Zürcher Hochschule Winterthur

Dozent: E. Mund

Diplomarbeit
Eclipse Entity Relationship Diagram Editor mit Codegenerierung

Rico Metzger <rico@ricosoft.ch>
Peter Kunz <pk@bostitch.ch>

IT 4b

Labor E425, 27. Oktober 2005

Abstract

One of our courses was about database theory which was taught with the book „Datenbanktheorie“ (Database Theory) written by H. Buff. This book introduces a special notation for Entity Relationship Diagrams. As there was no known Open Source editor available for designing and editing such diagrams, the idea of this diploma thesis was to develop such an editor.

This Entity Relationship Diagram Editor runs under Eclipse. It is implemented using the *Plug in Development Environment*, so it provides its own *plug in extension point* which allows other plug ins to implement their own code generation (such as SQL or Hibernate). The editor plug in is based on the *Graphical Editing Framework* of Eclipse. This framework provides the possibility to view and edit graphical diagrams.

The first release of this plug in delivers an editor that allows the user to design/edit diagrams, to undo recent actions and to load and save them. It also provides an interface for others to write their own code generation plug in. Unfortunately the implementation of a hibernate code generation plug in and the possibility of diagram validation are still missing.

Zusammenfassung

Im Rahmen des Studiums beschäftigten wir uns mit der Datenbanktheorie, wobei als Lehrmittel das gleichnamige Buch von H. Buff eingesetzt wurde. In diesem werden im Rahmen des Datenbankdesigns auch Entity-Relationship-Diagramme vorgestellt. Da keine Open Source-Lösung zum Zeichnen und Bearbeiten von solchen Diagrammen bekannt war, wurde im Rahmen dieser Diplomarbeit ein solcher Editor entwickelt.

Das Ziel der Diplomarbeit war die Entwicklung eines Entity-Relationship-Diagramm-Editors, welcher als Eclipse-Plugin realisiert werden sollte. Neben dem Editor wurde noch eine Schnittstelle zur Verfügung gestellt, um aus den erstellten Diagrammen direkt Code zu generieren (z.B. SQL oder auch Hibernate). Als Grundlage diente das Graphical Editing Framework von Eclipse, welches Möglichkeiten bietet, um Diagramme darzustellen und zu bearbeiten.

Der fertiggestellte Editor bietet die Funktionalität, die man von einem solchen in der heutigen Zeit erwartet. So werden zum Beispiel das Laden und Speichern von Diagrammen (im XML-Format) oder aber auch die Möglichkeit zum Rückgängig machen von Aktionen unterstützt. Daneben bietet er auch eine Schnittstelle, um Code direkt aus den Diagrammen erstellen zu lassen. Es fehlt allerdings ein Plugin, welches Hibernate-Code generiert. Zudem ist die Validierung von Diagrammen noch nicht implementiert, wobei dazu ebenfalls eine Schnittstelle zur Verfügung gestellt wurde.

Inhaltsverzeichnis

KAPITEL 1

Aufgabenstellung

Das Ziel unserer Diplomarbeit ist die Implementation der folgenden Funktionalität in Form eines Plugins für die Eclipse-Platform:

- Die Erstellung und Bearbeitung von ER-Diagrammen soll dem Benutzer durch einen visuellen, intuitiv bedienbaren Editor ermöglicht werden.
- Das Plugin soll die Korrektheit der ER-Diagramme überprüfen und den Benutzer auf Fehler aufmerksam machen. Als Vorlage für diese Überprüfung dient uns die Definition eines ER-Diagramms im Buch „Datenbanktheorie“ von H. Buff, welche eine Erweiterung der Chen-Notation darstellt.
- Die zu erwartende Grundfunktionalität des Ladens und Speicherns solcher Diagramme soll ebenfalls gewährleistet sein. Dazu definieren wir eine klare Dokumentenstruktur, welche durch eine entsprechende Definitionsdatei kontrolliert werden kann.
- Des weitern ist es möglich, direkt aus den Diagrammen entsprechenden Code zu generieren. Zu diesem Zweck wird gegen aussen eine Schnittstelle zur Verfügung gestellt. Als Referenz werden zwei Implementationen der Schnittstelle bereitgestellt, welche zum einen SQL- und zum anderen Hibernate-Code erzeugen.
- Eine Druckfunktion und der Export in Bilddateien der Diagramme werden ebenfalls bereitgestellt.

Dabei halten wir uns an die Eclipse-Hausregeln, wie sie im Buch von Erich Gamma und Kent Beck “Eclipse erweitern“[1] definiert werden.

[1] siehe [ECE]

KAPITEL 2

Pflichtenheft

Das Pflichtenheft soll einen Einblick geben, warum wir diese Diplomarbeit gewählt haben, welchen Zweck sie erfüllt, welche Technologien eingesetzt werden und wie wir dies umsetzen wollen.

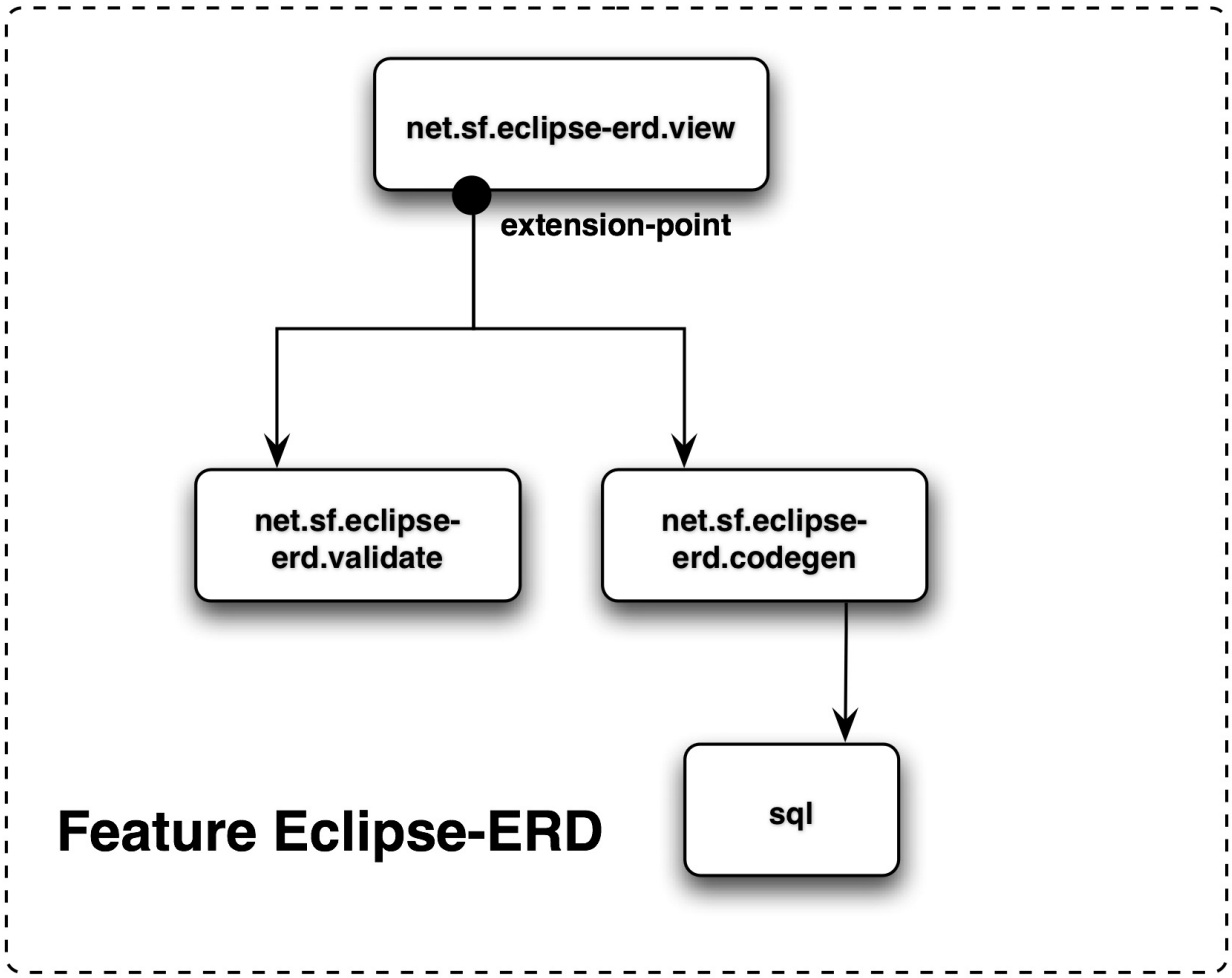
2.1 Warum?

Im Rahmen unseres Studiums beschäftigten wir uns mit der Datenbanktheorie, wobei wir als Lehrmittel das gleichnamige Buch von H. Buff[1] verwendeten. In diesem wird in einem eigenen Kapitel auf das Datenbankdesign mittels ER-Diagrammen eingegangen.
Einer der beiden Studenten beschäftigte sich in seiner zweiten Projektarbeit mit dem Thema „Hibernate“, dabei auch mit der Thema Codegenerierung. Dabei fiel ihm auf, dass es zur Zeit keine frei verfügbaren Editoren für die obengenannten ER-Diagramme gibt. Daraus entstand die Idee, im Rahmen der Diplomarbeit selbständig einen solchen zu realisieren.
Damit dieses Projekt auch innerhalb der vorgegebenen Zeitspanne verwirklicht werden kann, musste ein Framework gefunden werden, welches zum einen frei verfügbar ist und zum andern ein Gerüst für einen graphischen Editor zur Verfügung stellt.
Auf Grund dieser Rahmenbedingungen kristallisierten sich die beiden Entwicklungsumgebungen Netbeans und Eclipse heraus, welche sich für die gestellte Aufgabe eignen würden. Die Wahl fiel schlussendlich auf Eclipse, da einer der Autoren der Diplomarbeit sich mit diesem schon gut auskannte und es gleichzeitig auch noch über eine hervorragende Dokumentation verfügte. Hinzu kam, dass mit den beiden Teilprojekten Graphical Editing Framework (GEF)[2] und Eclipse Modeling Framework (EMF) [3] eine gute Grundlage für den visuellen Editor existierte.

[1] siehe [DBT]
[2] siehe [GEF]
[3] siehe [EMF]

2.2 Realisierung

2.2.1. Technischer Bereich



Das gesamte Projekt ist schlussendlich als Feature direkt von der Projekt–Homepage bei Sourceforge[4] installierbar (eine Anleitung finden Sie im Anhang unter „A.1 Installation“ auf Seite 27). Ein Feature ist dabei der Zusammenschluss von mehreren Plugins, wobei wir für unser Projekt folgende Plugins in der angegebenen Reihenfolge entwickeln:

1. Visueller Editor ohne Funktionalität ausserhalb des einfachen Zeichnens.

   Als Grundlage zur Programmierung dieses Plugins dient GEF aus den Eclipse-Tools.

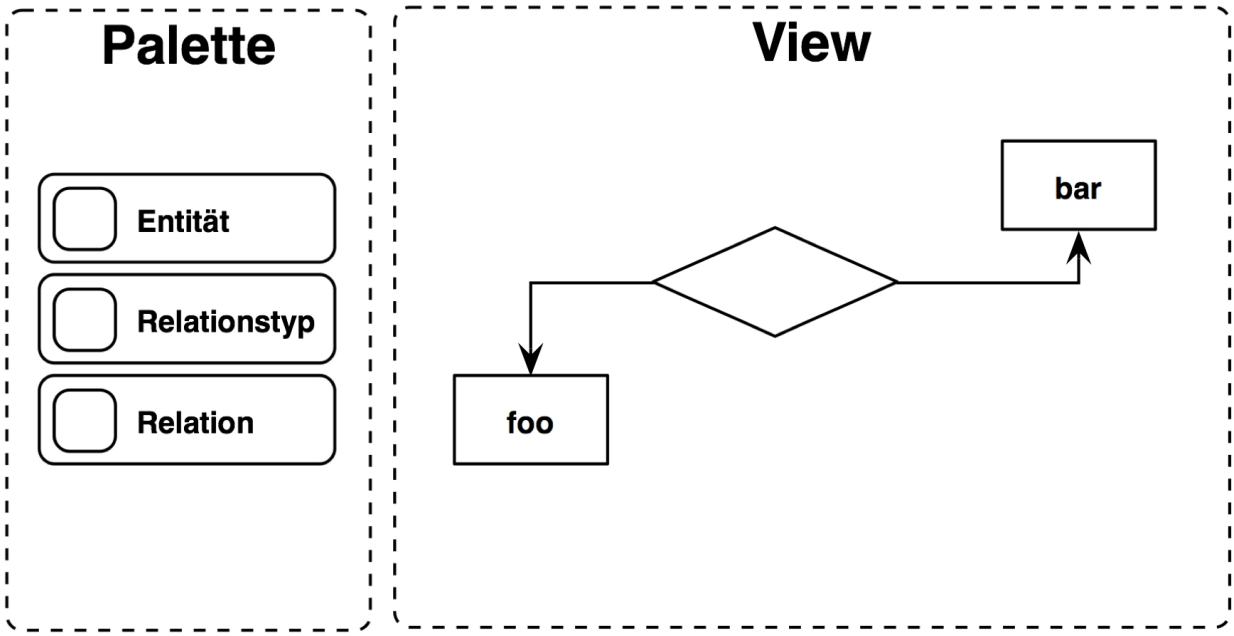
   Der visuelle Editor hat folgende Bestandteile:

   - Eine Palette, aus der man mittels Drag'n'Drop Elemente direkt ins Modell einfügen kann.
   - Einen Viewer-Part, in welchem das Modell dargestellt wird.
   - Popup-Menüs, welche auf Rechtsklick in den Viewer-Part eine angepasste Funktionalität zur Verfügung stellen, je nachdem welches Objekt zur Zeit ausgewählt wurde.
   - Eine Menü-Funktionalität, welche neben dem Erstellen neuer Objekte über Menüs auch die heute gewohnte Funktionalität einer automatischen Anordnung der Objekte, wie auch die normalen Bearbeitungs-Befehle (Undo, Cut, Copy, Paste) zur Verfügung stellt.

[4] siehe [SFN]

Dabei soll der visuelle Editor immer nur die gerade notwendige Funktionalität zur Verfügung stellen, damit der Benutzer nicht durch unnötige Funktionalität verwirrt wird.

Wichtig ist auch, dass der Benutzer des Systems möglichst nicht eingeschränkt wird, da wir davon ausgehen können, dass die Benutzer keine Anfänger sondern wohl eher Entwickler oder Datenbank-Administratoren sein werden.



2. Kontrolle der Korrektheit des Diagramms → Fehler einfärben.

   Dieses Plugin wird bei jeder Änderung des Diagramms aufgerufen, aber auch sofort nach dem Laden eines Dokumentes.

3. Automatisches Anordnen des Diagramms.

   Dieses Plugin soll vor allem dafür sorgen, dass bei Situationen, in welchen die Elemente nicht richtig auf dem Workspace verteilt sind, automatisch eine zweckvolle Anordnung erstellt wird.

4. Speichern als XML-Datei gemäss Dokumentendefinition.

   Wichtig ist hier, dass man nicht nur den Inhalt der Entitätstypen und die Zusammenhänge speichern kann, sondern auch dafür gesorgt wird, dass die Position der Elemente nicht verlorengeht.

5. Laden mit anschliessender Kontrolle der Korrektheit.

   Hier sollte zuerst überprüfen, ob die Original-Positionen wieder hergestellt werden können. Ansonsten soll eine automatische Anordnung am Ende des Ladens ausgeführt werden.

6. Exportieren als JPG / PNG.

   Es können nur Grafikformate unterstützt werden, die nicht durch Patente geschützt werden.

7. Codegenerierung SQL

8. Codegenerierung Hibernate

9. Drucken

   Wird nur implementiert, sofern die Zeit dafür noch reicht.

2.2.2. Projektmanagement

Bei der Programmierung stützen wir uns auf folgende Grundsätze aus dem Extreme Programming[5]:

Pair Programming Hierbei geht es um die Idee, dass zwei Programmierer gemeinsam an einem Monitor ihre Arbeit verrichten. Dadurch sollen Fehler reduziert und ein gegenseitiger Ansporn zu sauberer, strukturierter Programmierung gegeben werden.

Write Tests First Bevor auch nur eine Zeile funkionaler Code seinen Weg in das Projekt findet, wird zuerst für jede Routine ein Test geschrieben, welcher prüfen soll, dass die Funktion das erledigt, wofür sie vorgesehen ist.

Iteration Programming Die Idee hierbei ist, dass man eine Iteration, welche aus mehreren Aufgaben zusammengestellt wird, zuerst fertig implementiert, bevor man zur nächsten Iteration übergeht. Auf diese Weise ist es auch möglich, eine kontinuierliche Planung zu erstellen.

Auf diese Weise ist es uns möglich, kontinuierlich den Fortschritt unseres Projektes zu beobachten und zu sehen, ob Engpässe entstehen. Die Kombination aus Pair Programming und Write Tests First sorgt dafür, dass der Code auch eine gute Qualität aufweist.

[5] siehe [XPP]

KAPITEL 3

Software-Beschreibung

3.1 Eclipse

Am Ende des Jahres 2001 wurde die Eclipse Plattform in der Version 2.0 auf dem Web (von Beginn weg unter `http://www.eclipse.org`) zur Verfügung gestellt. Dies war der Startpunkt für diese äusserst erfolgreiche IDE, welche sich vor allem in der Welt der Java Entwickler einen Namen gemacht hat.
Die Gründe für den Erfolg der Plattform sind vielseitig. Hier nur einige:

Open Source Jeder kann mit entwickeln. Codeänderungen werden vom *Project Management Committee* und den *Project Leaders* begutachtet, bevor sie akzeptiert und allen zur Verfügung gestellt werden. Auf diese Weise wird die hohe Qualität der Software sichergestellt.

Benutzerorientiert Das oben genannte Committee geht auf die Wünsche der Benutzer ein und entwickelt nicht nach eigenen Wunschvorstellungen. Gerade deshalb erfreut sich die Plattform weiterhin einer sehr grossen Beliebtheit.

Modularer Aufbau mit Plugins Neben der direkten Entwicklung für die Plattform kann jedermann seine eigenen Plugins schreiben. Diese können dann gepackt und für alle zur Verfügung gestellt werden. Weitere Infos siehe weiter unten unter dem Abschnitt „3.1.2 Plugins“.

Viele Plugins Es existieren riesige Bibliotheken mit hunderten von Plugins für jedermann. Es gibt beinahe nichts, was man auf dem Netz nicht finden würde. So standen zum Zeitpunkt, als dieses Dokument geschrieben wurde, unter `http://eclipse-plugins.2y.net/` beinahe 1000 Plugins zum Download bereit.

3.1.1. Platform

Zu Beginn war Eclipse „nur“ eine gute IDE, welche aus Visual Age for Java von IBM entstand. Dieses wurde ab 1999 nicht mehr weiterentwickelt, aber unter Open Source freigegeben. Dadurch erfreute sich Eclipse von Beginn weg einer grossen Anzahl von Benutzern, da viele VisualAge Anwender mit Eclipse weiterarbeiteten.
Mit der Zeit konnte man mit Eclipse nicht nur Java-Code entwickeln, sondern auch C++, PHP und andere Sprachen fanden ihren Weg in die IDE. Die Beliebtheit der Plattform stieg kontinuierlich an. Alleine im Jahr 2004 erreichte die Eclipse Plattform einen Zählerstand von 30 Millionen Downloads (allerdings gibt es hier auch Mehrfachdownloads).

Die Plattform selber ist eigentlich nur ein kleines Set aus Plugins, welche die Grundfunktionalität von Eclipse zur Verfügung stellt. Mit der Plattform alleine erhält man nur sehr einfache Funktionalität im

Bereich des User Interfaces. Jedoch wird diese minimale Funktionalität verwendet, um zum Beispiel Applikationen zu entwickeln, die wie Eclipse aussehen (sogenannte Rich Client Application, siehe auch `http://eclipse.org/rcp/`). An die Plattform werden weitere Plugins angefügt, sehr häufig natürlich die *Java Development Tools*, die das Entwickeln von Java-Anwendungen erst ermöglichen.

3.1.2. Plugins

Abgsehen vom Kern setzt sich Eclipse ausschliesslich aus Plugins zusammen. Diese werden mit Hilfe des *Plug-in Development Environments (PDE)* entwickelt. Dabei ist das System sehr einfach gehalten. Um selber ein Plugin zu entwickeln, geht man wie folgt vor:

1. Man sucht einen geeigneten *Extension Point*.
2. Man startet ein „Plug-in Project“ (Eclipse stellt dazu einige *Wizards* zur Verfügung, die auch komplette Plugins erstellen).
3. Nun definiert man in der Datei `plugin.xml` die gewünschte Erweiterung (*Extension*), zum Beispiel wie folgt:

```
<plugin>
  ...
  <extension
    point="org.eclipse.ui.newWizards">
    <category
      id="net.sf.eclipse_erd.view"
      name="ER Diagrams"/>
    <wizard
      category="net.sf.eclipse_erd.view"
      class="net.sf.eclipse_erd.view.wizards.ERNewDiagram"
      id="net.sf.eclipse_erd.view.wizards.ERNewDiagram"
      name="Create ER Diagram">
      <selection class="org.eclipse.core.resources.IResource"/>
    </wizard>
  </extension>
  ...
</plugin>
```

   Dieses einfache Beispiel erstellt einen neuen Wizard, welcher in Eclipse unter `File / New` erscheint. Dies erreichen wir, indem wir den *Extension Point* `org.eclipse.ui.newWizards` erweitern.

4. Als nächstes implementiert man die notwendigen Klassen. Meist muss man vordefinierte Interfaces oder Klassen erweitern, dazu muss man sich mit der Dokumentation des gegebenen *Extension Points* auseinandersetzen. In unserem Beispiel müssten wir die Klasse `ERNewDiagram` entwickeln, welche das Interface `org.eclipse.ui.INewWizard` implementieren muss.

Natürlich kann man auch selber bei Plugins solche *Extension Points* zur Verfügung stellen. Dazu geht man wie folgt vor:

1. Man definiert den *Extension Point* in der Datei `plugin.xml` wie folgt:

```
<extension-point id="validate" name="Validation"
        schema="schema/validate.exsd"/>
```

2. Man schreibt ein *Schema*, welches den *Extension Point* definiert. Dabei sollte man auch genau definieren, welche Parameter wofür stehen und welche Interfaces oder Klassen implementiert werden müssen. Diese Informationen sieht derjenige, welcher den *Extension Point* implementieren will, nämlich auch. Eclipse stellt hierfür eigens einen eigenen Wizard zur Verfügung, mit welchem man die Einträge direkt editieren kann (Abb. 3.1).

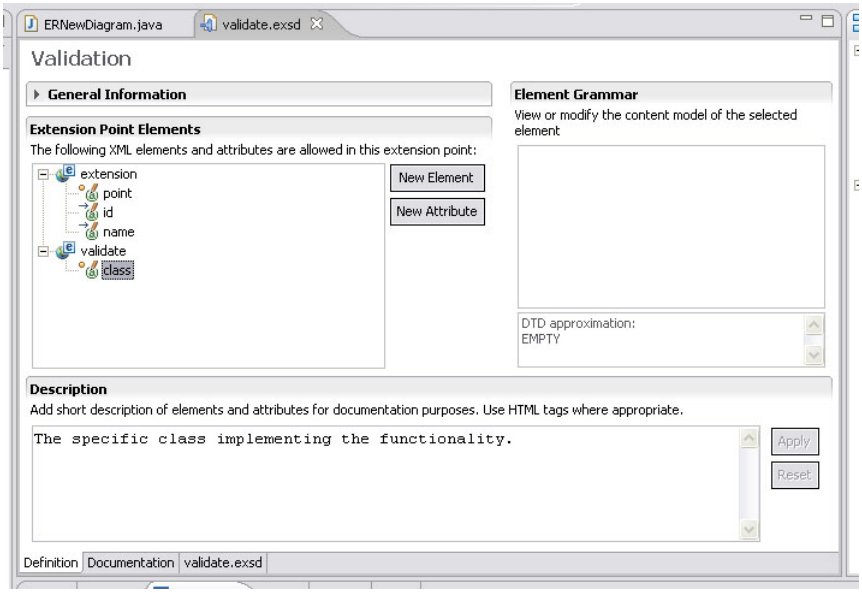


Abbildung 3.1.: Extension Point Schema Editor

3. Man definiert ein Interface oder eine Klasse, welche erweitert werden müssen und stellt diese zur Verfügung.

3.1.3. Draw2D

Draw2D stellt die Mittel zur Verfügung, um Diagramme in Eclipse darzustellen. Dabei basiert es auf folgender Funktionalität:

1. **Darstellung von Figuren**. Selbstentwickelte Figuren müssen das Interface `IFigure` erweitern. Es kann aber auch aus einer Selektion von vordefinierten Figuren ausgewählt werden.

2. **LayoutManager**. Diese definieren, wie die Figuren angeordnet werden — von einem einfachen XYLayout, bei dem der Entwickler selber Grösse und Position der Figuren festlegt, bis zu komplexen Tabellen-Layouts findet man hier alles.

3. **Darstellung von Verbindungen**. Dabei werden nur die Start- und Endfigur einer Verbindung festgelegt. Draw2D zeichnet dann eine direkte Verbindung bzw. man kann definieren, wie sich Draw2D einen Weg bahnen soll, indem man sogenannte *Router* programmiert. Ein einfaches Beispiel ist der `ShortestPathConnectionRouter`, welcher zwei Figuren auf dem kürzesten Weg verbindet, anderen Figuren dabei aber ausweicht.

   Wichtig ist auch, dass die Figuren sogenannte *Anchors* definieren. Diese legen fest, wo die Verbindung an den Rahmen einer Figur anstösst und somit enden soll. Dies ist wichtig, da es zum einen möglich ist, die Endpunkte mit Dekorationen wie Pfeilen oder ähnlichem zu versehen, zum andern aber auch, weil die Verbindungen immer über den Figuren angezeigt werden und somit bei falschen *Anchors* die Verbindungen in die Figur hineingezeichnet werden, was nicht erwünscht ist.

3.1.4. GEF

Während Draw2D dazu dient, Diagramme darzustellen, dient GEF dazu, diese Diagramme auch editierbar zu machen. Dabei basiert es auf dem *Model-View-Controller-Pattern* (Abb. 3.2). Die *View* wird von *Draw2D-Figuren* und *Connections* dargestellt, das *Model* stellt der Programmierer zur Verfügung und GEF übernimmt den Part des *Controllers*.

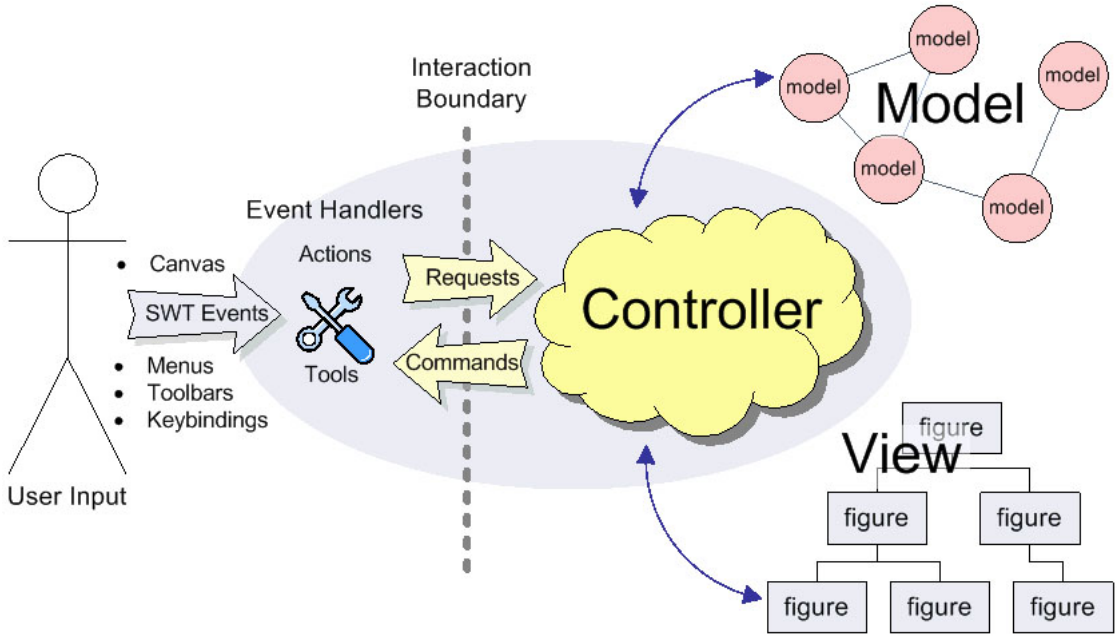


Abbildung 3.2.: MVC-Pattern nach GEF

Um mit GEF zu entwickeln, geht man am Besten wie folgt vor:

1. Zuerst entwickelt man das *Modell*, welches dargestellt werden soll. Wichtig ist, dass das Modell die Funktionalität des *Observable* im Observer-Pattern bereitstellt. Es muss also möglich sein, dass sich Figuren der View registrieren, um über Änderungen informiert zu werden.

2. Danach entwickelt man Figures und Connections, welche die Darstellung des Modells übernehmen.

3. Man implementiert für jede Verbindung zwischen Modell und View einen *EditPart*. Dieser registriert sich beim Modell als *Observer*, um über Änderungen informiert zu werden und kümmert sich darum, dass Modell und View von Beginn weg synchronisiert sind und auch bleiben. Wichtig ist, dass jeder *EditPart* sich darum kümmern muss, dass auch alle ihm untergeordneten Modelle richtig aktualisiert werden. Im Klartext: Wenn ich einen EditPart für unser Diagramm erstelle, muss sich dieser EditPart darum kümmern, dass alle Typen, welche für das Diagramm registriert sind, ebenfalls ihre EditParts kriegen. Dazu verwendet man ein *Factory Pattern*, welches automatisch dem richtigen Modell den richtigen EditPart zuweist.

4. Um jetzt das Modell auch noch *bearbeitbar* zu machen, muss man sogenannte *EditPolicies* zur Verfügung stellen. Hier ist auch das grösste Problem für uns entstanden, denn diese ändern sich von Version zu Version.

   Die Idee ist dabei wie folgt: Wenn das GEF eine Änderung feststellt, löst es einen *Request* aus, der an die richtige *Policy* weitergeleitet wird. Diese wird dann an das aktuell gewählte Element weitergeleitet, von diesem bearbeitet und danach kontinuierlich an den jeweils nächsten Elternteil weitergeleitet, bis es keinen solchen mehr gibt.

Es gibt eine Vielzahl von *EditPolicies*. Eine aktuelle Auflistung findet man jeweils in der Hilfe von Eclipse.

Man registriert nun eine eigene Klasse, welche sich um eine *EditPolicy* kümmert. Zum Glück gibt es für dies meisten Policies vordefinierte Klassen und Interfaces, welche man erweitern bzw. implementieren kann, so dass man nicht alle *Requests* selbständig den richtigen Methoden zuweisen muss. Man kann natürlich auch selber *Requests* definieren.

Um *Requests* zu beantworten, muss man selber ein `Command` definieren. Es ist nicht gestattet, selber direkt etwas am Modell oder der View zu ändern. Statt dessen generiert man ein `Command` und fügt dieses in den `CommandStack` des Editors ein. Dieser führt dann den Befehl aus, was folgende Vorteile bringt:

- Es ist für GEF sehr einfach, ein *Undo/Redo* zu ermöglichen
- Änderungen sorgen dafür, dass GEF sich merkt, dass der Inhalt „Dirty“ ist und gespeichert werden muss. Natürlich wird auch festgestellt, wenn man bis zum letzten Speicherpunkt zurückgeht, sodass das „Dirty“-Flag zurückgesetzt wird.
- Es ist ohne grossen Aufwand möglich, den `CommandStack` als „Version“ abzuspeichern und so eine Versionsverwaltung zu ermöglichen.

Beispiel:

Um das soeben erklärte ein bisschen näher zu bringen, hier ein einfaches Beispiel. Wir möchten darauf reagieren, wenn der Benutzer aus unserem ER Diagramm einen Typ oder ein Attribut löscht. Wir gehen davon aus, dass wir folgendes schon implementiert haben:

Wir haben die beiden Commands `ModelDeleteCommand` und `AttributeDeleteCommand`, welche das „Löschen“ ausführen und dabei auch noch gleich die notwendigen Verbindungen ebenfalls entfernen. Das Ganze so, dass wir problemlos ein Undo/Redo machen können.

Als erstes definieren wir eine Klasse `EREditPolicy`, welche `ComponentEditPolicy` erweitert. Diese Policy hat unter anderem schon definiert, dass sie auf den `Delete`-Request reagiert. Um auf den Befehl zu reagieren, müssen wir eine Methode dieser Policy-Klasse überschreiben, was wir wie folgt tun:

```
protected Command createDeleteCommand(GroupRequest deleteRequest) {
  Object parent = getHost().getParent().getModel();
  Object child = getHost().getModel();
  if (parent instanceof ERDiagram && child instanceof Attribute) {
    return new AttributeDeleteCommand((ERDiagram) parent,
                (Attribute) child);
  }
  if (parent instanceof ERDiagram && child instanceof ERType) {
    return new ModelDeleteCommand((ERDiagram) parent,
                (ERType) child);
  }
  return super.createDeleteCommand(deleteRequest);
}
```

Nun müssen wir natürlich unsere *Policy* noch registrieren, dies tun wir in unserem `EditPart` wie folgt:

```
protected void createEditPolicies() {
  // allow removal of the associated model element
  installEditPolicy(EditPolicy.COMPONENT_ROLE,
                new ERComponentEditPolicy());
}
```

Die Methode `createEditPolicies()` ist dabei schon vordefiniert und muss nur überschrieben werden.

KAPITEL 4

Projektorganisation und Endzustand

4.1 Projektorganisation

4.1.1. Iterative Programmierung

Es war unser Glück, dass wir uns für iterative Programmierung entschieden haben. Das Erlernen des GEFs brauchte ein Vielfaches der dafür eingeplanten Zeit. Es entstand die Problematik, dass wir mit einer aktuellen Version des Frameworks arbeiteten, aber jegliche verfügbare Dokumentation auf dem Netz noch für die Vorgängerversion geschrieben war. Speziell störend dabei war auch die Tatsache, dass nicht einmal die Beispiele, die auf `eclipse.org` zur Verfügung gestellt wurden, mit der aktuellen Version funktionierten.
Dieser Zustand änderte sich erst Mitte Oktober 2005 mit der Veröffentlichung der aktuellen Dokumentation. Bis zu diesem Zeitpunkt hatten wir uns allerdings die notwendigen Informationen schon auf andere Art und Weise (vor allem dank der API-Dokumentation und mittels Try-and-Error-Methodik) angeeignet.
Dank der iterativen Programmierung war es uns möglich, zunächst einmal einen sehr simplen Editor fertigzustellen. Den Rest der Funktionalität wollten wir dann mittels externer Plugins an den Editor heranführen. Für die Codegenerierung und die Validierung wurden deshalb Schnittstellen zur Verfügung gestellt.

4.1.2. Write tests first

Leider konnten wir unseren Vorsatz, dass wir zuerst die Tests schreiben wollten, nicht umsetzen. Das Testen unter dem Eclipse Plug-in Development Environment ist nicht sehr einfach und uns fehlte die Zeit, dies auch noch zu erlernen.
Natürlich wurden manuelle Tests durchgeführt, dies ist jedoch nicht zu vergleichen mit den automatisierten Tests, die wir gerne eingeführt hätten.

4.1.3. Pair Programming

Speziell am Anfang, als es um die Programmierung des Modells ging, entwickelten wir alles im Paar. Auf diese Weise war die wichtigste Grundlage von Beginn weg beiden klar. Es kam hinzu, dass sich zeigte, dass Fehler viel schneller entdeckt und zugleich korrigiert wurden. Dadurch erwies sich diese Art der Programmierung als sehr effizient.
Gegen Ende der Diplomarbeit wurde zum Teil getrennt programmiert, wobei speziell die Codegenerierung und der Editor getrennt entwickelt wurden.

4.2 Endzustand

- ☑ Visueller Editor
- ☑ Kontrolle der Korrektheit des Diagramms (muss mittels Plugins angefügt werden, aber die Funktionalität ist grundsätzlich vorhanden)
- ☒ Automatisches Anordnen des Diagramms (nur die Verbindungen werden automatisch angeordnet)
- ☑ Speichern als XML-Datei (die Formatierung der XML-Datei lässt noch zu wünschen übrig, aber es ist eine reguläre XML-Datei)
- ☑ Laden mit anschliessender Kontrolle der Korrektheit
- ☒ Exportieren als JPG / PNG
- ☑ Codegenerierung SQL — allerdings ist die Generierung noch nicht perfekt
- ☒ Codegenerierung Hibernate
- ☒ Drucken

4.3 Weiterführende Ideen

Dieses Projekt weist mit Sicherheit noch Punkte auf, an denen weitergearbeitet werden kann. Folgende Punkte bieten sich an:

- Fehlende Funktionalität des Editor:
  - Export als JPG / PNG
  - Drucken
  - Hilfedatei: Eclipse bietet die Möglichkeit, HTML-Hilfe für Eclipse zu entwickeln. Eine solche Hilfedatei könnte man sicherlich auch noch für dieses Plugin entwickeln und einbinden.

  Das Hauptproblem beim Editor ist, dass man einen grossen Teil seiner Zeit zur Einarbeitung in GEF selbst sowie zur Einarbeitung in den bestehenden Editor benötigt.
- Codegenerierung
  - SQL fertigstellen: Im Moment hat der SQL-Generator noch ein bekanntes Problem: Er generiert keine automatischen Primary Keys bei Entitätstypen. Man könnte ihn auch so erweitern, dass er den DDL-Code für verschiedene Datenbanken erstellt.
  - Hibernate-Code generieren
  - DAO-Code generieren

  Der Vorteil hier liegt darin, dass man dazu einfach ein entsprechendes Plugin implementieren muss. Da bereits ein Beispiel für eine solche Implementation vorhanden ist, braucht es auch eine viel geringere Einarbeitungszeit.
- Validierung: Das Plugin zur Validierung der Diagrammdaten macht im Moment nur marginale Prüfungen. Dieses könnte man ohne grössere Probleme um weitere Tests erweitern.

KAPITEL 5

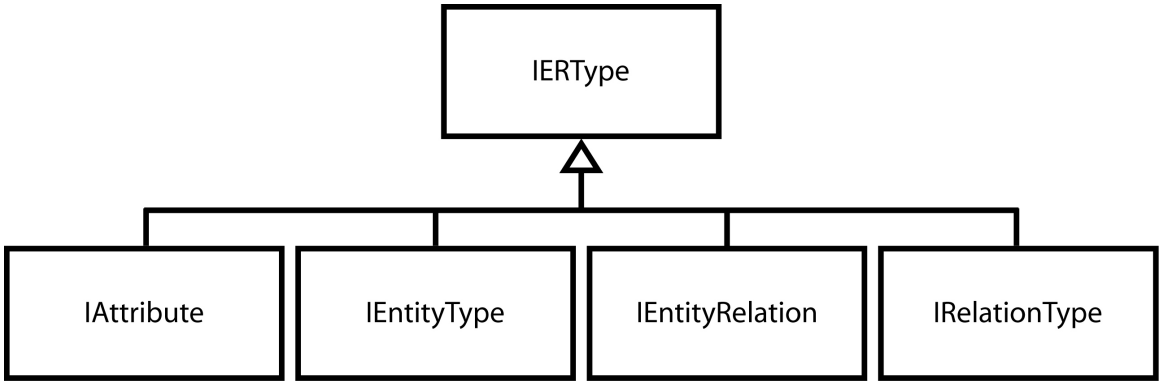
Problemanalyse und Lösungskonzept

5.1 Datenmodell

Die folgenden Interfaces definieren unser Datenmodell. Man kann auf sie zugreifen, wenn man das Plugin `net.sf.eclipse_erd.view` einbindet. Sie sind unter `net.sf.eclipse_erd.model.interfaces` zu finden. Schreibt man ein eigenes Codegenerierungs-Plugin, so wird einem das Modell in Form eines `IERDiagramms` übermittelt.

IERDiagram Dieses Interface stellt die Grundfunktionalität für ein einzelnes Diagramm zur Verfügung. Es speichert eine Liste aller Typen und Attribute, die vorhanden sind.

Typen Hier werden die Datentypen des ER Diagramms gespeichert.



IERType Dieses Interface ist eine *Oberklasse*. Die Interfaces für Entitätstypen, Beziehungstypen und Zusammengesetzte Beziehungstypen erweitern dieses Interface. Es stellt folgende Methoden zur Verfügung:

- `String getName()` gibt den Namen des Typs zurück.
- `List<IAttribute> getAttributes()` gibt eine Liste der Attribute für den Typ zurück.
- `List<IGeneralRelation> getSourceConnections()` Alle Verbindungen, die von diesem Typ weggehen.
- `List<IGeneralRelation> getTargetConnections()` Alle Verbindungen, die bei diesem Typ enden.
- `boolean isValid()` Gibt `true` zurück, wenn der Typ alle Validierungskriterien erfüllt.

- `setValid(boolean valid)` Setzt den Typ auf *gültig* oder *ungültig* (wird vor allem von den Validierungs-Plugins verwendet).

IEntityType Dieser Typ erweitert `IERType` und stellt einen *Entitätstypen* dar. Er stellt keine weiteren Methoden zur Verfügung.

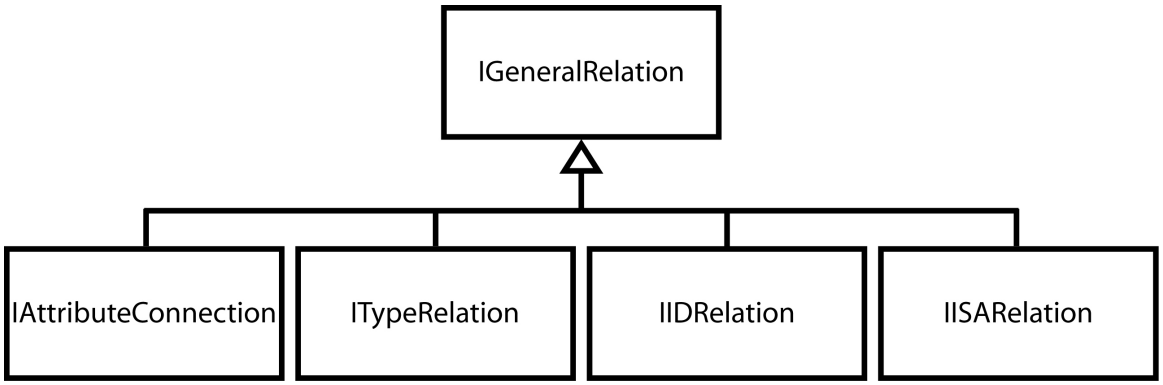
IRelationType Dieser Typ erweitert `IERType` und stellt einen *Beziehungstypen* dar. Er stellt keine weiteren Methoden zur Verfügung.

IEntityRelation Dieser Typ erweitert `IERType` und stellt einen *Zusammengesetzten Beziehungstypen* dar. Er stellt keine weiteren Methoden zur Verfügung.

IAttribute Dieser Typ erweiter `IERType` und stellt ein *Attribut* für einen Typ dar. Er hat folgende zusätzliche Funktionalität:

- `IERType getParent()` gibt den Typ zurück, an welchen dieses Attribut angehängt ist.
- `int getLength()` gibt die Länge des Attribut-Typs zurück, sofern dies benötigt wird.
- `String getType()` gibt den Attribut-Typ zurück, welcher für die Codegenerierung verwendet werden soll.
- `boolean isPrimaryKey()` gibt `true` zurück, wenn dieses Attribut zum Schlüssel des zugehörigen Typs gehört.

Verbindungen Hier werden Verknüpfungen zwischen verschiedenen Typen gespeichert.



IGeneralRelation Dies ist die *Oberklasse* für alle Interfaces, welche Verbindungen repräsentieren. Es definiert folgende Funktionalität:

- `IERType getSource()` Der Typ woher die Verbindung kommt.
- `IERType getTarget()` Der Typ wohin die Verbindung geht.

IIDRelation Definiert eine *ID-Beziehung* zwischen zwei Entitätstypen. Keine zusätzliche Funktionalität zu `IGeneralRelation`.

IISARelation Definiert eine *ISA-Verbindung* zwischen zwei Entitätstypen. Keine zusätzliche Funktionalität zu `IGeneralRelation`.

ITypeRelation Definiert eine Verbindung zwischen einem Beziehungstyp und einem Entitätstypen.

- `int getMultiplicity()` Gibt die Multiplizität für die Verbindung zurück, wobei 0 für die Multiplizität m steht.

IAttributeConnection Definiert eine Verbindung zwischen einem Attribut und dem Besitzer des Attributs. Keine zusätzliche Funktionalität zu `IGeneralRelation`

5.2 XML-Beschreibung

Ein Diagramm wird als XML-Datei mit der Endung `.erd` gespeichert und ist wie folgt aufgebaut:

5.2.1. <diagram>

Die Tags des `<diagram>`-Elements umschliessen die beiden Elemente `<elements>` und `<connections>`, welche die einzelnen Elemente des Diagramms beinhalten. Der Diagrammname wird im Attribut `name` des `<diagram>`-Elements gespeichert.

```
<?xml version="1.0" encoding="UTF-8"?>
<diagram name="Auktionshaus">
  <elements>
    (<entity> | <relation> | <entityrelation>)*
  </elements>
  <connections>
    (<connection> | <id> | <isa>)*
  </connections>
</diagram>
```

5.2.2. <elements>

Im Element `<elements>` werden die *Entitättypen* (`<entity>`), *Beziehungstypen* (`<relation>`) sowie *Zusammengesetzten Beziehungsypen* (`<entityrelation>`) des Diagramms gespeichert. Sie alle beinhalten das Element `<position>` sowie optional (mehrmals) das Element `<attribute>`.

<entity>

In der XML-Beschreibung repräsentiert eine `<entity>` einen *Entitätstypen*. Der Namen des entsprechenden Entitätstypen wird im Attribut `name` gespeichert. Die Position des Entitättyps wird in den Attributen `x` und `y` des Elements `<position>` gespeichert. Für jedes dem Entitättypen angehängten Attribut wird ein Element `attribute` angehängt. (*Das Element* `<attribute>` *wird weiter unten im Detail beschrieben.*)


```
<entity name="Rechnung">
  <position x="108" y="90"/>
  (<attribute>)*
</entity>|
```

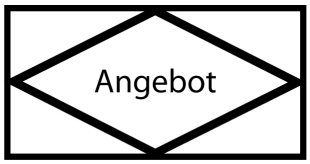
<relation>

Der *Beziehungstyp* wird in der XML-Beschreibung als `<relation>` bezeichnet, wobei hier (im Element `<elements>`) nur der Rombus beschrieben wird. Die Pfeile des Beziehungstyps werden getrennt im Element `<connections>` beschrieben. Wie schon beim Entitätstypen werden Position und optional Attribute beschrieben.


```
<relation name="Rechnungsstellung">
  <position x="217" y="80"/>
  (<attribute>)*
</relation>|
```

<entityrelation>

Der *Zusammengesetzte Beziehungstyp* wird in der XML-Beschreibung durch das Element `<entityrelation>` beschrieben. Wie schon beim Beziehungsyp werden auch hier die Pfeile losgelöst im Element `<connections>` gespeichert und Position und eventuell angehängte Attribute beschrieben.




```
<entityrelation name="Angebot">
  <position x="468" y="301"/>
  (<attribute>)*
</entityrelation>
```

<attribute>

Das Element `<attribute>` verfügt über die vier Attribute `name`, `primarykey`, `type` und `length`. Der Attributname wird in `name` gespeichert, während `primarykey` angibt, ob es sich beim Attribut um einen Bestandteil des Primary Keys handelt oder nicht. Der Typ des Attributs wird in `type` und `length` gespeichert. `type` beinhaltet den Typ (z.B. DATE, TIME, INTEGER, VARCHAR, ...) und `length` eine eventuell notwendige Länge zu `type`.

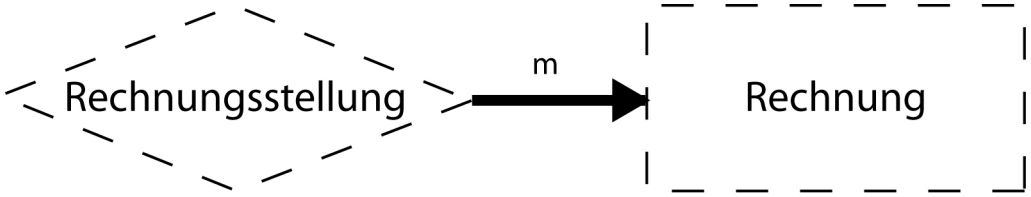

```
<attribute length="0" name="Mindestpreis"
        primarykey="false" type="DOUBLE">
  <position x="380" y="369"/>
</attribute>
```

5.2.3. <connections>

Im Element `<connections>` werden alle „Pfeile“ gespeichert, die von *Beziehungstypen* resp. *Zusammengesetzten Beziehungstypen* zu *Entitätstypen* und *Zusammengesetzten Beziehungstypen* führen sowie *ISA-* bzw *ID-abhängige Entitättypen* die auf *Entitätstypen* zeigen. Es wird dazu zwischen den folgenden drei Verbindungen unterschieden:

<connection>

Das Element `<connection>` verfügt über die Attribute `from`, `to` und `multiplicity`. Der Wert in `from` kennzeichnet den Startpunkt des *Pfeils* (Name des Beziehungstyps / Zusammengesetzten Beziehungstyps), der `to`-Wert das Ziel auf das der *Pfeil* zeigt (Name des Entitättyps / Zusammengesetzter Beziehungstyps). Das Attribut `multiplicity` speichert die Markierung des *Pfeils*, wobei 0 für m steht.




```
<connection from="Rechnungsstellung" multiplicity="0" to="Rechnung"/>
```

<id>

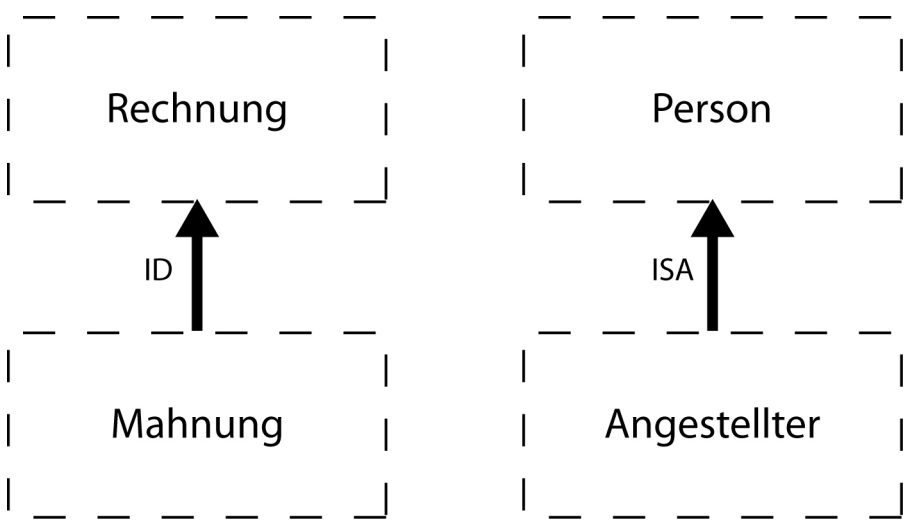
Das Element `<id>` verfügt wie schon das Element `<connection>` über die Attribute `from` und `to`, welche die Namen der Entitätstypen beinhaltet die der *ID-Pfeil* verbindet.

```
<id from="Mahnung" to="Rechnung"/>
```

<isa>

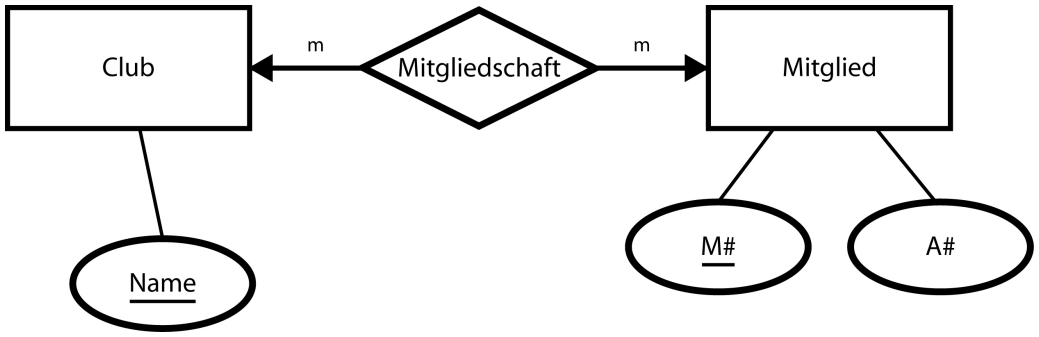
Das `<isa>`-Element ist analog zum `id`-Element aufgebaut und speichert wie dieses die Namen von Start- und Zielpunkt des *ISA-Pfeils* in den Attributen `from` und `to`.

```
<isa from="Angestellter" to="Person"/>
```



5.2.4. Beispiel

Das nachfolgende Beispiel zeigt den Aufbau einer `erd`-Datei.




```
<?xml version="1.0" encoding="UTF-8"?>
<diagram name="Beispiel-Diagramm">
  <elements>
    <entity name="Club">
      <position x="11" y="10"/>
      <attribute length="20" name="Name" primarykey="true"
                             type="VARCHAR">
        <position x="15" y="24"/>
      </attribute>
    </entity>
    <entity name="Mitglied">
      <position x="63" y="9"/>
      <attribute length="0" name="M#" primarykey="true"
                             type="INTEGER">
        <position x="57" y="22"/>
      </attribute>
      <attribute length="0" name="A#" primarykey="false"
                             type="INTEGER">
        <position x="74" y="22"/>
      </attribute>
    </entity>
    <relation name="Mitgliedschaft">
      <position x="38" y="9"/>
    </relation>
  </elements>
  <connections>
    <connection from="Mitgliedschaft" multiplicity="0" to="Mitglied"/>
    <connection from="Mitgliedschaft" multiplicity="0" to="Club"/>
  </connections>
</diagram>
```

KAPITEL 6

Schlusswort — Erfahrungen und Erkenntnisse

6.1 Erfahrungen allgemein

6.1.1. Positive Erfahrungen

Für uns war es sicherlich eine gute Sache, einmal ein Projekt auf der Eclipse Plattform entwickeln zu können. Diese Plattform bietet einiges, was man als vorbildlich bezeichnen kann:

- Ausführliche, übersichtliche Dokumentation
- Ein komplett modularer Aufbau
- Gut dokumentierter, einfacher Code, den man gut lesen kann

Da man bei der Plugin-Entwicklung sehr häufig mit dem Code von Eclipse selbst in Kontakt kommt, lernt man so von den Erfahrungen dessen Entwickler, was selbst eine nützliche Erfahrung ist.
Des weitern kann man auch sagen, dass wir mit der iterativen Programmierung und dem Pair Programming gute Erfahrungen gemacht haben. Beides ermöglichte uns einen kontinuierlichen Fortschritt, wobei wir nie zu viel planten, sondern möglichst bald kleine Codestücke schrieben. Das Pair Programming sorgte für eine markante Reduktion der Fehler.

6.1.2. Probleme

Der Entscheid mit der neuen Version des GEF zu arbeiten, während die gesamte Dokumentation noch auf der alten Version basiert, erschwerte das Vorankommen ungemein. So mussten wir uns einen Grossteil des Wissens mühsam zusammensuchen. Es war natürlich gleichzeitig auch eine Chance, jedoch wollten wir eigentlich die dadurch verloren gegangene Zeit in die Generierung von Hibernate-Code investieren.
Auch die Tatsache, dass automatisierte Tests fehlen, muss als negativer Punkt aufgefasst werden. Es wäre notwendig, solche zu entwickeln, damit andere Entwickler weiterentwickeln und sich sicher sein könnten, dass der alte Code noch immer funktioniert. Ohne automatisierte Tests ist es sehr schwierig, vorauszusagen, ob dies der Fall ist. Dies bedeutet auch, dass sich ein Entwickler, welcher am Editor weiterarbeiten will, viel länger in den Code einarbeiten muss, um sich sicher zu sein, dass er keine fehlerhafte Funktionalität entwickelt.

6.2 Schlusswort

Die Diplomarbeit war sicherlich eine lehrreiche Erfahrung und die Tatsache, dass wir am Schluss einen Editor mit Codegenerierung hatten, welcher auch funktionierte, war mit Sicherheit eine Genugtuung. Auf der anderen Seite lässt sich sagen, dass wir den Aufwand eher unterschätzt hatten und am Schluss einen Teil der geplanten Funktionalität doch nicht implementieren konnten.

6.3 Fazit von Peter Kunz

Da ich - wie ich im Verlauf der Diplomarbeit feststellen durfte - doch nicht so gut mit Eclipse vetraut war, wie ich es zuvor angenommen hatte, waren diese acht Wochen ein wahrer Segen für mich. So gab es einige Aha-Erlebnisse, als ich den bis anhin unbekannten Funktionsumfang von Eclipse zu entdeckten begann.
Da wir mehr Zeit, als wir eigentlich eingeplant hatten, für das Einarbeiten und Erlernen des Graphical Editing Frameworks verwendeten, fehlte uns diese am Schluss der Arbeit, wodurch Punkte wie z.B. die Hybernate-Codegenerierung oder der Export von Bildern nicht mehr realisiert werden konnte. Die Hauptgründe dafür waren, dass wir zum einen auf einem Beispielcode aufgebaut hatten, welcher sich leider als nicht ganz GEF-konform herausstellte und daher wieder verworfen wurde und zum anderen, dass die Dokumentation des aktuellen GEF-Releases dem Release hinterher hinkte (Sie wurde in Woche sechs der DA released).
Was mich aber positiv überrascht hatte, waren die Erfahrungen mit dem Pair Programming. Die Tatsache, dass zwei Augenpaare Fehlern eher auf die Schliche kommen und dass der Druck, Code sauber zu kommentieren doch ungemein grösser war, sprechen sehr wohl für dieses System.
Rückblickend bin ich mit dem Ergebnis unserer Arbeit zufrieden. Es wurden aus Mangel an Zeit leider nicht alle gesteckten Ziele erreicht, doch gelang es uns, die Hauptpunkte in unserem Plugin zu realisieren.

6.4 Fazit von Rico Metzger

Meine erste Feststellung war, dass sich Eclipse in den letzten zwei Jahren stark verändert hatte. Als ich zum letzten Mal ein Plugin für Eclipse entwickelte, war es noch Version 2.1 und seit dieser Version hat sich viel verbessert.
Die aktuelle Unterstützung für die Entwicklung eigener Plugins ist phänomenal. Dort wurden wesentliche Fortschritte gemacht. Auch die aktuelle Dokumentation hat sich um einiges verbessert.
Mit der Dokumentation kommt allerdings auch ein grosser Wermutstropfen: ausgerechnet die Dokumentation zum Graphical Editing Framework war in keinster Weise aktuell. Irgendwie ging gerade dieser für uns wichtige Teil bei der Aktualisierung der Dokumentation vorerst vergessen. So verbrachten wir sehr viel Zeit damit, herauszufinden, welche Teile der Beispiele sich kompilieren liessen (auch die GEF-eigenen Beispiele liessen sich nicht mehr kompilieren) und dann beim nichtfunktionalen Teil die API-Dokumentation zu konsultieren, wie man dies denn im aktuellen Release lösen muss. Gleichzeitig war es auch eine Chance, da wir auf diese Weise viel Source Code von GEF lesen und daraus lernen konnten.
Auch das Pair Programming hat mich positiv überrascht. Diese Art der Programmierung scheint effizienter zu sein, als ich erwartet hatte. Während dieser Zeit entstanden viel weniger Fehler bzw. sie wurden sofort korrigiert.
Sicherlich habe ich den Aufwand der Diplomarbeit unterschätzt. Speziell die Einarbeitung in GEF verbrauchte ein Vielfaches der geplanten Zeit. Dennoch bin ich mit dem Erreichten zufrieden.

KAPITEL A

Benutzerhandbuch

A.1 Installation

A.1.1. Java

Zur Verwendung dieses Plugins wird Java 2 Standard Edition, *mindestens* Version 1.5, vorausgesetzt. Dieses kann von der Seite `http://www.java.com` heruntergeladen und installiert werden. Auf dieser Seite finden Sie auch eine ausführliche Installationsanleitung.

A.1.2. Eclipse

Nachdem Java installiert ist, benötigen Sie Eclipse. Eine aktuelle Version von Eclipse findet sich unter `http://www.eclipse.org/downloads/`. Nach dem Download müssen Sie das Archiv an einem Ort ihrer Wahl extrahieren und danach die ausführbare Datei im Eclipse-Ordner starten (unter Windows: `eclipse.exe`). Eclipse sollte jetzt starten und Sie mit dem Willkommen-Bildschirm begrüssen.

A.1.3. GEF

Weiter brauchen wir das Graphical Editing Framework von Eclipse. Dies kann man auf zwei Arten installieren: entweder über den sehr nützlichen *Update Manager* von Eclipse (empfohlene Vorgehensweise) oder indem man *GEF direkt* von der Seite herunterlädt und installiert. Diese beiden Optionen werden hier vorgestellt.

Variante 1: Update Manager

Starten Sie Eclipse, wenn Sie es noch nicht getan haben. Gehen Sie dann auf das Menü „Help“, dort auf den Menüpunkt „Software Updates“ und dann auf „Find and Install“ (Abb. A.1).

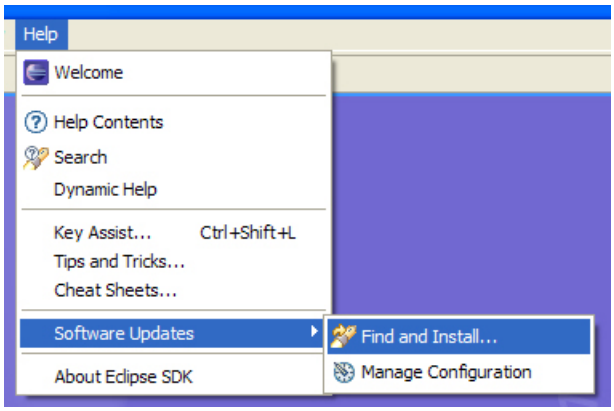


Abbildung A.1.: Update Manager - Schritt 1

Wählen Sie nun den zweiten Punkt „Search for new features to install“ und klicken auf „Next“ (Abb. A.2).

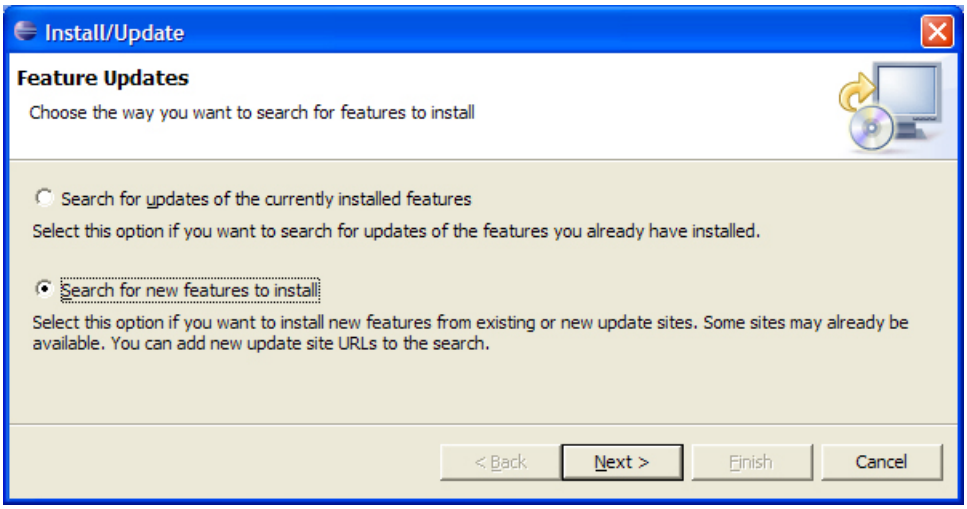


Abbildung A.2.: Update Manager - Neues Feature

Sie erhalten jetzt eine Liste mit Seiten. Falls Sie Eclipse frisch installiert haben, erscheint nur die Eclipse-Seite selbst. Wählen Sie diese aus und klicken Sie dann auf „`Finish`“ (Abb. A.3).

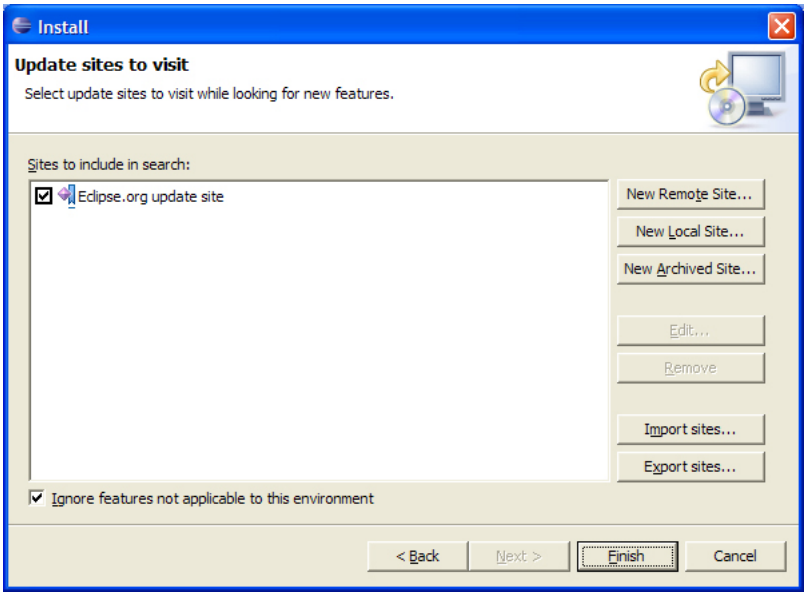


Abbildung A.3.: Update Manager - Seiten Auswahl

Sollte eine Frage nach der Seite erscheinen, von welcher Sie die Updates installieren wollen, wählen Sie eine Seite in Ihrer Nähe (Abb. A.4).

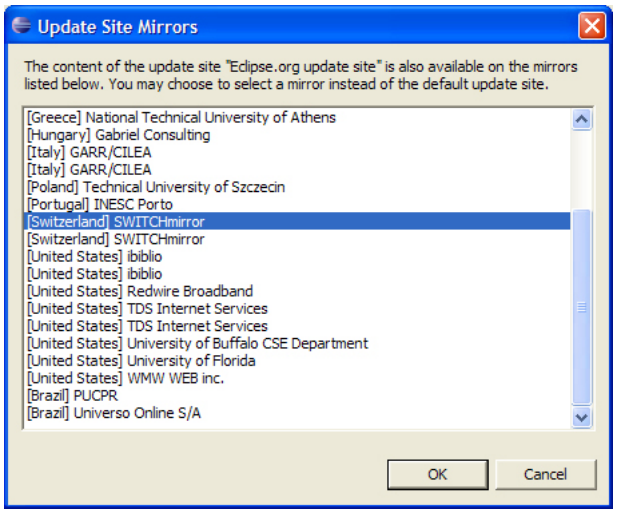


Abbildung A.4.: Update Manager - Quellen Wahl

Sie erhalten eine Auswahl von möglichen Installationspunkten. Stellen Sie sicher, dass Sie das Graphical Editing Framework (GEF) ausgewählt haben und klicken Sie auf „`Finish`“ (Abb. A.5).

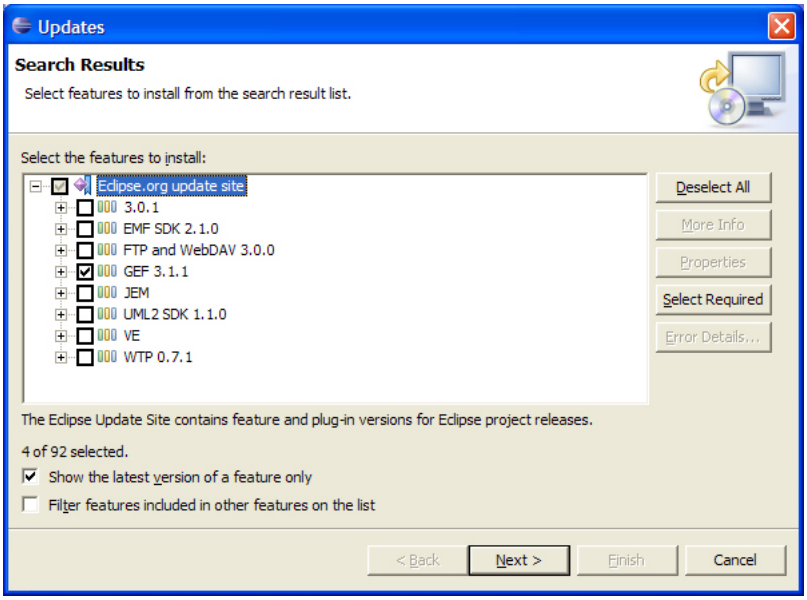


Abbildung A.5.: Update Manager - Auswahlbildschirm

Eclipse verlangt noch eine Bestätigung der Lizenzbedingungen (Abb. A.6 - Abb. A.8) und installiert danach GEF. Nach der Installation müssen Sie Eclipse neustarten (Abb. A.9). GEF ist jetzt installiert.

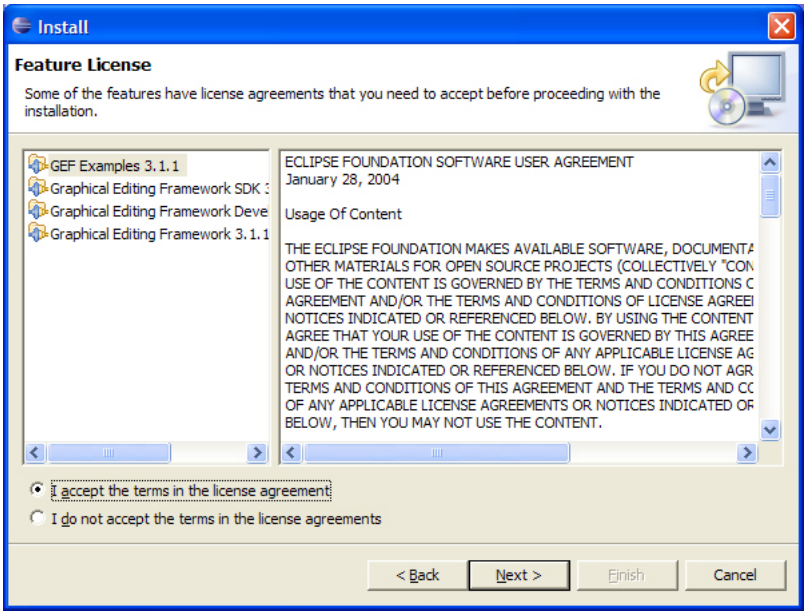


Abbildung A.6.: Update Manager - Installation

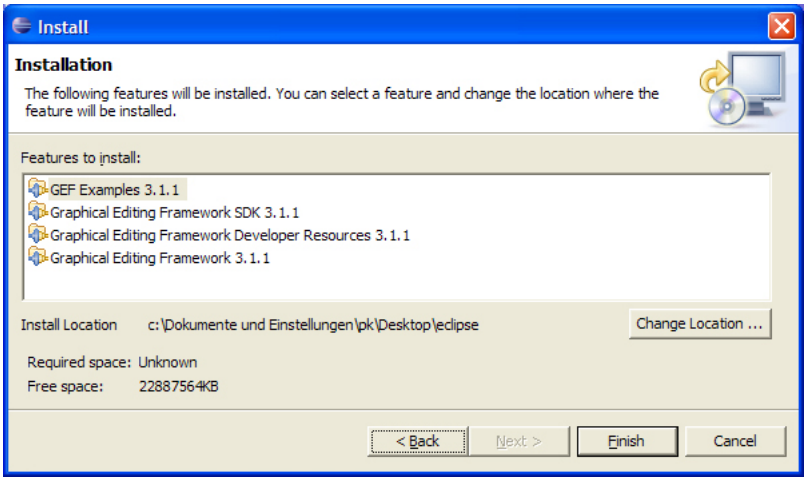


Abbildung A.7.: Update Manager - Installation

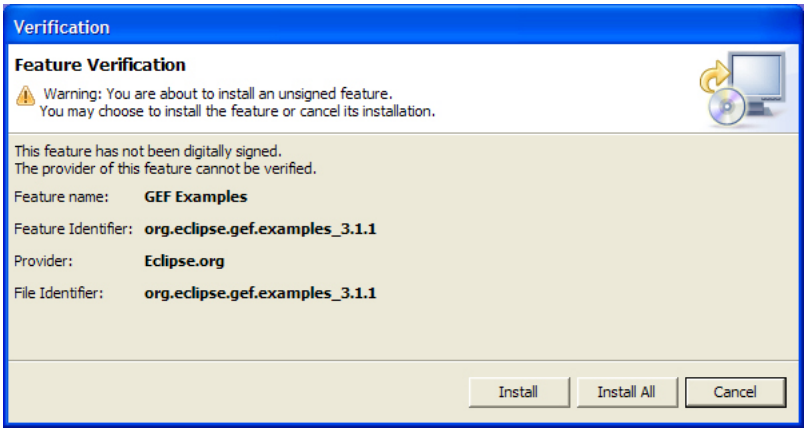


Abbildung A.8.: Update Manager - Installation

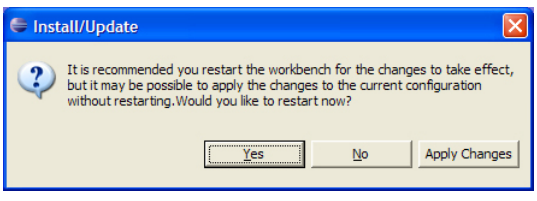


Abbildung A.9.: Update Manager - Neustart

Variante 2: Download

Beenden Sie Eclipse, wenn es noch läuft. Gehen Sie dann auf die Download-Seite von GEF (diese finden Sie über `http://www.eclipse.org/gef/`) und laden dort „`GEF-ALL`“ herunter (Abb. A.10 - Abb. A.11) und entpacken Sie das erhaltene Archiv in den Eclipse-Ordner.
Beim nächsten Start von Eclipse sollte GEF installiert sein.

Abbildung A.10.: GEF Download - Schritt 1

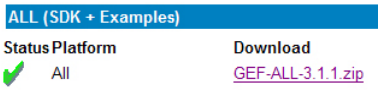


Abbildung A.11.: GEF Download - Schritt 2

A.1.4. Plugin

Das Letzte was Sie noch installieren müssen, ist das Plugin selber. Gehen Sie dazu auf die SourceForge-Seite des Plugins unter `http://www.sf.net/projects/eclipse-erd/`. Dort klicken Sie auf *Files* und laden sich das letzte Release herunter. Dieses entpacken Sie ebenfalls in den Eclipse-Ordner. Beim nächsten Neustart von Eclipse wird dieses Plugin ebenfalls installiert sein, was Sie wie folgt überprüfen können.

Klicken Sie auf das Menü „Help“ und dort auf den Menüpunkt „About Eclipse SDK“ (Abb. A.12).

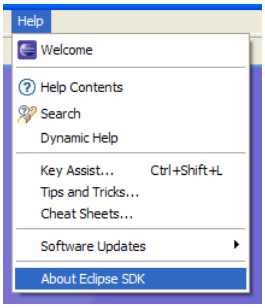


Abbildung A.12.: Plugin-Kontrolle - Help

Im nächsten Fenster können Sie sich mit einem Klick auf „Feature Detail“ und „Plugin Details“ (Abb. A.13) die installierten Features und Plugins anzeigen lassen. Sind diese wie in Abb. A.14 ersichtlich aufgeführt, so ist das Plugin korrekt installiert.

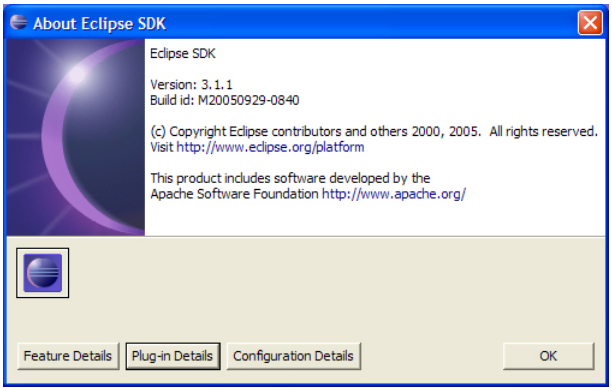


Abbildung A.13.: Plugin-Kontrolle - About

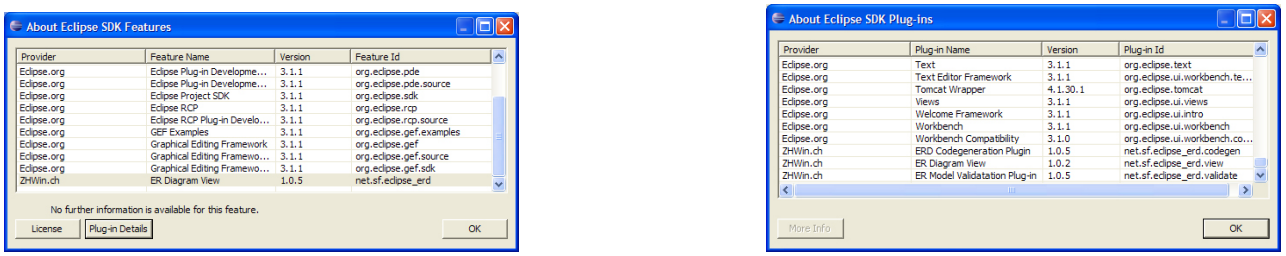


Abbildung A.14.: Plugin-Kontrolle - Feature und Plugins

A.2 Anleitung

Wir haben versucht, das Plugin soweit als möglich selbsterklärend zu gestalten. Ergänzend stellen wir hier diese kleine Bedienungsanleitung zur Verfügung.

A.2.1. Neue Datei erstellen

Hierzu müssen Sie in Eclipse bereits ein Projekt geöffnet haben. In „A.2.2 Neues Projekt erstellen“ weiter unten wird dieser Vorgang im Detail beschrieben.

Klicken Sie nun im Menü `File` unter „`New`“ auf den Menüpunkt „`Other...`“ (Abb. A.15).

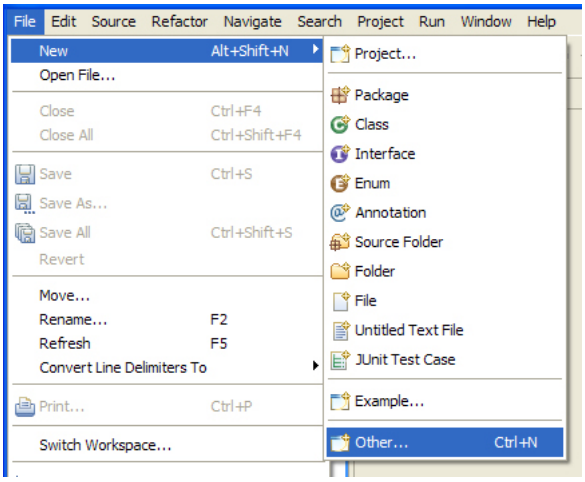


Abbildung A.15.: Neue ERD-Datei - Schritt 1

Wählen Sie nun im Ordner „ER Diagrams“ „Create ER Diagram“ aus und klicken auf „Next“ (Abb. A.16).

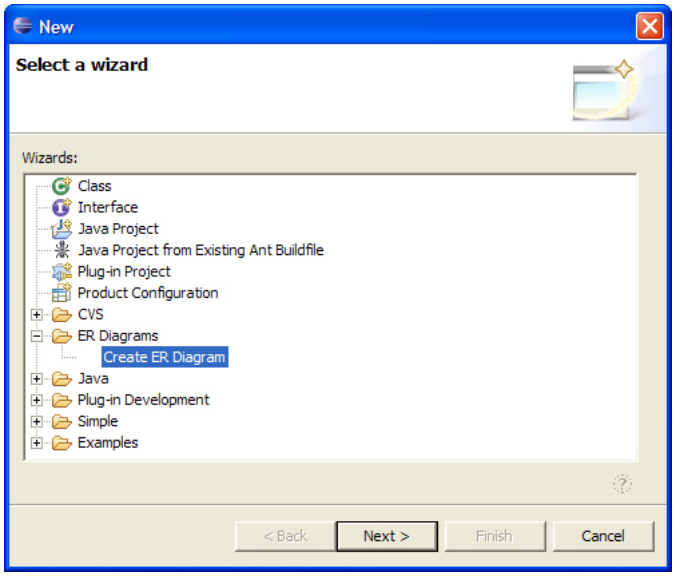


Abbildung A.16.: Neue ERD-Datei - Schritt 2

Wählen Sie nun noch Ihr Projekt als Container aus und ändern den Dateinamen bevor Sie auf „Finish“ klicken und die Datei erstellt wird (Abb. A.17).

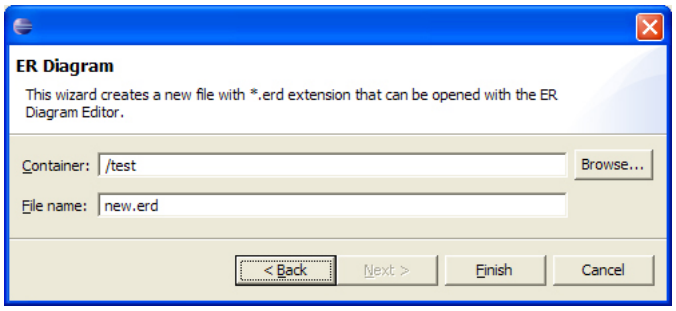


Abbildung A.17.: Neue ERD-Datei - Schritt 3

A.2.2. Neues Projekt erstellen

Sie erstellen ein neues Projekt indem Sie im Menü „File“ unter „New“ den Menüpunkt „Project“ auswählen (Abb. A.18).

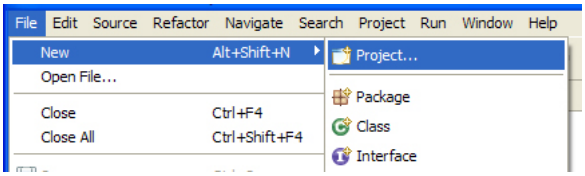


Abbildung A.18.: Projekt erstellen - Schritt 1

Wählen Sie nun „`Project`“ aus und klicken auf „`Next`“ (Abb. A.19).

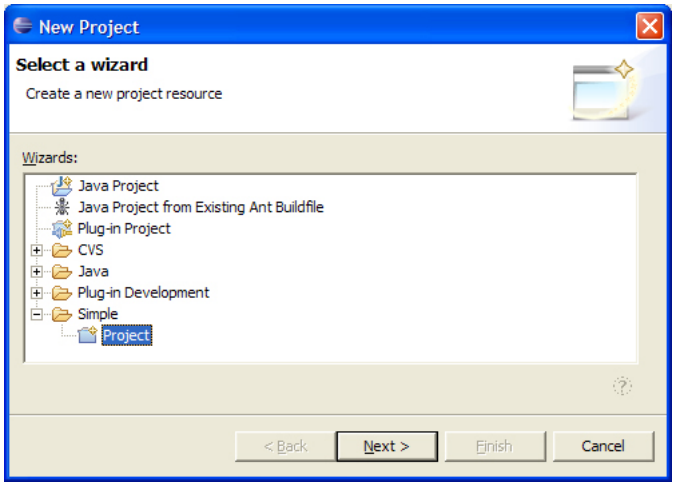


Abbildung A.19.: Projekt erstellen - Schritt 2

Im nächsten Schritt geben Sie dem Projekt noch einen Namen und klicken auf „`Finish`“ (Abb. A.20).

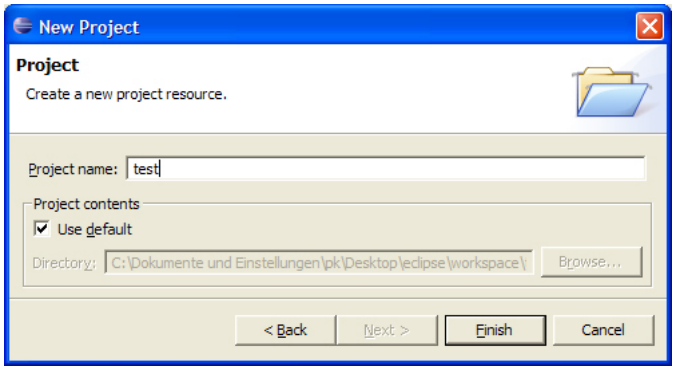


Abbildung A.20.: Projekt erstellen - Schritt 3

A.2.3. Diagramm bearbeiten

Haben Sie ein Diagramm, welches Sie bearbeiten möchten (Datei mit der Endung `.erd`), so öffnen Sie diese mittels einem simplen Doppelklicks. Der *ER Diagram Editor* sollte sich jetzt präsentieren (Abb. A.21).

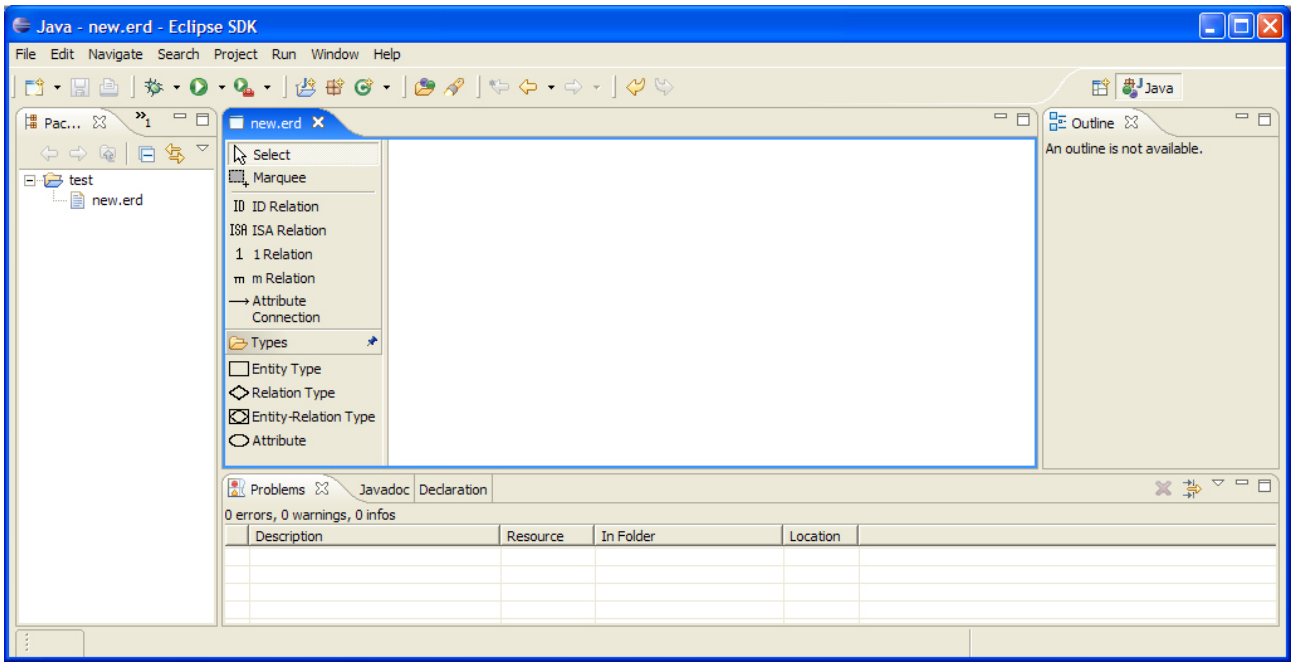


Abbildung A.21.: ER Diagram Editor

Properties-View

Um den Editor richtig nutzen zu können benötigen Sie noch die *Properties-View*. Falls Sie diese noch nicht aktiviert haben, so klicken Sie im Menü „`Window`“ im Submenü „`Show View`“ auf den Menüpunkt „`Other...`“ (Abb. A.22).

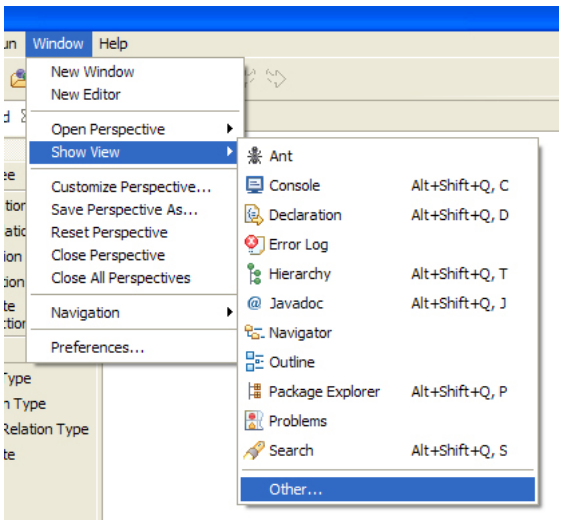


Abbildung A.22.: Properties-View aktivieren - Schritt 1

Wählen Sie nun im Ordner „Basic“ die View „Properties“ aus und klicken auf „OK“ (Abb. A.23).

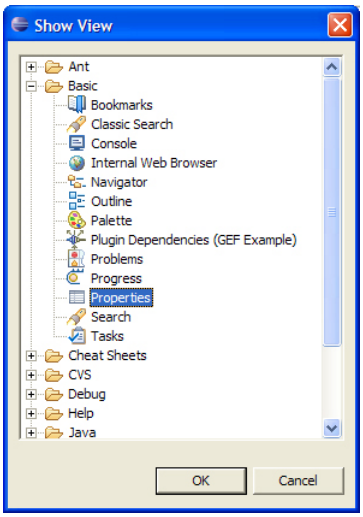


Abbildung A.23.: Properties-View aktivieren - Schritt 2

In der neu aktivierten *Properties-View* können Sie nun gleich den Namen des Diagramms ändern.

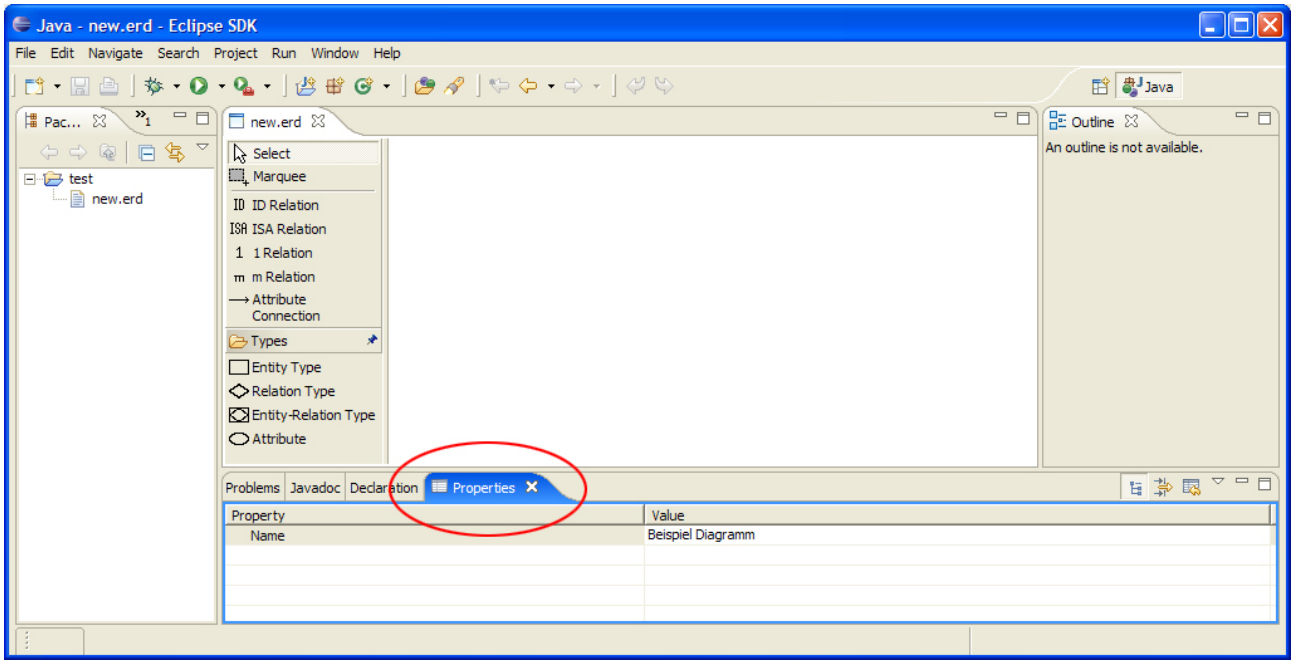


Abbildung A.24.: ER Diagram Editor mit Properties-View

Ein einfaches Diagramm

Das Erstellen eines Diagramms ist ganz einfach. Um beispielsweise einen *Entitätstypen* „Mitarbeiter“ zu erstellen, klicken Sie links in der Palette auf das Symbol „Entity Type“ (1), danach in der Arbeitsfläche an die Stelle, an der der Entitätstyp erstellt werden soll (2). Der Editor erstellt diesen und weist ihm automatisch einen Namen zu, der nun noch geändert werden muss. Dies geschieht in der *Properties-View* (3). Nach Ändern des Namens einfach die Änderung per *Eingabetaste* oder Klick auf die Arbeitsfläche

bestätigen und der Name wird übertragen (4).

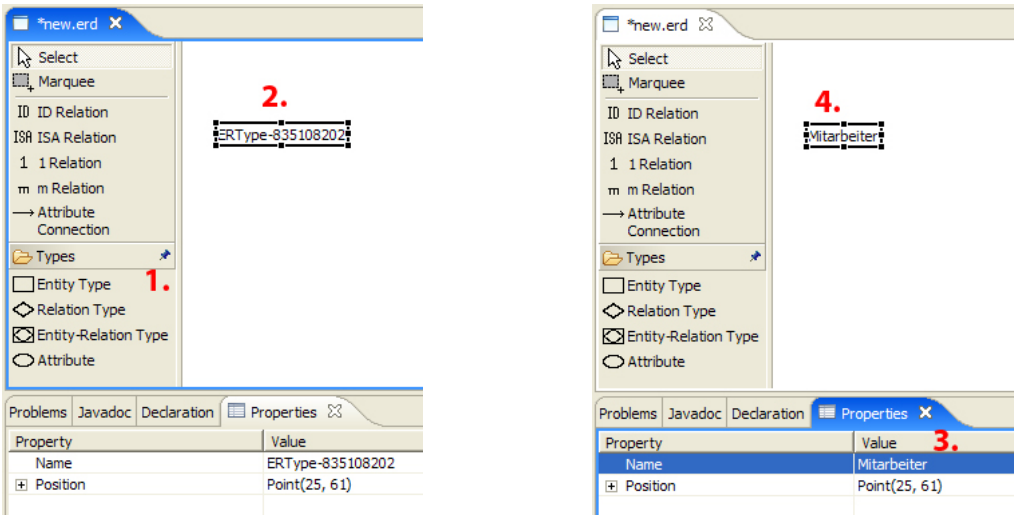


Abbildung A.25.: ER Diagram Editor

A.2.4. Code generieren

Um Code zu generieren, müssen Sie zuerst ein Diagramm haben, welches die Validierungskriterien erfüllt (dh. es darf kein Element mehr rot dargestellt werden). Ist dies der Fall, klicken Sie mit der *rechten Maustaste* auf eine leere Fläche des Diagramms und wählen dann im Menü „`Code Generation`“ die gewünschte Codegenerierung aus (A.26).
Falls Ihr Diagramm nicht gültig sein sollte, werden Sie den Menüpunkt „`Code Generation`“ nicht zur Auswahl haben.

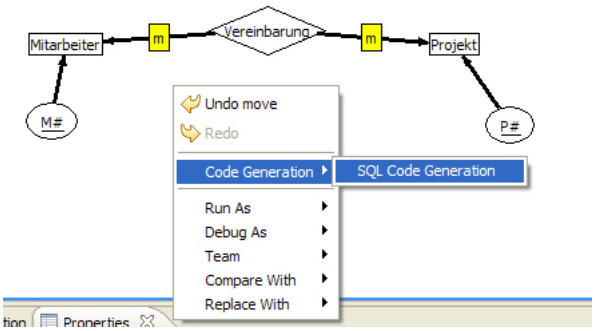


Abbildung A.26.: Codegenerierung - Auswahl

Sie werden nun aufgefordert anzugeben, wohin der generierte Code gespeichert werden soll. Wählen Sie einen Ordner im Projekt aus und geben Sie der Datei einen Namen (Abb. A.27).

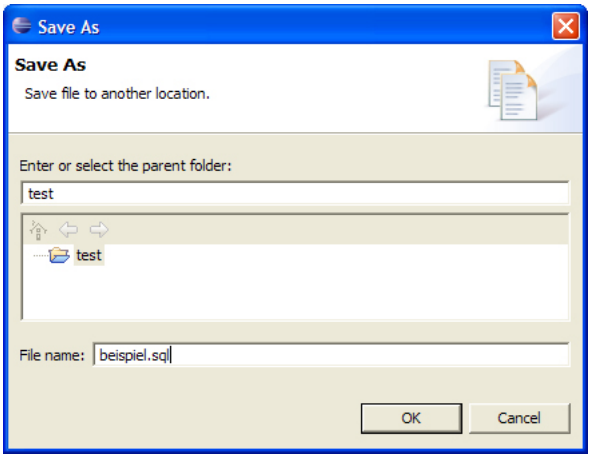


Abbildung A.27.: Codegenerierung - Save As

Aus dem kleinen Diagram in A.26 wurde nun der folgende SQL-Code generiert.

```
CREATE TABLE Mitarbeiter
(
        M# INTEGER  NOT NULL,
        PRIMARY KEY ( M# )
);

CREATE TABLE Projekt
(
        P# INTEGER  NOT NULL,
        PRIMARY KEY ( P# )
);

CREATE TABLE Vereinbarung
(
        P# INTEGER,
        M# INTEGER,
        PK1779980878 INTEGER NOT NULL,
        FOREIGN KEY ( P# ) REFERENCES Projekt,
        FOREIGN KEY ( M# ) REFERENCES Mitarbeiter,
        PRIMARY KEY ( PK1779980878 )
);
```

KAPITEL B

Kontaktmöglichkeiten

Sie erreichen die beiden Autoren dieser Diplomarbeit wie folgt:

Metzger Rico
Langgasse 63
8400 Winterthur

rico@ricosoft.ch
078/698 70 85

Kunz Peter
Seestrasse 203
8708 Männedorf

pk@bostitch.ch
077/413 36 77

Literaturverzeichnis

[DBT] H. Buff
Datenbanktheorie
Juni 2003 — BoD GmbH

[XPP] chromatic
Extreme Programming Pocket Guide
Juli 2003 — O'Reilly

[ECE] Kent Beck, Erich Gamma
Eclipse erweitern
Januar 2005 — Addison Wesley

[DPA] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
Design Patterns: Elements of Reusable Object-Oriented Software
Januar 1995 — Addison Wesley

[EMF] *Eclipse Modeling Framework*
`http://www.eclipse.org/emf/`

[GEF] *Graphical Editing Framework*
`http://www.eclipse.org/gef/`

[SFN] *Projekt–Homepage bei Sourceforge*
`http://eclipse-erd.sf.net/`

[IRB] *IBM Redbooks – GEF and EMF*
`http://publib-b.boulder.ibm.com/Redbooks.nsf/RedbookAbstracts/sg246302.html`

Abbildungsverzeichnis