mit dem Ziel das am besten dafür geeignete Framework zu finden, durchgeführt. Graphiti deckt alle Anforderungen des BiCEPS-Editors ab, ist individuell erweiterbar und noch dazu einfach zu erlernen, im Gegensatz zu GEF und GMF. Unter Berücksichtigung des Evaluierungsergebnisses wird das Implementierungsdesign des Editors entworfen. Dieses Design beschreibt den Aufbau des Editors mit seinen Modulen und der verwendeten Technologien. Das Design sieht eine Implementierung des Editors in drei Eclipse Plug-ins (BiCEPSModel, BiCEPSEditor und BiCEPSMapper) vor. Das BiCEPSModel wird in EMF implementiert; als Basis wird das bereits bestehende Java-basierte BiCEPS Framework herangezogen. Der BiCEPSEditor ist für die visuelle Darstellung der Komponenten eines BiCEPS zuständig und wird mit dem Graphiti Framework implementiert. Die BiCEPS Komponenten und deren Funktionalitäten, die im Editor verwendet werden, können ohne Änderung der Implementierung über eine XMI Schnittstelle konfiguriert werden. Dies ermöglicht, dass der BiCEPSEditor unabhängig vom BiCEPS Framework eingesetzt und von außen angepasst werden kann. Der BiCEPSMapper ist eine Java Anwendung welche das visuelle BiCEPS Modell in ein ausführbares Programm transformiert. Zum Schluss dieser Arbeit wird in Form eines Benutzerhandbuchs das Erstellen eines BiCEPS mit dem entwickelten Editor gezeigt.

A. Literaturverzeichnis

Balzert, H. (2009). Anforderungen und Anforderungsarten. In *Lehrbuch der Softwaretechnik: Basiskonzepte und Requirements Engineering* (S. 455-474). Spektrum Akademischer Verlag. Zugriff auf `http://dx.doi.org/10.1007/978-3-8274-2247-7_16` doi: 10.1007/978-3-8274-2247-7_16

Bilnoski, P. (2010). *Starting Out with Eclipse GEF.* Zugriff auf `http://pbwhiteboard.blogspot.co.at/2010/12/notes-for-starting-out-with-eclipse-gef.html` (zuletzt zugegriffen am 05.02.2014)

Brand, C., Gorning, M., Kaiser, T., Pasch, J. & Wenz, M. (2011). Graphiti - Development of High-Quality Graphical Model Editors. *Eclipse Magazin*.

Dave, S. (2008). *Fundamentals of the Eclipse Modeling Framework.* Zugriff auf `http://www.eclipse.org/modeling/emf/docs/presentations/EclipseCon/EclipseCon2008_309T_Fundamentals_of_EMF.pdf` (zuletzt zugegriffen am 05.02.2014)

Demers, A. J., Gehrke, J., Panda, B., Riedewald, M., Sharma, V. & White, W. M. (2007). Cayuga: A general purpose event monitoring system. In *Third Biennial Conference on Innovative Data Systems Research (CIDR 2007)* (S. 412-422).

Eckert, M. & Bry, F. (2009). Complex Event Processing (CEP). *Informatik-Spektrum*, *32* (2), 163–167. doi: 10.1007/s00287-009-0329-6

Eclipse Documentation, W. (2013). *Eclipse property view documentation.* Zugriff auf `http://help.eclipse.org/juno/index.jsp?topic=2Forg.eclipse.platform.doc.user2FconceptsFcpropview.html` (zuletzt zugegriffen am 10.03.2014)

Eclipse EMF Documentation, W. (2013). *Eclipse online documentation about eclipse modeling framework.* Zugriff auf `http://help.eclipse.org/kepler/nav/18` (zuletzt zugegriffen am 05.02.2014)

Eclipse Foundation JFace. (2014). *JFace.* Zugriff auf `http://www.eclipse.org/jface/` (zuletzt zugegriffen am 05.02.2014)

Eclipse Modeling Framework, W. (2013). *Online documentation about eclipse modeling framework.* Zugriff auf `http://www.eclipse.org/modeling/emf/docs/` (zuletzt zugegriffen am 05.02.2014)

Eclipse Plattform, W. (2012). *Eclipse platform architecture.* Zugriff auf `http://help.eclipse.org/indigo/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2Farch.htm` (zuletzt zugegriffen am 05.02.2014)

Furche, T., Gottlob, G., Grasso, G., Schallhart, C. & Sellers, A. (2011). Oxpath: A language for scalable, memory-efficient data extraction from web applications. *Proceedings of the VLDB Endowment*, *4* (11), 1016–1027.

Furche, T., Grasso, G., Huemer, M., Schallhart, C. & Schrefl, M. (2013a). Bitemporal complex event processing of web event advertisements. In *14th International Conference on Web Information Systems Engineering (WISE 2013)* (S. 333–346). Springer.

Furche, T., Grasso, G., Huemer, M., Schallhart, C. & Schrefl, M. (2013b). Peace-ful web event extraction and processing. In X. Lin, Y. Manolopoulos, D. Srivastava & G. Huang (Hrsg.), *Web Information Systems Engineering (WISE 2013)* (Bd. 8181, S. 523-526). Springer Berlin Heidelberg.

GMF, E. F. (2014). *Graphical modeling framework documentation.* Zugriff auf `http://wiki.eclipse.org/Graphical_Modeling_Framework/Tutorial/Part_4` (zuletzt zugegriffen am 07.02.2014)

Graphiti, E. F. (2014). *Online eclipse documentation of graphiti.* Zugriff auf `http://help.eclipse.org/kepler/index.jsp?nav=%2F25` (zuletzt zugegriffen am 07.02.2014)

Gronback, R. C. (2009). *Eclipse modeling project: a domain-specific language (DSL) toolkit.* Pearson Education.

Jacobsen, H.-A., Cheung, A. K. Y., Li, G., Maniymaran, B., Muthusamy, V. & Kazemzadeh, R. S. (2010). The PADRES publish/subscribe system. In *Principles and applications of distributed event-based systems* (S. 164-205).

Li, G. & Jacobsen, H.-A. (2005). Composite subscriptions in content-based publish/subscribe systems. In *Proceedings of the 6th international middleware conference (middleware 2006)* (S. 249-269).

Luckham, D. C. (1998). Rapide: A language and toolset for causal event modeling of distributed system architectures. In *Proceedings of the second international conference on worldwide computing and its applications (wwca 1998)* (S. 88-96).

Luckham, D. C. (2002). *The power of events - an introduction to complex event processing in distributed enterprise systems* (Bd. 204). Addison-Wesley Reading.

Luckham, D. C. & Vera, J. (1995). An event-based architecture definition language. *IEEE Transactions on Software Engineering*, *21* (9), 717-734.

Moore, B., Dean, D., Gerber, A., Wagenknecht, G. & Vanderheyden, P. (2004). *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework.* Red Books.

Object Management Group, W. (2014). *Object Management Group (OMG) documentation.* Zugriff auf `http://www.omg.org/mof/` (zuletzt zugegriffen am 05.02.2014)

Refsdal, I. (2011). Comparison of gmf and graphiti based on experiences from the development of th prediqt tool. *Master's thesis - University of Oslo*.

Rivieres, J. & Wiegand, J. (2004). Eclipse: A platform for integrating development tools. *IBM Systems Journal*, *43* (2), 371–383.

Schultz-Møller, N. P., Migliavacca, M. & Pietzuch, P. R. (2009). Distributed complex event processing with query rewriting. In *Proceedings of the third acm international conference on distributed event-based systems (debs 2009).*

Srivastava, U. & Widom, J. (2004). Flexible time management in data stream systems. In *Proceedings of the twenty-third acm sigact-sigmod-sigart symposium on principles of database systems (pods 2004)* (S. 263-274).

Steinberg, D., Budinsky, F., Merks, E. & Paternostro, M. (2008). *EMF: Eclipse Modeling Framework* (Second Aufl.). Pearson Education.

SWT, E. F. (2014). *Standard widget tookit.* Zugriff auf `http://www.eclipse.org/swt/` (zuletzt zugegriffen am 05.02.2014)

Teneo, W. (2013). *Eclipse Teneo.* Zugriff auf `http://wiki.eclipse.org/Teneo` (zuletzt zugegriffen am 05.03.2014)

Top Cased, Website. (2013). *Top cased software.* Zugriff auf `http://www.topcased.org/` (zuletzt zugegriffen am 05.03.2014)

Uwe Kloos, G. T., Natividad Martinex. (2012). *Informatics Inside 2012: Reality++ - Tomorrow comes today!* Hochschule Reutlingen. Zugriff auf `http://books.google.at/books?id=k5bJODQ6cOkC`

Wenz, M. (2012). The graphical tooling infrastructure speaking plain java. SAP AG.

Wu, E., Diao, Y. & Rizvi, S. (2006). High-performance complex event processing over streams. In *Proceedings of the acm sigmod international conference on management of data (acm sigmod conference 2006)* (S. 407-418).

B. Abbildungsverzeichnis

C. Anhang

C.1. EclipseEditor Manifest

Der Quellcode C.1 zeigt die Manifest Datei des Editors (BiCEPSEditor). Hier werden die Informationen (Version, Name, etc.) des Plugins spezifiziert. Auch die benötigen externen Plugins (Required-Bundles) werden hier beschrieben.

```
1  Manifest-Version: 1.0
2  Bundle-ManifestVersion: 2
3  Bundle-Name: BicepsEditor
4  Bundle-SymbolicName: BicepsEditor;singleton:=true
5  Bundle-Version: 1.0.0.qualifier
6  Bundle-Activator: bicepseditor.Activator
7  Require-Bundle: org.eclipse.ui,
8   org.eclipse.core.runtime,
9   org.eclipse.graphiti,
10  org.eclipse.graphiti.ui,
11  org.eclipse.core.resources,
12  org.eclipse.ui.views.properties.tabbed,
13  org.eclipse.emf.transaction;bundle-version="1.4.0",
14  org.eclipse.swt,
15  org.eclipse.graphiti.mm,
16  org.eclipse.equinox.registry,
17  org.eclipse.ui.views,
18  org.eclipse.emf.ecore,
19  BICEPSEMM,
20  org.eclipse.jface,
21  org.eclipse.ui.workbench,
22  org.eclipse.gef;bundle-version="3.9.0"
23 Bundle-RequiredExecutionEnvironment: JavaSE-1.7
24 Bundle-ActivationPolicy: lazy
25 Import-Package: org.eclipse.emf.transaction,
26  org.eclipse.ui.actions,
27  org.eclipse.ui.ide,
28  org.eclipse.ui.wizards.newresource
```

Quellcode C.1: MANIFEST.ML

C.2. Plugin.xml

Der Quellcode C.2 zeigt die Plugin.xml Datei des visuellen Editors (BiCEPSEditor). In dieser werden die Module (Wizard, Graphiti Komponenten, Property View, etc.) und die dazu-

gehörigen Klassen definiert. Die Module werden in Kapitel 7.3 beschrieben.

```
11 <?eclipse version="3.4"?>
12 <plugin>
13    <extension
14          point="org.eclipse.graphiti.ui.diagramTypes">
15       <diagramType>
16             name="BicepsEditor Diagram Type"
17             type="BicepsEditor"
18             id="BicepsEditor.BicepsEditorDiagramType">
19       </diagramType>
20    </extension>
21    <extension
22          point="org.eclipse.graphiti.ui.diagramTypeProviders">
23       <diagramTypeProvider
24             name="BicepsEditor Diagram Type Provider"
25             class="bicepseditor.diagram.
                   BicepsEditorDiagramTypeProvider"
26             id="BicepsEditor.BicepsEditorDiagramTypeProvider">
27          <diagramType
28                id="BicepsEditor.BicepsEditorDiagramType">
29          </diagramType>
30            <imageProvider
31             id="BicepsEditor.BicepsEditorImageProvider">
32       </imageProvider>
33       </diagramTypeProvider>
34
35    </extension>
36
37          <extension
38                point="org.eclipse.ui.newWizards">
39             <category
40                   id="BicepsEditor.category.wizards"
41                   name="BICEPs Model Editor">
42             </category>
43             <wizard
44                   category="BicepsEditor.category.wizards"
45                   class="bicepseditor.wizard.project.
                         ProjectWizard"
46                   id="BicepsEditor.wizard.project"
47                   name="BiCEPs Project">
48             </wizard>
49             <wizard
50                   category="BicepsEditor.category.wizards"
51                   class="bicepseditor.wizard.diagram.
                         DiagramWizard"
52                   id="BicepsEditor.wizard.diagram"
53                   name="Create BiCEPs Model"
54                   project="false">
```

```
55              </wizard>
56          </extension>
57
58
59
60      <extension
61         point="org.eclipse.graphiti.ui.imageProviders">
62      <imageProvider
63            class="bicepseditor.diagram.BicepsEditorImageProvider
                "
64            id="BicepsEditor.BicepsEditorImageProvider">
65      </imageProvider>
66  </extension>
67
68          <extension
69             point="org.eclipse.ui.views.properties.tabbed.
                propertyContributor">
70             <propertyContributor contributorId="BicepsEditor.
                PropertyContributor">
71                <propertyCategory category="Graphiti">
72                </propertyCategory>
73             </propertyContributor>
74          </extension>
75
76          <extension
77             point="org.eclipse.ui.views.properties.tabbed.
                propertyTabs">
78             <propertyTabs contributorId="BicepsEditor.
                PropertyContributor">
79                <propertyTab label="Main" category="Graphiti"
80                   id="graphiti.main.tab">
81                </propertyTab>
82
83                   <propertyTab label="Parameters" category="
                      Graphiti"
84                   id="graphiti.parameters.tab"
85                   afterTab="BicepsEditor.main">
86                      </propertyTab>
87                     <propertyTab label="ComplexEvents"
                         category="Graphiti"
88                   id="graphiti.complexevents.tab"
89                   afterTab="BicepsEditor.main">
90                </propertyTab>
91
92             </propertyTabs>
93          </extension>
94
95          <extension
96             point="org.eclipse.ui.views.properties.tabbed.
```

```
            propertySections">
97
98          <!-- GENERAL ELEMENTS -->
99          <propertySections contributorId="BicepsEditor.
            PropertyContributor">
100          <propertySection tab="graphiti.main.tab"
101           class="bicepseditor.diagram.properties.
                 GENERALNameSectionProperty"
102           filter="bicepseditor.diagram.properties.filter.
                 GeneralPropertiesFilter"
103           id="graphiti.main.tab.main.name">
104          </propertySection>
105
106          <propertySection tab="graphiti.main.tab"
107           class="bicepseditor.diagram.properties.
                 GENERALClassPathSectionProperty"
108           filter="bicepseditor.diagram.properties.filter.
                 GeneralPropertiesFilter"
109           id="graphiti.main.tab.main.classpath"
110           afterSection="graphiti.main.tab.main.name">
111          </propertySection>
112         <!-- GENERAL BUFFER ELEMENTS -->
113             <propertySection tab="graphiti.main.tab"
114           class="bicepseditor.diagram.properties.
                 GENERALBUFFERPurgeProperty"
115           filter="bicepseditor.diagram.properties.filter.
                 GeneralBufferProperties"
116           id="graphiti.main.tab.main.purge">
117          </propertySection>
118         <!-- EDT  -->
119          <propertySection tab="graphiti.main.tab"
120           class="bicepseditor.diagram.properties.
                 EDTIntervalProperty"
121           filter="bicepseditor.diagram.properties.filter.
                 EDTPropertiesFilter"
122           id="graphiti.main.tab.main.interval"
123           afterSection="graphiti.main.tab.main.classpath"
                 >
124          </propertySection>
125           <propertySection tab="graphiti.main.tab"
126           class="bicepseditor.diagram.properties.
                 EDTBufferSelectionProperty"
127           filter="bicepseditor.diagram.properties.filter.
                 EDTPropertiesFilter"
128           id="graphiti.main.tab.main.buffer"
129           afterSection="graphiti.main.tab.main.interval">
130          </propertySection>
131
132           <propertySection tab="graphiti.main.tab"
```

```
133                 class="bicepseditor.diagram.properties.
                        EDTEventDefinition"
134                 filter="bicepseditor.diagram.properties.filter.
                        EDTPropertiesFilter"
135                 id="graphiti.main.tab.main.edtevent"
136                 afterSection="graphiti.main.tab.main.buffer">
137               </propertySection>
138
139               <propertySection
140                   class="org.eclipse.ui.views.properties.tabbed
                          .AdvancedPropertySection"
141                   filter="bicepseditor.diagram.properties.
                          filter.EDTPropertiesFilter"
142                   id="graphiti.parameters.tab.param"
143                   tab="graphiti.parameters.tab">
144               </propertySection>
145
146       <!-- BICEP -->
147       <propertySection tab="graphiti.main.tab"
148                 class="bicepseditor.diagram.properties.
                        BICEPChrononProperty"
149                 filter="bicepseditor.diagram.properties.filter.
                        BICEPPropertiesFilter"
150                 id="graphiti.main.tab.main.chronon"
151                 afterSection="graphiti.main.tab.main.classpath"
                        >
152               </propertySection>
153
154                <propertySection tab="graphiti.main.tab"
155                 class="bicepseditor.diagram.properties.
                        BICEPBufferConnectionProperty"
156                 filter="bicepseditor.diagram.properties.filter.
                        BICEPConnectionsPropertiesFilter"
157                 id="graphiti.main.tab.main.connection" >
158               </propertySection>
159
160
161                <propertySection tab="graphiti.complexevents.
                        tab"
162                 class="bicepseditor.diagram.properties.
                        BICEPComplexEventTable"
163                 filter="bicepseditor.diagram.properties.filter.
                        BICEPPropertiesFilter"
164                 id="graphiti.main.tab.main.event" >
165               </propertySection>
166
167                <propertySection tab="graphiti.complexevents.
                        tab"
168                 class="bicepseditor.diagram.properties.
```

```
                    BICEPComplexEventCheck"
169                 filter="bicepseditor.diagram.properties.filter.
                    BICEPPropertiesFilter"
170                 id="graphiti.main.tab.main.biceplparser"
171                 afterSection="graphiti.main.tab.main.event" >
172                </propertySection>
173
174
175          <!-- AEX -->
176
177          <propertySection tab="graphiti.main.tab"
178                 class="bicepseditor.diagram.properties.
                    AEXBufferSelectionProperty"
179                 filter="bicepseditor.diagram.properties.filter.
                    AEXPropertiesFilter"
180                 id="graphiti.main.tab.main.aex"
181                 afterSection="graphiti.main.tab.main.classpath"
                    >
182                </propertySection>
183
184                <!-- Filter for AEX Connection/Buffer -->
185                      <propertySection tab="graphiti.main.tab"
186                 class="bicepseditor.diagram.properties.
                    AEXBufferConnectionProperty"
187                 filter="bicepseditor.diagram.properties.filter.
                    SimpleConnectionsPropertiesFilter"
188                 id="graphiti.main.tab.main.connection" >
189                </propertySection>
190
191
192             </propertySections>
193          </extension>
194
195
196
197             <extension
198           point="org.eclipse.core.runtime.adapters">
199                    <factory
200                           adaptableType="org.eclipse.graphiti.ui
                               .platform.GraphitiShapeEditPart"
201                           class="bicepseditor.diagram.properties
                               .params.
                               ParameterPropertyAdapterFactory">
202                           <adapter
203                                  type="org.eclipse.ui.views.
                                      properties.IPropertySource"
                                      >
204                           </adapter>
205                    </factory>
```

```
206         </extension>
207
208
209
210
211
212
213 </plugin>
```

Quellcode C.2: Plugin.xml

C.3. Erweiterte BICEP Klassen

In diesem Kapitel des Anhangs werden die Quellcodes der erweiterten BiCEPS Komponenten, die für die Umsetzung des Fallbeispiels (Kapitel 1.3) benötigt werden, aufgelistet.

Quellcode C.3 zeigt jene Klasse, die einen EDT für ein Meeting (Kapitel 2.3) implementiert. Quellcode C.4 und C.5 zeigen die Implementierung eines EDT der Zugbuchung bzw. der Zugankunft. Diese drei Services generieren mittels einem Zufallsgenator (Quellcode C.6) Events. Quellcode C.7 zeigt den erweiterten AEX, der ein SMS Service repräsentiert.

```
1  package biceps.custom.edt;
2
3  import java.util.ArrayList;
4  import java.util.HashMap;
5
6  import biceps.event.Event;
7  import biceps.event_detector.EDTTask.EventDetectionTask;
8  import biceps.mapper.util.SimulatorUtil;
9
10 public class MeetingService extends EventDetectionTask{
11
12
13   private static final String et = "Meeting";
14   private static final String[] att = {"meetingNr","buchungsNr
       ","anlass","personNamen","braucheAbholung","occTime"};
15
16   @Override
17   public ArrayList<Event> performTask(HashMap<String, String>
       parameterMap) {
18     ArrayList<Event> eventList = new ArrayList<Event>();
19     long occTime = System.currentTimeMillis()/1000L;
20
21     for(int i=0; i<13; i++) //on Time
22       eventList.add(SimulatorUtil.createEvent(et,att,new
           String[]{""+i,"1","hochzeit","daniel","JA",""+occTime
           }));
23
```

```
24
25
26      return eventList;
27    }
28  }
```

Quellcode C.3: MeetingService.java

```
1
2  package biceps.custom.edt;
3
4  import java.util.ArrayList;
5  import java.util.HashMap;
6
7  import biceps.event.Event;
8  import biceps.event_detector.EDTTask.EventDetectionTask;
9  import biceps.mapper.util.SimulatorUtil;
10
11
12
13 public class ZugBuchungService extends EventDetectionTask{
14
15
16   private static final String et = "ZugBuchung";
17   private static final String[] att = {"buchungsNr","zugNr","
       datum","abfahrtsbahnhof","ankunftsbahnhof","occTime"};
18
19   @Override
20   public ArrayList<Event> performTask(HashMap<String, String>
       parameterMap) {
21     ArrayList<Event> eventList = new ArrayList<Event>();
22     long occTime = System.currentTimeMillis()/1000L;
23
24     for(int i=0; i<3; i++) //on Time
25       eventList.add(SimulatorUtil.createEvent(et,att,new
           String[]{""+i,"2","12.03.2014","Wien","Linz",""+
           occTime}));
26
27
28
29     return eventList;
30   }
31
32 }
```

Quellcode C.4: ZugBuchungService.java

```
1  package biceps.custom.edt;
2
```

```java
import java.util.ArrayList;
import java.util.HashMap;

import biceps.event.Event;
import biceps.event_detector.EDTTask.EventDetectionTask;
import biceps.mapper.util.SimulatorUtil;

public class ZugAnkunftService extends EventDetectionTask {

  private static final String et = "ZugAnkunft";
  private static final String[] att = {"zugNr","datum","
     bahnhof","occTime"};

  @Override
  public ArrayList<Event> performTask(HashMap<String, String>
     parameterMap) {
    ArrayList<Event> eventList = new ArrayList<Event>();
    long occTime = System.currentTimeMillis()/1000L;

    for(int i=0; i<13; i++) //on Time
      eventList.add(SimulatorUtil.createEvent(et,att,new
         String[]{""+i,"12.03.2014", "linz",""+occTime}));

    return eventList;
  }
}
```

Quellcode C.5: ZugAnkunftService.java

```java
package biceps.mapper.util;

import biceps.event.Event;
import biceps.event.SimpleAttribute;
import biceps.event.SimpleEventClass;

public class SimulatorUtil {

  public static Event createEvent(String name, String[] att,
     String[] value){
    Event e = new Event();
    e.setEventClass(new SimpleEventClass(name));
    for(int i=0; i<att.length; i++){
      e.addAttribute(new SimpleAttribute(att[i],"",value[i]));
    }
    return e;
  }
}
```

Quellcode C.6: SimulatorUtil.java

```java
package biceps.custom.aex;

import java.util.HashMap;

import biceps.BiCEPSException;
import biceps.action_executor.ActionExecutor;

public class SMSServiceAEX extends ActionExecutor{

  @Override
  public void execute(HashMap<String, String> attributes)
      throws BiCEPSException {
    System.out.println("
       ----------------------------------------------------");
    System.out.println("SMS Service was called: " + attributes
       .toString());
    System.out.println("
       ----------------------------------------------------");
  }
}
```

Quellcode C.7: SMSServiceAEX.java

C.4. Fallbeispiel Paletten XMI

Der Quellcode C.8 zeigt die Konfiguration der Palette für das Fallbeispiel (Kapitel 1.3) in einer XMI Datei an. Es werden hier die erweiterten Komponenten (Kapitel C.3) registriert.

```
<possibleElements xsi:type="biceps:EDT" name="
   ZugAnkunftService" classPath="biceps.custom.edt">
 <parameterList key="zugNr"/>
</possibleElements>

  <possibleElements xsi:type="biceps:EDT" name="
     MeetingService" classPath="biceps.custom.edt.
     MeetingService">
</possibleElements>

<possibleElements xsi:type="biceps:EDT" name="
   ZugBuchungService" classPath="biceps.custom.edt.
   ZugBuchungService">
 <parameterList key="zugNr"/>
</possibleElements>
```

```
229   <possibleElements xsi:type="biceps:AEX" name="SMSService"
          classPath="biceps.custom.aex.SMSServiceAEX">
230   <parameterList key="handynummer"/>
231      </possibleElements>
232
233
234   <possibleElements xsi:type="biceps:BICEP" name="SQLITEBICEP"
           classPath="biceps.bicep.BiCEPHandler"
          classPathTaskManager="biceps.bicep.execution_model.
          SQLiteTaskManager" classPathMapper="biceps.bicep.
          knowledge_mapper.sqLite.SQLiteMapper"/>
235   <possibleElements xsi:type="biceps:BICEP" name="H2BICEP"
          classPath="biceps.bicep.BiCEPHandler"
          classPathTaskManager="biceps.bicep.execution_model.
          H2TaskManager" classPathMapper="biceps.bicep.
          knowledge_mapper.h2.H2Mapper"/>
236
237   <possibleAEXBuffer name="Memory" classpath="biceps.
          event_buffer.aexBuffer.AEXBufferManager"/>
238   <possibleEDTBuffer name="SQLITE" classpath="biceps.
          event_buffer.edtBuffer.EDTSQLiteBuffer"/>
239
240
241 </biceps:InitialModel>
```

Quellcode C.8: Palette.xml

C.5. Serialisiertes BiCEPS

Quellcode C.9 beschreibt das serialisierte BiCEPS, welches mit dem visuellen Editor erstellt wird. Diese Serialisierung wird in einer XMI Datei spezifiziert.

```
1  <?xml version="1.0" encoding="ASCII"?>
2  <biceps:BICEPS_MODEL
3  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4  xmlns:biceps="http://www.mod4j.org/biceps" dbPath="Abholung.db
       "
5  modelPath="\Tutorial\src\templates\bicep.jar"
6  picoPath="C:\Users\seb\workspace2\BicepsMapper\freundabholung-
       palette.xml">
7
8    <elementList xsi:type="biceps:EDT" name="ZugAnkunftService"
9    classPath="biceps.custom.edt.ZugAnkunftService"
10   subscribedBiCEP="//@elementList.3" interval="10.0">
11     <detector name="SQLITE"
12     classpath="biceps.event_buffer.edtBuffer.EDTSQLiteBuffer"
           /> <subscribedEvent
```

```
13      name="ZugAnkunft" definition="CREATE MUTABLE SUBSCRIBED
            EVENT CLASS
14      ZugAnkunft(zugNr VARCHAR(200), datum VARCHAR(200), bahnhof
            VARCHAR(200)) ID
15      (zugNr, datum) LIFESPAN (2d);"/> <parameterList key="zugNr
            " value="2"/>
16    </elementList>
17
18    <elementList xsi:type="biceps:EDT" name="MeetingService"
19    classPath="biceps.custom.edt.MeetingService"
20    subscribedBiCEP="//@elementList.3" interval="10.0">
21      <detector name="SQLITE"
22      classpath="biceps.event_buffer.edtBuffer.EDTSQLiteBuffer"
            /> <subscribedEvent
23      name="Meeting" definition="CREATE MUTABLE SUBSCRIBED EVENT
            CLASS
24      Meeting(meetingNr VARCHAR(200),buchungsNr VARCHAR(200),
            anlass VARCHAR(200),
25      personNamen VARCHAR(200), braucheAbholung VARCHAR(200)) ID
            (meetingNr)
26      LIFESPAN (2d);"/>
27    </elementList>
28
29    <elementList xsi:type="biceps:EDT" name="ZugBuchungService"
30    classPath="biceps.custom.edt.ZugBuchungService"
31    subscribedBiCEP="//@elementList.3" interval="10.0">
32      <detector name="SQLITE"
33      classpath="biceps.event_buffer.edtBuffer.EDTSQLiteBuffer"
            /> <subscribedEvent
34      name="ZugBuchung" definition="CREATE MUTABLE SUBSCRIBED
            EVENT CLASS
35      ZugBuchung(buchungsNr VARCHAR(200),zugNR VARCHAR(200),
            datum VARCHAR(200),
36      abfahrtsbahnhof VARCHAR(200),ankunftsbahnhof VARCHAR(200))
            ID (buchungsNr)
37      LIFESPAN (2d);"/> <parameterList key="zugNr" value="2"/>
38    </elementList>
39
40    <elementList xsi:type="biceps:BICEP" name="SQLITEBICEP"
41    classPath="biceps.bicep.BiCEPHandler" hasEvents="//
          @eventList.0"
42    registeredEDT="//@elementList.0 //@elementList.1 //
          @elementList.2"
43    registeredAEX="//@elementList.4"
44    classPathMapper="biceps.bicep.knowledge_mapper.sqLite.
          SQLiteMapper"
45    classPathTaskManager="biceps.bicep.execution_model.
          SQLiteTaskManager"
46    chronon="3"/> <elementList xsi:type="biceps:AEX" name="
```

```
      SMSService"
47    classPath="biceps.custom.aex.SMSServiceAEX"
48    publishedEvent="//@eventList.0/@epsStatements.0/
         @publishedEvents.0"
49    registeredBiCEP="//@elementList.3">
50      <executor name="Memory"
51      classpath="biceps.event_buffer.aexBuffer.AEXBufferManager"
           /> <parameterList
52      key="handynummer"/>
53    </elementList>
54
55    <eventList xsi:type="biceps:ComplexEventClass"
56    name="FreundAbholung" definition="CREATE COMPLEX EVENT CLASS
         FreundAbholung
57    (zugNr VARCHAR(200),datum VARCHAR(200),buchungsNr VARCHAR
         (200), meetingNr
58    VARCHAR(200),ankunftsbahnhof VARCHAR(200)) ID
59    (zugNr,datum,buchungsNr,meetingNr) AS SELECT an.zugNr,an.
         datum,zb.buchungsNr,
60    me.meetingNr, zb.ankunftsbahnhof FROM ZugAnkunft an,
         ZugBuchung zb, Meeting me
61    WHERE &#x9;an.zugNr=zb.zugNr AND an.datum=zb.datum AND
62    &#x9;zb.buchungsNr=me.buchungsNr OCCURRING AT an.occTime ON
         ANNOUNCEMENT DO
63    sendSMS();">
64      <epsStatements condition="ON ANNOUNCEMENT">
65        <publishedEvents name="sendSMS"/>
66      </epsStatements>
67    </eventList>
68  </biceps:BICEPS_MODEL>
```

Quellcode C.9: Tutorial.bicep