FAKULTÄT FÜR !NFORMATIK

Datengesteuerte Erstellung von Eclipse-basierten Client-Applikationen

zum einfachen Erlernen des Eclipse-Frameworks

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Magister der Sozial- und Wirtschaftswissenschaften
im Rahmen des Studiums

Informatikmanagement

eingereicht von

Bernhard Hablesreiter
Matrikelnummer 0251249

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuer: Ao.Univ.Prof. Dipl.-Ing. Dr. techn. Gerald Futschek

Wien, 01.09.2008 _____ _____
(Unterschrift Verfasser/in) (Unterschrift Betreuer/in)

Kurzbeschreibung

Die vorliegende Arbeit beschäftigt sich mit der Erstellung von Client-Applikationen unter Verwendung von Java und der Eclipse Plattform. Dabei werden auf die Voraussetzungen zur Erstellung solcher Applikationen eingegangen und mögliche Problembereiche beschrieben.

Im Rahmen der Arbeit wurde ein Framework zur Erstellung von Client-Applikationen auf Basis von Eclipse entwickelt, das es ermöglicht, beliebige geeignete Datenquellen an Client-Applikationen anzubinden und die ausgetauschten Daten zu verwalten. Der Hauptteil der Arbeit beschäftigt sich mit der Architektur, Funktionsweise und Implementierung dieses Frameworks, wobei diverse Beispiele gegeben werden.

Desweiteren wird gezeigt, dass das Framework dazu geeignet ist, die Eclipse Plattform besser kennenzulernen und deren Funktionalitäten einfacher zu verstehen.

Abstract

This paper deals with the creation of client-applications based on the Eclipse Platform and Java. It describes the prerequisites for creating such applications and points out various problems during this process.

To counteract on these problems, a framework was developed during this thesis. It allows the connection between a client-application and various data sources and the management of the transferred data. The main part of the thesis describes the architecture, functions and implementations of the framework and gives examples for better understanding.

In addition, the thesis will point out the frameworks possible usage as a guideline for understanding the Eclipse Platform and its functionality.

Inhaltsverzeichnis

1 Einleitung

Das Thema dieser Arbeit ist die datengesteuerte Erstellung von Client-Applikationen auf Basis der Eclipse Plattform und Java. Die Arbeit deckt dabei die Beschreibung der zur Erstellung solcher Applikationen erforderlichen Hilfsmittel und Werkzeuge ab und gibt gleichzeitig einen Einblick in das komplexe Thema der Applikations-Programmierung unter Eclipse. Zum Zwecke der Erleichterung der Erstellung von Client-Applikationen wurde ein Framework entwickelt, auf dessen Basis Applikationen mit relativ geringem Aufwand erstellt werden können. Dieses Framework stellt den Hauptteil der Arbeit dar und die Erläuterungen decken dabei die erwähnten Aspekte der Client-Programmierung unter Eclipse weitreichend ab. Dabei richtet sich die Arbeit vor allem an Entwickler[1], die sich mit dem Thema Client-Applikationen unter Eclipse auseinandersetzen wollen und schnelle Lösungen sowie kurze Entwicklungszeiten anstreben.

Im Verlauf dieser Arbeit werden zunächst die Problemstellungen und die damit verbundene Motivation zur Entwicklung des Frameworks erläutert. Dabei wird vor allem auf die Problematik bei der Entwicklung von Client-Applikationen unter Eclipse eingegangen. Anschließend werden didaktische Aspekte der Arbeit näher beleuchtet, wobei die Vereinfachung und Erleichterung der Client-Applikations-Entwicklung auf Basis von Eclipse bei Verwendung des entwickelten Frameworks aufgezeigt werden. Zur besseren Einordnung in das Umfeld werden bereits vorhandene Ansätze und deren Vor- und Nachteile diskutiert, wobei auch Parallelen und Differenzen herausgearbeitet werden. In weiterer Folge werden die verwendeten Methoden und Modelle näher beleuchtet, was vor allem dem Verständnis des entwickelten Frameworks dienen soll, welches auf diesen Modellen basiert. Dieser Abschnitt deckt damit die Grundlagen für die datengesteuerte Erstellung von Eclipse-basierten Client-Applikationen ab.

Der Hauptteil der Arbeit widmet sich dem Framework, wobei hier die praktische Anwendung im Vordergrund steht. Es werden Architektur, Funktionen und, vor allem für den Entwickler interessante, Implementierungen anhand eines Beispiels erläutert.

[1] Die Formulierungen „Entwickler“, „Programmierer“, „Benutzer“, usw. sind willkürlich gewählt, wobei grundsätzlich immer beide Geschlechter angesprochen werden

1.1 Problemstellung

Die Entwicklung von Applikationen, die lokal auf dem Computer eines Benutzers ausgeführt werden, erfordert oftmals viel Aufwand bei der Planung sowie entsprechendes Know-how. Der Programmierer will in der Regel eine Applikation entwickeln, die dem Benutzer ein effizientes und befriedigendes Arbeiten ermöglicht. Meist stehen ihm jedoch diverse Probleme gegenüber, die erst gelöst werden müssen, um dieses Ziel zu erreichen. Es müssen sehr viele Aspekte wie Design, Layout, Datenbeschaffung und -management, Prozessabwicklung oder Fehlerbehandlung berücksichtigt werden. Unter Verwendung des Eclipse-Frameworks und dessen Werkzeugen zur Entwicklung von Client-Applikationen, können viele dieser Aufgaben mit relativ geringem Aufwand gelöst werden. Wie jedoch bei jeder größeren und komplexeren Software, ist auch hier eine nicht zu vernachlässigende Einarbeitungszeit erforderlich. Zudem ist ein großes Maß an Know-how erforderlich, um alle Funktionen des Eclipse-Frameworks sinnvoll nutzen zu können. Ein Hauptproblem bei der Erstellung von Client-Applikationen kann das Datenmanagement darstellen. Dieses kann von Applikation zu Applikation verschieden sein, wenn Datenkomplexität, Datenvielfalt und Datenmenge variieren. Wenn der Benutzer der Applikation mit der Handhabung von Daten konfrontiert wird, so sollte ihm ein einfaches und intuitives Management der Daten zur Verfügung stehen; Dies umfasst vor allem die Navigation und Bearbeitung der Daten. Dabei ist es grundsätzlich dem Programmierer überlassen, welche Daten in welcher Form verwaltet werden sollen. Jedoch ist es mit herkömmlichen Mitteln nicht einfach möglich, dem Entwickler diese Arbeit in weiten Bereichen zu ersparen, was im folgenden Beispiel verdeutlicht werden soll.

1.2 Beispiel

Als einfaches Beispiel soll eine Applikation dienen, die Personendaten in einer Datenbank verwalten kann. Es muss eine Applikation entwickelt werden, die Daten aus der Datenbank lesen, verarbeiten und wieder in diese zurückschreiben kann. Die Anbindung einer Datenbank stellt in der Regel kein großes Problem dar, es stehen hier bereits in den meisten Programmiersprachen für die gängigen Datenbanksysteme vorgefertigte Schnittstellen zur Verfügung. Das Problem beginnt bei der Darstellung und Verwaltung der Daten in der Applikation. Hier müssen viele Aspekte, wie das richtige Design, eine intuitive grafische Benutzeroberfläche und natürlich die Prozesse, die die Applikation abdecken soll, berücksichtigt werden. Solche Prozesse können beispielsweise das Öffnen, Speichern, Löschen oder Umbenennen von Personen und deren Daten sein. Es muss also jeder dieser Prozesse über die Datenbankschnittstelle abgewickelt werden und es muss eine entsprechende Fehlerbehandlung vorhanden sein. Es wäre zum Beispiel denkbar, dass die Datenbank nicht verfügbar ist, oder die entsprechenden Personen aufgrund von Zugriffsberechtigungen oder Verknüpfungen zu anderen Daten nicht bearbeitet werden können; Diese Probleme müssen in der Applikation berücksichtigt werden

Zu Vereinfachung wäre hier ein Werkzeug von Vorteil, welches die Datenverwaltung übernimmt und lediglich über Konfigurationen oder geringen Programmieraufwand gesteuert wird.

Dies stellt gleichzeitig die Motivation für diese Arbeit dar, wie im Folgenden näher erläutert wird.

1.3 Motivation

Die Motivation für diese Arbeit hat sich vor allem durch die in Abschnitt 1.1 erläuterte Problemstellung der Datenverwaltung in Client-Applikationen unter Eclipse ergeben. Vor allem auch durch die Ähnlichkeiten der Anforderungen an Client-Applikationen ist die Motivation für eine Vereinheitlichung und Vereinfachung von Standard-Prozessen (siehe Beispiel in Abschnitt 1.2) erwachsen. Zudem ist auch der Wunsch nach Abstraktion komplexer Programmierung der Benutzerinteraktivität entstanden, also eine Vereinfachung für den Entwicklungsprozess von Standard-Aktionen, die immer wiederkehren oder sich ähneln.

1.4 Thesen der Arbeit

Das Ziel dieser Arbeit besteht darin, den in den vorherigen Kapiteln beschriebenen Problemen entgegenzuwirken. Es soll gezeigt werden, dass es möglich ist, mit relativ geringem Aufwand Client-Applikationen unter Eclipse zu entwickeln, ohne dabei zu tief in das sehr komplexe Thema eintauchen zu müssen. Daraus ergibt sich die erste These dieser Arbeit:

„Es ist möglich, Client-Applikationen unter Eclipse, mit Anbindung verschiedener Datenquellen, auf einfache Art und Weise zu erstellen, ohne dabei großes Vorwissen oder Know-how zu besitzen.“

Zur Erreichung dieses Ziels soll eine Abstraktionsebene geschaffen werden, die es dem Entwickler ermöglicht, eine Client-Applikation ohne tiefe Einsicht in das Thema erstellen und betreiben zu können. Hierzu soll ein Framework entwickelt werden, das diese Anforderungen erfüllt und darüberhinaus dem Entwickler die Möglichkeit bietet, die erstellten Applikationen mit Standard-Werkzeugen der Eclipse Plattform weiterzuentwickeln. Desweiteren soll gezeigt werden, dass unter Verwendung des Frameworks der Umgang mit den Eclipse-Werkzeugen zu Erstellung von Client-Applikationen erleichtert und verständlicher wird. Der Entwickler soll sich einfacher in das Thema einarbeiten können und dabei wichtige Informationen zu den üblichsten Anwendungsfällen für die Client-Applikations-Entwicklung unter Eclipse erhalten.

Diese Erleichterung bei der Erlernung des Eclipse Frameworks stellt die zweite These dieser Arbeit dar:

„Das entwickelte Framework ist geeignet, Entwicklern das Erlernen des Eclipse Frameworks zu erleichtern.“

Die aufgestellten Thesen werden im weiteren Verlauf noch genauer betrachtet und sollen im Rahmen dieser Arbeit entsprechend validiert werden.

2 Anforderungen

Wie bereits in Abschnitt 1.1 angesprochen, kann das Entwickeln von Client-Applikation äußerst aufwändig sein und dabei viel Know-how erfordern. Um die Problemstellungen fassbarer zu machen, werden in den nächsten Abschnitten die Anforderungen an den Entwickler sowie die Applikation, vor allem in den Bereichen Java und Eclipse, näher behandelt.

2.1 Anforderungen an die Applikation

Eine Client-Applikation wird in der Regel von Personen verwendet, die damit sicher, effizient und befriedigend umgehen wollen. Der Entwickler muss sich daher Gedanken zu der Benutzeroberfläche machen und diese entsprechend designen. Gängige Prozesse müssen modelliert und mögliche Fehler behandelt werden. Neben diesen Arbeitsschritten müssen auch Daten berücksichtigt werden, sofern die Applikation Daten verwalten soll. Diese Daten werden üblicherweise über eine grafische Benutzeroberfläche zugänglich gemacht, weshalb zur Verwaltung auch hier die bereits angesprochenen Prozesse (siehe 1.2) und möglichen Fehler bedacht werden müssen.

Gerade die Benutzerschnittstelle, auch „User-Interface“ genannt, bedarf besonderer Aufmerksamkeit. Es gibt zu dieser Thematik bereits eine Fülle von Ansätzen zur Gestaltung von (grafischen) Benutzeroberflächen.[2] Den Design-Prinzipien von IBM zufolge soll beispielsweise eine Benutzerschnittstelle intuitiv bedienbar und vorhersehbar sein, Funktionen einfach zugänglich machen, dem Benutzer die Kontrolle über die Funktionen in üblichen und gebräuchlichen Abläufen geben, sich das bisherige Know-how des Benutzers zu Nutze machen, sich an die Bedürfnisse des Benutzers anpassen können und zu jeder Zeit ein befriedigendes Arbeiten ermöglichen.[3]

Neben den Anforderungen an die Benutzerschnittstelle ergeben sich auch technische Ansprüche an die Applikation, wie die eigentliche Umsetzung sowie die zu

[2] vgl.: GUI Design - Linktipps zu Usability
[3] IBM Design principles

verwendenden Werkzeuge. Diesen muss der Entwickler, der mit der Umsetzung der Applikation betraut ist, genügen.

2.2 Anforderungen an den Entwickler

Der Entwickler muss die in Abschnitt 2.1 beschriebenen Applikations- sowie technischen Anforderungen erfüllen.

Wenn eine Client-Applikation mit Hilfe des Eclipse-Frameworks erstellt werden soll, so lassen sich viele der Anforderungen an die Benutzeroberfläche einfach erfüllen, da das Eclipse-Framework bereits eine Fülle an vorgefertigten Elementen zu Erstellung dieser User-Interfaces bietet. Problematischer ist die tatsächliche Programmierung und Umsetzung, die eine genaue Kenntnis des Frameworks und dessen Werkzeuge voraussetzt. In jedem Fall muss der Entwickler Java (siehe 5.1) beherrschen, sowie mit dem Standard Widget Toolkit („SWT“, siehe 5.3) umgehen können. Desweiteren muss er die Eclipse- und Plug-In-Architektur (siehe 5.5.3) verstehen und daran anknüpfen können.

Dasselbe gilt für die Anbindung von Datenquellen an die Applikation, wie beispielsweise einer Datenbank (siehe Beispiel in Abschnitt 1.2). Wie im Verlauf der Arbeit noch gezeigt werden wird, müssen hier viele Aspekte beachtet werden, um Daten aufzubereiten und dem Benutzer zugänglich zu machen. Der Entwickler muss alle Aktionen, die die eingebetteten Daten betreffen, nicht nur auf Seite der Benutzeroberfläche, sondern auch auf Datenbankebene abwickeln. Er muss also sehr viele Aspekte bei der Erstellung von Client-Applikationen, auch unter Verwendung des Eclipse-Frameworks, beachten und ein großes Wissen mitbringen.

Im weiteren Verlauf sollen Lösungsvorschläge für die einfachere Erstellung von Client-Applikationen unter Eclipse, vor allem unter Erfüllung der angesprochenen Anforderungen, dargebracht werden.

3 Lösungsvorschläge

In Abschnitt 1.4 sind bereits erste Ziele und Lösungsvorschläge angedeutet worden, die im Folgenden nun erweitert und detaillierter beschrieben werden.

Um den angesprochenen Problemen entgegenzutreten und den Anforderungen (siehe Kapitel 2) gerecht zu werden, soll ein Framework entwickelt werden, mit dessen Hilfe auf einfache Weise Client-Applikationen mit externer Datenanbindung erstellt werden können. Dabei müssen die Ansprüche an die zu erstellende Applikation herunter gebrochen und auf das zu entwickelnde Framework umgelegt werden. Anforderungen an die grafische Benutzeroberfläche müssen ebenso erfüllt werden, wie solche an die Anbindung von Datenquellen und Dateninteraktion. Da das Eclipse-Framework, auf dessen Basis gearbeitet werden soll, bereits eine Fülle an Werkzeugen (siehe SWT 5.3 und JFace 5.4) bereitstellt, die Teile der Richtlinien für das User-Interface-Design[4] umsetzen, liegt der Hauptschwerpunkt des zu entwickelnden Frameworks in der Anbindung und Verwaltung von Datenquellen sowie der Interaktion mit diesen Daten.

Es soll eine Abstraktionsebene zwischen Entwickler und der tatsächlichen Programmierung und Konfiguration der Client-Applikation geschaffen werden, die es dem Entwickler erlaubt, ohne tiefe Einsicht in das komplexe Thema, ansprechende Anwendungen erstellen zu können.

Nach reiflichen Überlegungen ergeben sich für das Framework und dessen Funktionalitäten folgende Hauptanforderungen, die im Folgenden näher erläutert werden sollen:

- Verständlichkeit
- Bedienbarkeit
- Generalisierung
- Flexibilität
- Erweiterbarkeit
- Integrierbarkeit
- Performance

[4] Eclipse User Interface Guidelines

3.1 Verständlichkeit

Unter dem Punkt der Verständlichkeit lassen sich zwei Arten von „Verständnis“ einordnen: Das Verständnis des Entwicklers, der die Software erstellt sowie das Verständnis des Benutzers, der die Software anschließend verwendet. Diese zwei Verständnisse unterscheiden sich grundlegend, da sich das erste auf die Software-Architektur und den Programmcode, und das zweite auf die fertige Applikation und deren Funktionen richtet.

3.1.1 Verständlichkeit der Software-Architektur und des Programmcodes

Um mit dem Framework effizient arbeiten zu können, muss die grundlegende Architektur verstanden werden. Hier spielen vor allem die Erweiterbarkeit (siehe 3.5) und Integrierbarkeit (siehe 3.6) entscheidende Rollen. Gerade wenn der Entwickler die Applikation um weitere Funktionen ergänzen will, müssen diese gegeben sein.

Dabei sollte die Architektur leicht verständlich und die einzelnen Aufgabenbereiche klar von einander abgetrennt sein. Die Interaktion der Komponenten miteinander sollte in Form von leicht verständlichen Diagrammen dargestellt sein.

Um das Framework ohne größeren Aufwand erweitern zu können (siehe 3.5) muss auch der Programmcode verständlich, klar gegliedert und mit Hilfe von JavaDoc [5] dokumentiert sein. Desweiteren soll sich der Code an den Eclipse-Namens-Konventionen [6] ausrichten, um das einfachere Anknüpfen und Erweitern des Frameworks zu gewährleisten.

3.1.2 Verständlichkeit der Client-Applikation

Für den Benutzer der Client-Applikation sind die zugrunde liegende Architektur sowie der Programmcode in der Regel irrelevant. Hier spielt das Verständnis der Programmoberfläche, der zur Verfügung stehenden Funktionen und zu setzenden Aktionen eine weit wichtigere Rolle. Der Benutzer sollte einfach in der Lage sein, die Funktionen zu begreifen sowie die möglichen Aktionen problemlos auszuführen.

---

[5] Javadoc Tool
[6] Eclipse Naming Conventions

3.2 Bedienbarkeit

Die Bedienbarkeit lässt sich ebenfalls in zwei Teilbereiche zerlegen, die wiederum den Entwickler und den Benutzer der Software betreffen.

3.2.1 Bedienbarkeit des Frameworks

Grundsätzlich stützt sich ein großer Teil der leichten Bedienbarkeit des Frameworks auf die verständliche Erklärung seiner Funktionalität (siehe 3.1.1). Die einzelnen Funktionen müssen klar abgegrenzt und einfach zu bedienen sein. Für die Bedienung des Frameworks kann hier die Verwendung von Standards wie XML (siehe 5.2) oder SWT (siehe 5.3) von Vorteil sein, da auf eventuell vorhandenes Know-how des Entwicklers zurückgegriffen werden kann. Sind diese Standards noch unbekannt, so lassen sich Dokumentationen zu diesen, beispielsweise über das Word Wide Web, leicht abrufen. Desweiteren soll es mit einfachen Mitteln möglich sein, innerhalb von Eclipse das Framework einzusetzen, um damit eine Client-Applikation entwickeln zu können. Unter Verwendung von Eclipse als Basis, ist dies leicht zu verwirklichen und gewährleistet eine vollständige Integration.

3.2.2 Bedienbarkeit der Client-Applikation

Die Bedienbarkeit der Client-Applikation leitet sich ebenfalls zum Teil aus dem Verständnis (siehe 3.1.2) ab. Zu dem kommt die Einhaltung von Design-Richtlinien für Benutzerschnittstellen, die dem Anwender der Applikation eine einfache und intuitive Bedienung ermöglichen sollen.

Wie im weiteren Verlauf der Arbeit noch gezeigt wird, deckt das Eclipse-Framework mit seinen Werkzeugen und vordefinierten Benutzerschnittstellen bereits einen Großteil gängiger Design-Richtlinien ab. Trotzdem soll die zu erstellende Client-Applikation die Design-Richtlinien der Eclipse Plattform[7] befolgen, um dem Benutzer ein einfaches Arbeiten zu ermöglichen.

[7] Eclipse User Interface Guidelines

3.3 Generalisierung

Da das Hauptaugenmerk hier auf der Anbindung von Datenquellen liegt, muss das zu entwickelnde Framework möglichst generisch programmiert werden. Es soll dem Benutzer die Möglichkeit gegeben werden, die zur Verfügung stehenden Schnittstellen zu nutzen und eigene Implementierungen auf Basis seiner Anforderungen vorzunehmen. Da gerade die Datenquellen variieren können, muss das Framework in allen Bereichen der Datenverwaltung generisch sein und nach außen hin Schnittstellen zum Anknüpfen der Daten bereitstellen.

Die Generalisierung lässt sich dabei auf drei Teilbereiche herunter brechen:

- Generalisierung der Datenquellen
- Generalisierung der einzelnen Datenobjekte
- Generalisierung der Aktionen auf die Datenobjekte

Diese stellen die Hauptaufgabe des Frameworks dar und sollen im Folgenden näher erläutert werden.

3.3.1 Generalisierung der Datenquellen

Da meist erst zum Zeitpunkt der Planung oder tatsächlichen Entwicklung der Client-Applikation bekannt ist, aus welcher Datenquelle die Daten stammen werden, muss die Anbindung an diese Datenquellen auf Seite des Frameworks generisch sein. Es müssen Schnittstellen definiert werden, die es erlauben, sämtliche geeigneten Datenquellen anzubinden. Solche können beispielsweise Datenbanken, Web-Services oder Dateisysteme sein. Die Implementierung soll vom Entwickler vorgenommen werden, wobei für gängige Datenquellen eine beispielhafte Implementierung vorgenommen werden soll.

3.3.2 Generalisierung der einzelnen Datenobjekte

Die Datenobjekte, die in der Applikation verwaltet werden, müssen auf Basis der Datenquelle ebenfalls generisch gestaltet werden. Da auch hier verschiedenste Objekte verwaltbar sein sollen, muss auch hier eine Implementierung vom Entwickler

vorgenommen werden. Auf Seiten des Frameworks müssen alle datenobjektbezogenen Abläufe abstrakter Natur sein, um mit den verschiedenen Objekten nach der Intention des Entwicklers umgehen zu können. Solche Datenobjekte können beispielsweise Datenbankeinträge, Ergebnisse eines Web-Services oder Dateien sein.

3.3.3 Generalisierung der Aktionen auf Datenobjekte

Aktionen oder Befehle auf Datenobjekte, sind ebenfalls generisch zu halten. Der Entwickler muss selbst festlegen können, welche Aktionen unter welchen Bedingungen ermöglicht werden, und wie diese Aktionen ablaufen sollen. Hier könnten das Löschen eines Datenbankeintrags, das Abrufen eines Web-Services oder das Öffnen einer Datei als Beispiele genannt werden. Auch diese Aktionen stehen dem Entwickler frei und müssen an Schnittstellen des Frameworks angeknüpft werden. Der Hauptfokus soll hier auf Standard-Aktionen liegen, die auf Datenobjekten üblicherweise ausgeführt werden können. Solche Standard-Aktionen könnten beispielsweise sein:

- Öffnen
- Bearbeiten
- Speichern
- Anlegen
- Löschen
- Umbenennen
- Aktualisieren

Das Framework soll diese Standard-Aktionen bereits vorsehen und einfache Anknüpfungspunkte und Konfigurationsmöglichkeiten bieten.

Wenn der Entwickler weitere Aktionen implementieren möchte, soll dies ebenfalls über ähnliche Schnittstellen und auch unter Verwendung von Eclipse-Werkzeugen möglich sein (siehe 3.5)

3.4 Flexibilität

Das Framework muss vor allem in Bezug auf die anzubindenden Datenquellen flexibel sein, was durch die angestrebte Generalisierung unter Punkt 3.3 bereits erläutert wurde.

Außerdem muss dem Entwickler für die Gestaltung der Benutzeroberfläche ebenfalls ein gewisser Freiraum gegeben werden, um flexible und anforderungsspezifische Benutzerschnittstellen programmieren zu können. Durch den Einsatz des Eclipse-Frameworks und Werkzeugen wie SWT (siehe 5.3) und JFace (siehe 5.4) sind diese Voraussetzungen bereits teilweise erfüllt.

3.5 Erweiterbarkeit

Eine äußerst wichtige Anforderung an das Framework stellt die Erweiterbarkeit dar. Ziel ist hier, mögliche Schnittstellen des Eclipse-Frameworks zu nutzen und verfügbar zu machen sowie Schnittstellen nach außen hin zu definieren, um dem Entwickler Anknüpfungspunkte zu bieten.

Eine Client-Applikation, die mit Hilfe des Frameworks erstellt worden ist, soll sich nahtlos in das Eclipse-Plug-In-Development (siehe 5.5.3) einfügen und mit Hilfe der Eclipse-eigenen Werkzeuge weiterentwickelt werden können. So soll es beispielsweise möglich sein, Aktionen unabhängig von der Datenquelle auch über die Eclipse-eigenen Werkzeuge implementieren oder zusätzliche Funktionen über Plug-Ins einbinden zu können. Schnittstellen, die von Eclipse angeboten werden, sollen somit problemlos implementierbar sein.

Für alle generischen Aufgaben innerhalb des Frameworks, müssen Schnittstellen definiert werden, um die Standard-Implementierungen jederzeit durch eigene ersetzen zu können. Das hat den Vorteil, dass auch spezielle Anforderungen erfüllt werden können, ohne an die Standard-Implementierungen gebunden zu sein. Dieses Konzept kann vor allem bei der Erweiterung um zusätzliche Datenquellen, als die bereits standardmäßig implementierten, von Nutzen sein. Dasselbe gilt für die Gestaltung der Benutzer-Oberflächen. Der Entwickler muss jederzeit die Möglichkeit haben, Standard-

Oberflächen und -Funktionen durch spezialisierte zu ersetzen – und das mit möglichst geringem Aufwand.

3.6 Integrierbarkeit

Die Integrierbarkeit ist neben der Erweiterbarkeit ein weiterer wichtiger Anspruch. Hier kann wiederum zwischen der Integration des Frameworks und der Integration der erstellten Client-Applikation unterschieden werden.

3.6.1 Integrierbarkeit des Frameworks

Das Framework muss sich in die Eclipse-Architektur einfügen, um somit auch von anderen Teilen von Eclipse nutzbar zu sein. Durch die Verwendung von Eclipse-Werkzeugen soll das Arbeiten mit dem Framework möglich sein, wobei hier vor allem die Integration in die Entwicklungs-Oberfläche von Bedeutung ist.

3.6.2 Integrierbarkeit der Client-Applikation

Neben der Integration des Frameworks ist auch die Integration der erstellten Client-Applikationen in andere Anwendungen von Bedeutung. Es soll beispielsweise möglich sein, Teile einer Applikation, die nur den generischen Datenquellen-Teil betreffen, in andere Anwendungen zu integrieren. Dies soll grundsätzlich ebenfalls über die Eclipse-eigenen Werkzeuge realisierbar sein. Der Entwickler muss hier allerdings mehr Einsicht in die Architektur besitzen, um die erforderlichen Konfigurationen an den richtigen Stellen vornehmen zu können.

3.7 Performance

Client-Applikationen, die mit Hilfe des Frameworks entwickelt worden sind, müssen auch performant sein. Da hier Eclipse als Basis dient, müssen Optimierungen an der Performance in den Bereichen der Datenanbindung, Datenverwaltung sowie der Benutzerschnittstellen vorgenommen werden.

Im Folgenden werden die wichtigsten Konzepte der Performance-Optimierung, die für das Framework relevant sind, erläutert.

3.7.1 Skalierbarkeit

Unter Skalierbarkeit versteht man die Eigenschaften und das Verhalten von Programmen, wenn die Datenmengen steigen.[8]

Die Client-Applikation darf bei größeren Datenmengen nicht überproportional langsamer werden oder Performance einbüßen. Das trifft vor allem die zur Verwaltung verwendeten Datenstrukturen sowie die grafische Benutzeroberfläche.[8]

3.7.2 Speicherauslastung

Die Speicherauslastung muss für die zu verwaltenden Daten und das User-Interface angemessen sein und darf nicht darüber hinauswachsen. Wenn die Speicherauslastung zu groß wird, dann kann dies zur Folge haben, dass die Applikation sehr langsam wird, gar nicht mehr reagiert oder sich sogar unsachgemäß beendet.[9]

3.7.3 Design und Coding

Für die Performance einer Applikation kann auch das richtige Design entscheidend sein. Hier sollen vor allem objektorientierte Konzepte umgesetzt werden, um die bereits unter Punkt 3.7.1 angesprochene Skalierbarkeit und Wartbarkeit zu erreichen.[10]

Desweiteren sollen effiziente und geeignete Datenstrukturen eingesetzt werden, um möglichst performante Zugriffe zu erzielen. Für unterschiedliche Anforderungen müssen auch unterschiedliche Datenstrukturen eingesetzt werden. Dasselbe gilt für Algorithmen, die möglichst effizient implementiert werden sollen.[11]

Ein weiterer wichtiger Aspekt ist das saubere und effiziente Coding. Hier muss der richtige Einsatz von Objekten und Datenstrukturen einerseits und die Wartbarkeit andererseits berücksichtigt werden.[12]

---

[8] vgl.: Java Platform Performance Strategies and Tactics, S.6
[9] vgl.: Java Platform Performance Strategies and Tactics, S.53ff
[10] vgl.: Java Platform Performance Strategies and Tactics, S.12
[11] vgl.: Java Platform Performance Strategies and Tactics, S.103ff
[12] vgl.: Java Platform Performance Strategies and Tactics, S.85ff

3.7.4 Logging

Das Framework soll zur leichteren Handhabung ein effizientes Logging-System integrieren. Dem Entwickler soll dieses Logging-System ohne großen Aufwand nutzbar und bei Bedarf erweiterbar oder konfigurierbar gemacht werden.

3.8 Technische Anforderungen

Das Framework soll auf Basis von Eclipse in der Version 3.3 („Europa“)[13] programmiert werden und dessen Schnittstellen und Werkzeuge nutzen.

Als Programmiersprache wird Java in der Version 6[14] dienen.

Alle Konfiguration sollen in XML verfasst und gehandhabt werden.

[13] Eclipse.org
[14] Java 6 Standard Edition

4 Didaktische Aspekte

Das Framework soll neben den in Kapitel 3 beschriebenen Anforderungen auch dazu geeignet sein, durch entsprechende Dokumentation, das Eclipse Framework einfacher erlernen zu können. Das Hauptaugenmerk liegt hierbei auf der Client-Applikations-Entwicklung sowie den gängigen Funktionen von Eclipse.

Dieses Kapitel beschäftigt sich mit didaktischen Überlegungen, wobei Ziele und Zielgruppen sowie die Methodik definiert werden, mit der die Inhalte vermittelt werden sollen.

4.1 Didaktische Vorüberlegungen

Im Folgenden werden didaktische Vorüberlegungen erläutert, wie Zielgruppe, Eingangsvoraussetzungen sowie grundsätzliche didaktischen Ziele, die mit dieser Arbeit erreicht werden sollen.

4.1.1 Zielgruppe

Die Zielgruppe des entwickelten Frameworks sind Programmierer, die sich mit der Client-Programmierung unter Eclipse sowie der Eclipse Plattform an sich auseinandersetzen wollen. Hier muss angemerkt werden, dass die Zielgruppe auch solche Programmierer umfasst, die mit dem Framework lediglich schnell und effizient Client-Applikationen entwickeln möchten – hier spielen didaktische Überlegung aber eine untergeordnete Rolle, da in diesem Fall die Ziele des Entwicklers über das Anwenden des Frameworks nicht oder nur bedingt hinausgehen. Daher soll hier primär auf die erste Zielgruppe eingegangen werden, die das grundlegende Verständnis von Eclipse sowie das Erlernen dessen Funktionen zum Ziel hat.

Die Zielgruppe lässt sich weiter auf jene Entwickler einschränken, die bereits Erfahrung mit Java besitzen und sowohl Java-Code als auch Code-Kommentare verstehen und mit Code-Dokumentationen umgehen können. Diese Einschränkung erfolgt deshalb, weil Großteile des erklärten Codes sehr technisch sind und davon auszugehen ist, dass dieser

von Laien nicht verstanden wird. Zudem sind alle Code-Erläuterungen in englischer Sprache und verwenden Fachausdrücke aus den Bereichen Java, XML und Eclipse.

4.1.2 Eingangsvoraussetzungen

Wie bereits in Abschnitt 4.1.1 erläutert, muss der Entwickler, der das Framework einsetzen will, die Programmiersprache Java und deren Sprachfeatures beherrschen. Dabei sollte zumindest Java in der Version 5.0 beherrscht werden. Desweiteren muss der Entwickler Code-Kommentare verstehen und mit der Code-Dokumentation *JavaDoc* umgehen können. Das ist besonders bei der Erweiterung und Modifikation des Frameworks erforderlich, da hier die Beschreibung der Schnittstellen wesentlich ist. Zudem sollte der Entwickler in der Lage sein, selbständig arbeiten und lernen zu können.

4.1.3 Ziele

Die didaktischen Ziele lassen sich grob in zwei Hauptpunkte gliedern:

- Erlernen und Verstehen der Architektur sowie der Funktionen des entwickelten Frameworks
- Erlernen und Verstehen der Architektur und der Funktionen von Eclipse

Der erste Punkt bezieht sich auf das entwickelte Framework und weniger auf die Eclipse Plattform. Hier sollen die Architektur sowie die Funktionen einfach erlernt und Client-Applikationen mit wenig Aufwand erstellt werden können. Da das Framework auf Eclipse basiert, lassen sich auch Parallelen zwischen den beiden Punkten erkennen. Um mit dem Framework arbeiten zu können, müssen auch die Eclipse-Architektur sowie die Funktionen zumindest grob verstanden werden. Soll tiefer in die Applikationsebene eingegriffen werden, so müssen diese genauer bekannt sein. Dies soll vor allem mit dem praktischen Teil dieser Arbeit ermöglicht werden, in dem viele der Funktionen implementiert und erklärt sind. Ziel ist es, dem Entwickler einen breiten Einblick in das Thema der Client-Applikations-Entwicklung unter Eclipse zu geben, mit besonderer Berücksichtigung üblicher Problemstellungen.

4.2 Inhalte

Um die in 4.1.3 grob definierten Ziele erreichen zu können, müssen entsprechende Inhalte vermittelt werden. Dabei lassen sich die Inhalte wiederum in Inhalte zum Thema Eclipse sowie Inhalte für das entwickelte Framework aufteilen.

Da der praktische Teil der Arbeit das selbstentwickelte Framework darstellt, widmet sich ein Großteil der Arbeit der Beschreibung der Architektur, der Funktionen sowie der Implementierungen.

Das Framework ist darüberhinaus geeignet, Entwicklern Einsicht in viele Eclipse-eigene Funktionen zu geben und die damit verbundenen Inhalte zu vermitteln.

Im Folgenden werden die einzelnen Inhalte erläutert, die zum Teil für das Erlernen des entwickelten Frameworks und zum Teil für das Erlernen der Eclipse Plattform geeignet sind.

4.2.1 Gemeinsame Inhalte

Zu den gemeinsamen Inhalten zählen diejenigen, welche große Überschneidungen zwischen dem Framework und Eclipse aufweisen. Dazu zählen:

- Java
- XML
- Eclipse Plattform (Architektur)

Java ist für das Verstehen beider Komponenten erforderlich, da diese darauf basieren. Daher ist es Grundvoraussetzung, dass der Entwickler Java beherrscht. Hier spielen nicht nur die Syntax der Sprache eine Rolle, sondern auch die Komponenten von Java, die für die Verwendung der beiden Frameworks erforderlich sind. Inhaltlich soll nur auf die wichtigsten Java-Komponenten eingegangen werden.

XML dient der Konfiguration der beiden Frameworks und wird ebenfalls vorausgesetzt. Die Inhalte beschränken sich hier auf die Erläuterungen zu den jeweiligen Konfigurationen.

Die Eclipse Plattform stellt die Basis für das Framework sowie für die Eclipse-Funktionen dar. Inhaltlich sollen dem Programmierer die Eclipse-Architektur sowie die Rich-Client-Platform nähergebracht werden.

4.2.2 Inhalte zu dem entwickelten Framework

Die Inhalte zu dem entwickelten Framework decken die im Folgenden beschriebenen Bereiche ab.

4.2.2.1 Architektur

Hier werden die Architektur und alle relevanten Schnittstellen des Frameworks erklärt. Das Verständnis der Architektur ist Voraussetzung für das Arbeiten sowie das Erweitern und Modifizieren des Frameworks und steht damit am Anfang des Erarbeitungs-prozesses.

4.2.2.2 XML-Konfigurationen

Die XML-Konfigurationen werden verwendet, um eine Client-Applikation zu definieren. Dabei stehen die Konfiguration von Datenquellen, der Navigation sowie der Aktionen auf Datenobjekte im Vordergrund. Der Entwickler soll auf Basis der Erläuterungen in der Lage sein, eigene Applikationen konfigurieren zu können.

4.2.2.3 Datenquellen-Anbindungen

Die Anbindung der Datenquellen wird sowohl in XML-Konfigurationen vorgenommen als auch in Java-Code implementiert. Das Verstehen der Schnittstellen zum Anbinden von Datenquellen steht somit hier im Vordergrund.

4.2.2.4 Navigationen

Die Definition der Navigationen sowie Navigations-Objekte wird ebenfalls in XML-Konfigurationen vorgenommen und kann, je nach Anforderung, auch Java-Implementierungen erfordern. Das Definieren und Verwenden solcher Navigationen wird noch genau erklärt werden.

4.2.2.5 Aktionen

Aktionen auf Datenobjekte werden sowohl in XML als auch in Java implementiert. Der Entwickler soll in der Lage sein, eigene Aktionen zu definieren und in seinen Applikationen einzusetzen.

4.2.3 Inhalte zu Eclipse-eigenen Funktionen

Das entwickelte Framework deckt die Verwendung einer Vielzahl der von Eclipse zur Verfügung gestellten Mittel ab und gibt dem Entwickler Einsicht in die Implementierung und Handhabung vieler Funktionen. Im Folgenden werden die implementierten und veranschaulichten Funktionen jeweils mit einer Beschreibung und dem Ziel tabellarisch dargestellt.

Funktion	**Beschreibung/Ziel**
Workbench	Verwenden der Eclipse-Oberfläche und dem Hauptfenster
Views	Konfigurieren und Implementieren von Views und Navigationen
Viewer Model	Erstellen von Modellen zur Navigation in einem View (Navigations-Objekte)
TreeViewer/FilteredTree	Definieren von TreeViewern zur Darstellung von Navigations-Objekten mit Such- und Sortierfunktionen (FilteredTree)
ContentProvider	Definieren des Inhalts eines TreeViewers
LabelProvider	Definieren der Darstellung der im TreeViewer verwalteten Elemente
Object-Listener	Reagieren auf Ereignisse von Navigations-Objekten
Mementos	Speichern des letzten Status der Applikation (Fensteranordnung, Menüs, Toolbars)
Editoren	Implementieren von Editoren in Eclipse

EditorInput	Verwenden von EditorInput zum Bearbeiten eines Elements in einem Editor
View-Editor-Kommunikation	Implementierungen zur Kommunikation zwischen View und Editor (Editor-View-Verlinkung, Umbenennungen)
Perspektiven	Konfigurieren und Verwenden von Perspektiven
Actions	Verwenden von Aktionen zum Ausführen von Befehlen
Jobs	Aktionen über das Eclipse Jobs API ausführen und dem Benutzer Prozessinformationen bieten
Menüs	Verwenden von Menüs und Untermenüs in der Hauptnavigation
Kontextmenüs	Implementieren von Kontextmenüs in Navigations-Views
Toolbars	Verwenden von Toolbars in Views und Editoren
Standard-Dialoge	Verwenden von Standard-Dialogen (OK/Abbrechen, Ja/Nein, Fehler-, Warnungs- und Informations-Dialoge)
Shortcuts	Verwenden von Shortcuts für gängige Funktionen wie Löschen, Umbenennen, Aktualisieren
RetargetAction	Verwenden von Standard-Aktionen (RetargetActions) in Eclipse
Wizards	Verwenden von Wizards zum Durchführen eines Prozesses
ImageDescriptor	Verwenden von Icons und Bildern unter Eclipse
Drag & Drop	Implementieren von Drag & Drop-Aktionen (Konfigurieren/Persistieren/Laden)
Fehlerbehandlung / Status-Objekte	Handhaben von Status-Objekten und Fehlermeldungen

Plug-In-Konfiguration	Konfigurieren von Eclipse-Plug-Ins
Erweiterung / Extension Points	Applikationen über Extension Points erweitern
Product-Konfiguration	Definieren und Konfigurieren von Eclipse Products inklusive Branding

Tabelle 4.1 - Framework Funktionen

4.3 Methodik

Die Methodik zum Vermitteln der in 4.2 definierten Inhalte besteht primär aus einer Anleitung zum Selbststudium. Diese Form ist aus dem Grund gewählt worden, da die Zielgruppe vor allem Software-Entwickler sind und davon ausgegangen werden kann, dass diese bereits Erfahrungen im Selbststudium von Software gesammelt haben. Da Software-Beschreibungen oftmals im Word Wide Web zu finden sind und ebenfalls auf ein Selbststudium abzielen, kann diese Form der Vermittlung von Inhalten an bereits gemachte Erfahrungen der Entwickler anknüpfen.

Desweiteren sollen die Inhalte, mit den im Folgenden genauer beschriebenen Methoden, Software-Entwicklern in adäquater Form präsentiert werden.

4.3.1 Benutzerhandbuch

Die Erläuterungen und Beschreibungen der Ergebnisse dieser Arbeit sollen genutzt werden, um die Eclipse Plattform besser verstehen und erlernen zu können. Dabei werden die in der Arbeit erläuterten Funktionen sowohl in bildlicher, als auch verbaler Form präsentiert, was das Verständnis vereinfachen soll. Dem Entwickler sollen dabei zunächst die Funktionen verbal nähergebracht und verständlich erklärt werden. Um die Auswirkungen einer Funktion besser verstehen zu können, wird des Öfteren eine veranschaulichende Abbildung beigelegt. Nach diesem Schritt hat der Entwickler die Möglichkeit, die erklärte Funktion implementiert im Framework anzusehen und bei Bedarf zu modifizieren (siehe 4.3.4). Damit ist eine schrittweise theoretische und praktische Annäherung an die Lernziele gewährleistet.

Die Beschreibungen der Architektur sind ebenfalls sowohl in bildlicher als auch verbaler Form vorhanden. Dabei sollen Prozess-Diagramme die Abläufe innerhalb des Frameworks verständlicher machen.

4.3.2 Beispiel-Applikation

Neben den bloßen Erläuterungen und Beschreibungen soll, zum besseren Verständnis des Frameworks, eine Beispiel-Applikation entworfen, entwickelt und beschrieben werden. Dazu wird eine Client-Applikation erstellt werden, welche das Verwalten von fiktiven Firmen, Managern und Angestellten ermöglichen soll, wobei eine fiktive Firma mehrere Manager und diese wiederum mehrere Angestellte haben können. Diese Daten sollen teilweise in einer Datenbank und teilweise am Dateisystem verwaltet werden. Die einzelnen Daten-Elemente (Firmen, Manager, Angestellte) sollen gelöscht, bearbeitet und neu angelegt werden können. Dabei soll für diese fiktive Firma ein Name definiert werden können, für Manager und Angestellte ist die Eingabe von Vor- und Nachname vorgesehen.

Das gewählte Beispiel soll dabei die Konfiguration der Datenquellen, Datenobjekte, Navigationen sowie Aktionen auf die Datenobjekte veranschaulichen. Desweiteren sollen die wichtigsten Funktionen und Aktionen anhand von Screenshots demonstriert werden.

Diese Beispielanwendung ist auch in implementierter Form Bestandteil des Frameworks und soll dem Entwickler, der sich damit auseinander setzt, die Möglichkeit bieten, die Beispiel-Implementierungen heranzuziehen, um die Abläufe innerhalb des Frameworks besser zu verstehen. Er soll damit in die Lage versetzt werden, auf Basis dieser Implementierungen, spezifischere Anforderungen umzusetzen und das Framework um eigene Implementierungen zu erweitern.

4.3.3 Code-Dokumentation (JavaDoc)

Einen weiteren Hauptaspekt stellt die Code-Dokumentation bei der Vermittlung der Inhalte dar. Das Framework ist vollständig dokumentiert, wobei die Beschreibungen an die Eclipse-Dokumentationen angelehnt sind. Die Code-Dokumentation (JavaDoc)

erklärt dabei in standardisierter Form, wie die einzelnen Klassen und Methoden vom Entwickler verwendet werden können. Neben der JavaDoc sind die wichtigsten Code-Stellen ebenfalls ausführlich „in-line“, also im Code selbst, dokumentiert, um Prozesse und Abläufe verständlicher zu machen.

Alle Kommentare sind in englischer Sprache verfasst und folgen den Richtlinien der Eclipse-Dokumentationen – das verbessert die Lesbarkeit und damit die Verständlichkeit der Dokumentation für erfahrene Java- und Eclipse-Entwickler.

4.3.4 Code-Präsentation (Implementierungen)

Die Basis-Implementierungen des Frameworks können herangezogen werden, um die in 4.2.3 beschriebenen Eclipse-Funktionen besser verstehen zu können. Der Entwickler hat die Möglichkeit, durch Einsicht in den Quellcode, die Funktionsweisen zu verstehen und deren Ausführungen zu erlernen. Diese Form der Aufbereitung, auch *Code-Snippets* genannt, wird für diverse Code-Dokumentationen eingesetzt und soll als Beispiel zur Implementierung dienen. Der Entwickler kann diese Beispiel-Implementierungen umwandeln und an eigene Anforderungen anpassen.

Die soeben erläuterten Methoden sollen dem Entwickler einen unkomplizierten Einstieg in das entwickelte Framework sowie die Eclipse Plattform bieten und dabei die wichtigsten Funktionen einfach zugänglich machen. Dies wird durch das komplette und ausführliche Benutzerhandbuch, die Beispielapplikation, die Code-Dokumentationen mit JavaDoc und in-line-Kommentaren sowie die beispielhaft implementierten Eclipse-Funktionen (siehe 4.2.3) erreicht. Wie bereits erwähnt, richtet sich die gesamte Dokumentation dabei an erfahrene Java-Entwickler, welche diese Übermittlungsformen kennen oder kennen lernen wollen. Durch den Einsatz dieser Standard-Hilfsmittel ist gewährleistet, dass sich Entwickler sofort zurechtfinden und das Framework sowie die Eclipse-Funktionen einfach verwenden können.

5 Grundlagen, Methoden und Modelle

Dieser Abschnitt widmet sich der Beschreibung der für das entwickelte Framework notwendigen Grundlagen sowie mit den verwendeten Methoden und Modellen. Zunächst werden die softwaretechnischen Grundlagen wie die Programmiersprache Java (Abschnitt 5.1), die Beschreibungssprache XML (Abschnitt 5.2), das Standard Widget Toolkit (Abschnitt 5.3), JFace (Abschnitt 5.4) sowie das Eclipse-Framework (Abschnitt 5.5) behandelt.

Das Verstehen dieser Grundlagen ist Voraussetzung für die weiteren Erläuterungen zu dem entwickelten Framework und der Analyse vorhandener Ansätze (siehe Kapitel 6).

5.1 Java

Java ist eine objektorientierte Programmiersprache und wurde im Jahr 1992 von James Gosling, Patrick Naughton, Chris Warth, Ed Frank und Mike Sheridan von Sun Microsystems, Inc. konzipiert, entwickelt und schließlich 1995 der Öffentlichkeit vorgestellt.[15] Einer der Hauptaspekte bei der Entwicklung von Java war es, Programme auf unterschiedliche Systeme zu portieren, um höchste Flexibilität zu gewährleisten. Obwohl Java ursprünglich für den Einsatz in mobilen Endgeräten vorgesehen war, ergab sich, durch das immer größer werdende Interesse am World Wide Web, die Möglichkeit, Java auch für Internet-Anwendungen zu verwenden. Es sollte dabei ermöglicht werden, Programmcode auf unterschiedlichen Plattformen wie Intel, Macintosh oder UNIX ausführen zu können.[16]

Java wurde in Anlehnung an die Programmiersprache *C* und die objektorientierte Programmiersprache *C++* entwickelt. Vor allem die ähnliche Syntax sollte Entwicklern den Einstieg in Java erleichtern. Trotz allem hat Java konzeptionell wenig mit C/C++ gemein, da hier unterschiedliche Ziele angestrebt werden.[17]

---

[15] The Complete Reference - Java2 5th Edition, S.7
[16] vgl.: The Complete Reference - Java2 5th Edition, S.8f
[17] The Complete Reference - Java2 5th Edition, S.8

Seit der Einführung von Java ist die Sprache unter der Leitung von Sun Microsystems Inc. immer weiter entwickelt worden und bietet ein breites Spektrum an Einsatzmöglichkeiten.[18]

Aktuell sind folgende Versionen von Java verfügbar:[19]

- Java Standard Edition (Java SE)
- Java Enterprise Edition (Java EE)
- Java für Rich Internet Applications (Java FX)
- Java Micro Edition (Java ME)
- Java für Smart Card Applications (Java Card)

Java ist objektorientiert und unterstützt dabei wichtige Konzepte wie Vererbung, Polymorphismus, Datenkapselung und Generizität (seit Version 5.0).[20]

Durch die Java-Technologie können Programme auf unterschiedlichen Betriebssystemen, wie z.B. Windows, Linux, Solaris oder Macintosh, ausgeführt werden, was zu einer der Kernkompetenzen zählt.

Die folgende Grafik zeigt die Architektur von Java sowie die einzelnen Komponenten:

[18] vgl.: The Complete Reference - Java2 5th Edition, S.10ff
[19] Sun Products
[20] vgl.: The Complete Reference - Java2 5th Edition, S.19ff

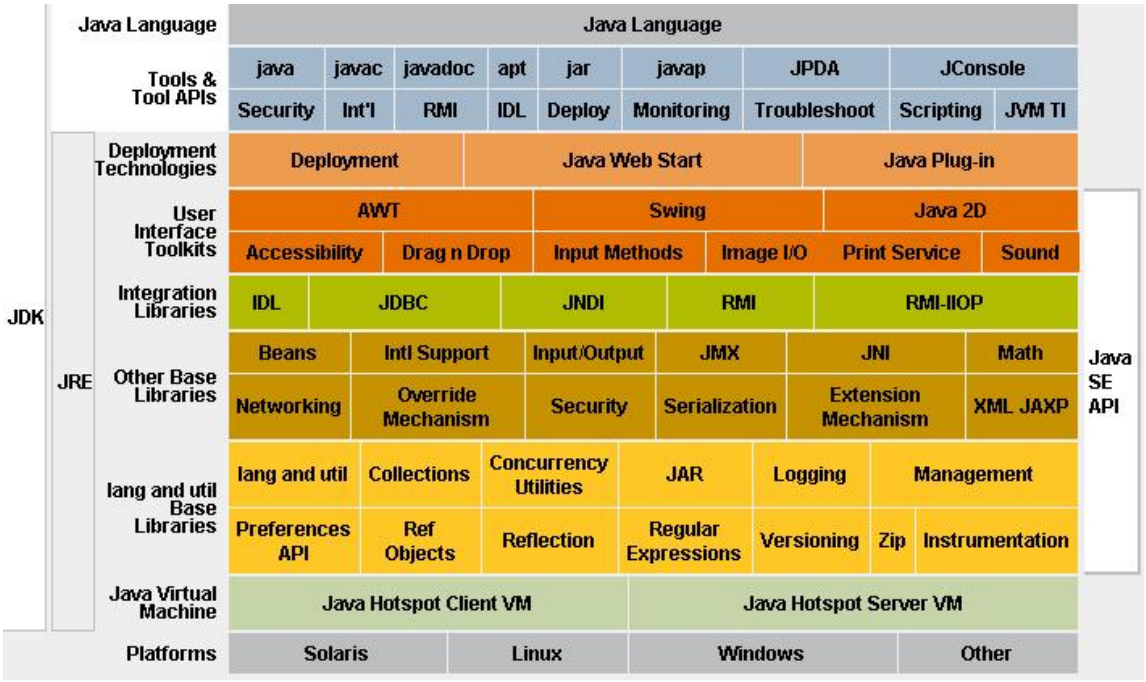

Abbildung 5.1 - Java Architektur

Java bietet neben den Kern-Komponenten weitere nützliche Funktionen wie grafische Benutzerschnittstellen mit Swing und AWT, Bildbearbeitung mit Image I/O, Java2D und Java3D, oder Sound-Bearbeitung an.[21] Für die Anbindung von Datenquellen dient JDBC (Java Database Connectivity), mit dessen Hilfe z.B. alle gängigen Datenbanksysteme angebunden werden können.[22] Desweiteren sind in Java nützliche Funktionen wie Mathematische Rechenfunktionen[23], Logging[24], Datenkomprimierung[25] oder XML-Verarbeitung[26] standardmäßig enthalten.

Speziell für Web-Anwendungen ist die Java Enterprise Edition (Java EE) entstanden. Diese bietet für das Entwickeln von Web-Applikationen zahlreiche Funktionen und Dienste, die bereits einen hohen Reifegrad erreicht haben.[27]

[21] Java Standard Edition – Technologies
[22] Java Standard Edition – JDBC
[23] Java Standard Edition – Math Functionality
[24] Java Standard Edition – Logging
[25] Java Standard Edition – Package Summary
[26] XML in the Java Platform Standard Edition
[27] Java Enterprise Edition – Technologies

5.2 XML

XML steht für "Extensible Markup Language" und ist eine Technologie zum Beschreiben von Daten.[28] Das Ziel der Sprache ist es, jede Art von Daten beschreiben zu können, um damit einen universellen Datenaustausch zu ermöglichen.[29] Dabei ist XML keine Sprache im eigentlichen Sinne, sondern vielmehr eine Möglichkeit, mit Hilfe einer vordefinierten Syntax, eine Sprache zu erstellen.[30] In XML verfasste Dokumente sind somit generisch und können grundsätzlich jede Art von Daten beschreiben.

Ein Beispiel wäre die Beschreibung von Personendaten, die aus „Vorname", „Nachname" und „Alter" bestehen sollen. Die XML-Darstellung könnte sich wie folgt ergeben:

```
<person>
     <vorname>Max</vorname>
     <nachname>Mustermann</nachname>
     <alter>35</alter>
</person>
```

In XML können, wie auch in dem Beispiel, Daten hierarchisch angeordnet werden. Die Tiefe der Hierarchie ist dabei unbeschränkt. Um das Beispiel zu erweitern, könnten zu jeder Person noch weitere Elemente mit zusätzlichen Sub-Elementen, wie beispielsweise Kinder hinzugefügt werden.[31]

XML hat sich in den letzten Jahren sehr weit verbreitet und wird von einer Vielzahl an Applikationen aus den unterschiedlichsten Bereichen eingesetzt. Vor allem für den universellen Datenaustausch zwischen Applikationen hat sich XML als Standard etabliert.[32] So sind beispielsweise Microsofts Office Dokumente in der Version 2007 bereits vollständig XML-basiert.[33]

---

[28] vgl.: Beginning XML, 3rd Edition, S.3
[29] vgl.: Beginning XML, 3rd Edition, S.6f
[30] Beginning XML, 3rd Edition, S.7
[31] vgl.: Beginning XML, 3rd Edition, S.15
[32] vgl.: W3C - Extensible Markup Language (XML)
[33] Microsoft Office (2007) Open XML-Dateiformate

Neben der reinen XML-Syntax gibt es noch viele weitere Teile wie DTD, Namespaces, XPath oder XSLT[34], die hier aufgrund der Komplexität nicht weiter behandelt werden sollen.

5.2.1 XML und Java

Für die Verarbeitung von XML-Dokumenten in Java sind bereits diverse Schnittstellen definiert.[35] Eclipse verwendet zur Konfiguration der einzelnen Module ebenfalls XML, weshalb für das entwickelte Framework auch alle Konfigurationen in XML vorgenommen werden. Durch die Erweiterbarkeit und Definition eigener Sprachen ist XML hier sehr gut geeignet, komplexe Konfigurationsmöglichkeiten und –Variationen definieren zu können. Diese XML-Konfigurationen sollen somit für das Framework eine Schnittstelle zwischen dem Entwickler und der zu erstellenden Client-Applikation darstellen.

5.3 Das Standard Widget Toolkit

Das *Standard Widget Toolkit*, oder kurz „SWT“ genannt, ist ein Werkzeug zum Entwickeln von grafischen Benutzeroberflächen und wurde 2001 von IBM für Eclipse entwickelt. Dabei besteht das SWT aus einer Vielzahl von Java-Paketen, die entsprechende Funktionalitäten zur Verfügung stellen.[36]

Java bietet mit *Swing* und dem *Abstract Windows Toolkit* (AWT) bereits zwei Komponenten zur Erstellung von grafischen Benutzeroberflächen, jedoch unterscheiden sich diese von SWT in den meisten Bereichen.

Das SWT ist wie Swing oder AWT eine Programmierschnittstelle (kurz „API“) zum Erstellen von grafischen Benutzeroberflächen, basiert jedoch auf keiner der beiden Komponenten. SWT bietet dem Programmierer direkten Zugriff auf die GUI-Komponenten des Betriebssystems, auf dem die Applikation ausgeführt wird.[37]

---

[34] Beginning XML, 3rd Edition, S.19
[35] vgl.: XML in the Java Platform Standard Edition
[36] vgl.: SWT/JFace in Action, S.2
[37] vgl.: SWT/JFace in Action, S.2f

Jedes Betriebssystem mit grafischer Schnittstelle bietet eine Reihe von GUI-Komponenten zur Darstellung von Inhalten an. Diese sind z.B. Fenster, Menüs, Buttons, Textfelder, Listenelemente oder Tabellen. Das Betriebssystem wiederum stellt Schnittstellen zur Verfügung, um diese Komponenten verwenden zu können. SWT ist somit eine weitere Schnittstelle, die es dem Programmierer ermöglicht, direkten Zugriff auf die GUI-Komponenten des Betriebssystems zu erhalten, ohne sich dabei um die betriebssystemspezifischen Implementierungen kümmern zu müssen. Eine Applikation, die mit SWT erstellt worden ist, lässt somit den Anschein erwecken, sie wäre für das jeweilige Betriebssystem entwickelt worden. Damit erscheinen die Applikationen auf jedem Betriebssystem in dem jeweiligen Design und dem ausgewählten Look & Feel.[38]

Neben der Verwendung von betriebssystemspezifischen Komponenten bietet SWT auch ein entsprechendes Event-Handling. Das bedeutet, dass der Programmierer auf Ereignisse von Komponenten reagieren kann. Wenn beispielsweise ein Button gedrückt oder die Maus über einen Bereich bewegt wird, so kann der Programmierer auf diese Ereignisse adäquat reagieren und den entsprechenden Programmcode ausführen.[38] Das Event-Handling ist ein wesentlicher Bestandteil, da ohne diesbezügliche Kontrolle oder Steuerung des Verhaltens der GUI-Komponenten eine Applikation unbrauchbar wäre.

Für die Erstellung von Grafiken bietet SWT ebenfalls eine Fülle von Werkzeugen. Hier gibt es Funktionen zum Verarbeiten von Bildern, Zeichnungen oder Schriftarten, die sich dann in die Applikationen leicht einbetten lassen.[39]

Grundsätzlich lassen sich mit SWT sehr komplexe und umfangreiche grafische Benutzeroberflächen erstellen, die, wie bereits erwähnt, auf dem jeweiligen Betriebssystem das Design annehmen und damit ein natürliches, natives Erscheinungsbild aufweisen. Das hat den Vorteil, dass sich Benutzer der Anwendung nicht an eine anders aussehende Oberfläche gewöhnen müssen.

[38] vgl.: SWT/JFace in Action, S.3
[39] vgl.: SWT/JFace in Action, S.4

5.3.1 Vergleich mit Swing und AWT

Die Funktionen und Schnittstellen von SWT sind grundsätzlich nicht neu und wurden bereits mit AWT und Swing eingeführt.

Das Abstract Windows Toolkit ist die ursprünglichste Komponente von Java zum Erstellen von grafischen Benutzeroberflächen und war bereits in der ersten Version von Java enthalten. Wie auch SWT, benutzt das AWT dabei die nativen GUI-Komponenten des Betriebssystems und stellt Schnittstellen zu diesen zur Verfügung.[40] Aus diesem Grund wird es auch als „heavyweight GUI-Toolkit“ bezeichnet, weil es für jede Komponente ein eigenes Fenster des Betriebssystems verwendet.[41]

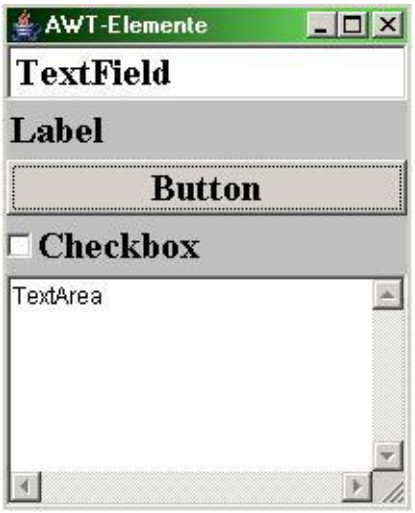

Abbildung 5.2 - AWT Beispiel

Sun Microsystems wollte mit dem AWT auf allen Betriebssystemen dieselben Voraussetzungen schaffen, weshalb nur eine beschränkte Anzahl an GUI-Komponenten standardmäßig enthalten ist. So werden beispielsweise Tabellen („Tables“), Baum-Listen („Trees“) oder Fortschrittsbalken („Progress Bars“) nicht unterstützt, was die Flexibilität von AWT deutlich einschränkt.[42]

[40] vgl.: SWT, Swing or AWT: Which is right for you?
[41] Java Programming on Linux, S.523
[42] SWT, Swing or AWT: Which is right for you?

Um diesen Problemen entgegenzuwirken wurde das Swing GUI-Toolkit 1998 von Sun veröffentlicht und entwickelte sich schnell zum beliebtesten Toolkit zum Erstellen von grafischen Benutzeroberflächen.[43] Swing ist im Gegensatz zu AWT und SWT ein „lightweight GUI-Toolkit“, da es sich nicht der GUI-Komponenten des jeweiligen Betriebssystems bedient, sondern das grafische Rendering selbst übernimmt.[41] Dabei werden die Komponenten immer auf die gleiche Art und Weise erstellt und verhalten sich auf jedem Betriebssystem, das Java unterstützt, gleich. Der Nachteil dabei ist jedoch, dass Swing-Applikationen in der Regel langsamer sind, da alle Komponenten von der Java Virtual Machine verarbeitet werden müssen. Desweiteren ist das Aussehen der Applikation für viele Entwickler ein wichtiger Faktor. Swing-Applikationen basieren immer auf einem Look & Feel, das spezifiziert werden muss. Daher ist das Aussehen mit demselben Look & Feel auf jedem Betriebssystem gleich. Das hat den Nachteil, dass der Benutzer nicht mit der ihm von seinem Betriebssystem bekannten Oberfläche arbeiten kann und sich so an das Swing-Design gewöhnen muss.[44]

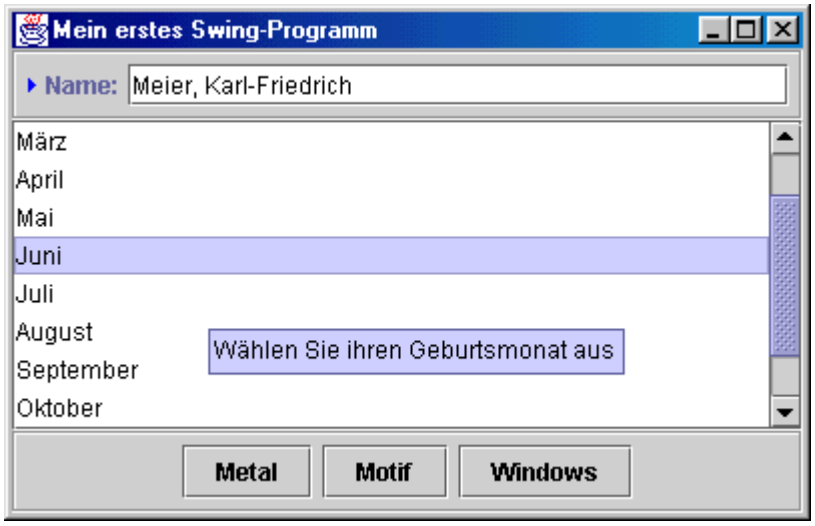

Abbildung 5.3 - Swing Applikation im "Metal"-Look & Feel

[43] SWT/JFace in Action, S.4f
[44] SWT/JFace in Action, S.5

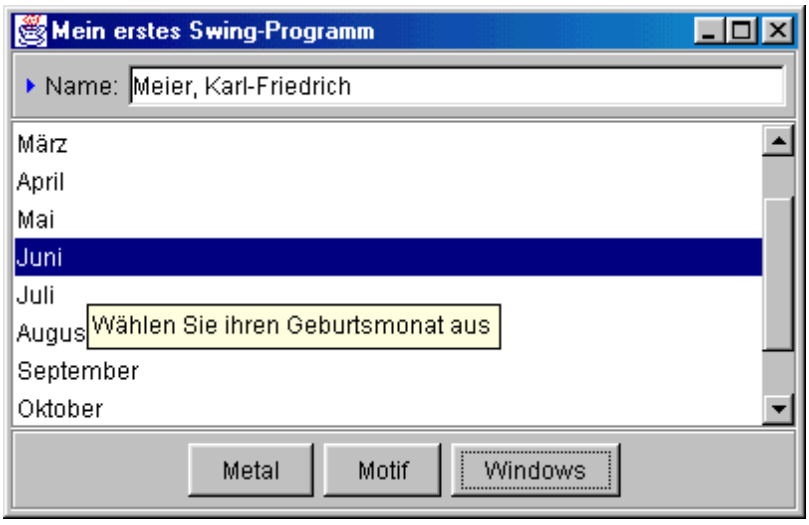

Abbildung 5.4 - Swing Applikation im "Windows"-Look & Feel

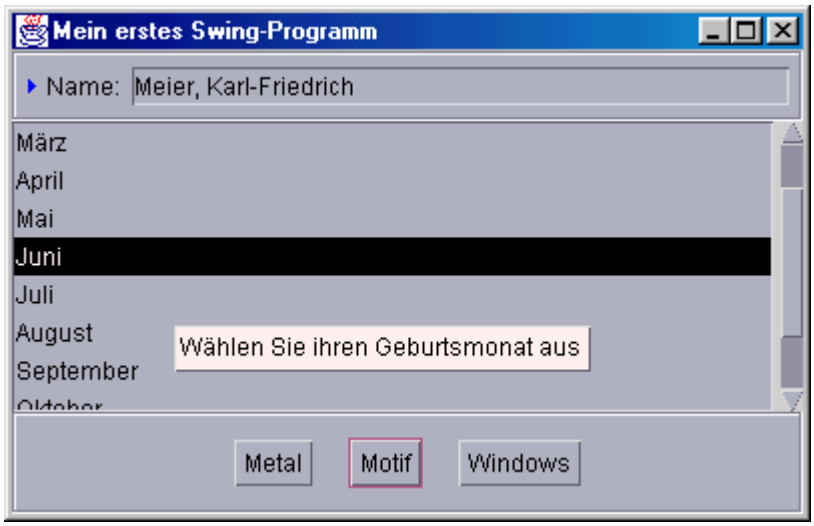

Abbildung 5.5 - Swing Applikation im "Motif"-Look & Feel

Der Vorteil von Swing liegt in der hohen Flexibilität und der Möglichkeit, grundsätzlich jede Form von grafischer Oberfläche zu realisieren, da eben keine Betriebssystem-spezifischen Komponenten verwendet werden. Zudem ist das Swing-Toolkit sehr weit entwickelt, stabil und einfach in der Handhabung.[45]

[45] SWT/JFace in Action, S.6

SWT basiert von der Technologie am ehesten auf dem AWT, verwendet aber keine der Implementierungen. Im Folgenden soll die Technologie von SWT näher betrachtet werden.

5.3.2 Technologie

Wie bereits erläutert, bietet das SWT eine Schnittstelle zu den heavyweight-GUI-Komponenten des Betriebssystems. Dabei wird das „Java Native Interface“ (JNI) verwendet, welches es ermöglicht, Programmcode und Bibliotheken zu verwenden, die nicht in Java geschrieben worden sind.[46] Mit JNI wird die Interaktion mit dem Betriebssystem arrangiert, was das Verwenden von dessen GUI-Komponenten erst ermöglicht.

Oberhalb von JNI ist bei allen SWT-Applikationen die Klasse *Display* von Bedeutung. Diese ist für die Übersetzung von allen Aufrufen innerhalb der Applikation, in die für das Betriebssystem verständliche Form, zuständig. Auf einem Windows Betriebssystem beispielsweise werden alle Aufrufe der Applikation über JNI in C-Funktionen übersetzt und ausgeführt. Die Rückgabeparameter werden anschließend vice versa in Java-lesbare Form umgewandelt. Dasselbe gilt für alle anderen Betriebssysteme, die über JNI angesprochen werden können. Es ist somit auch möglich, selbstprogrammierte Betriebssysteme über SWT anzubinden, vorausgesetzt die verwendeten Methoden werden in die für das Betriebssystem verständliche Form übersetzt.[47]

Neben der Display-Klasse spielt die Klasse *Shell* eine bedeutende Rolle. Diese besitzt eine Implementierung für die Darstellung und repräsentiert das primäre Applikations-Fenster.[48] Alle weiteren GUI-Elemente können innerhalb einer Shell angeordnet werden, welche damit die Kontrolle über diese Elemente übernimmt. Solche Elemente können auch weitere Shell-Komponenten sein, die innerhalb der sogenannten „Parent-Shell“

[46] Java Native Interface
[47] vgl.: SWT/JFace in Action, S.16f
[48] SWT/JFace in Action, S.18

operieren. In einer Shell lassen sich dann alle GUI-Elemente wie beispielsweise Buttons, Labels, Tables, Trees oder Progress Bars anordnen.[49]

Die Darstellung der Komponenten ist, wie bereits erläutert, vom Betriebssystem abhängig und unterscheidet sich somit auf jeder Plattform. Die folgenden Abbildungen sollen diesen Umstand verdeutlichen:

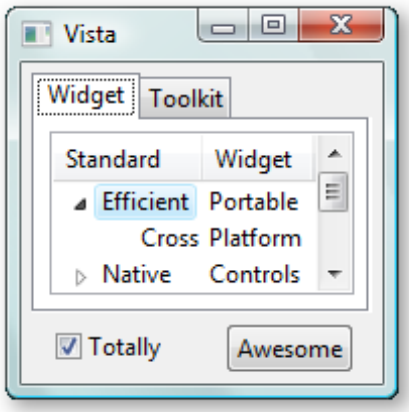

Abbildung 5.6 - SWT unter Windows Vista

Abbildung 5.7 - SWT unter Windows XP

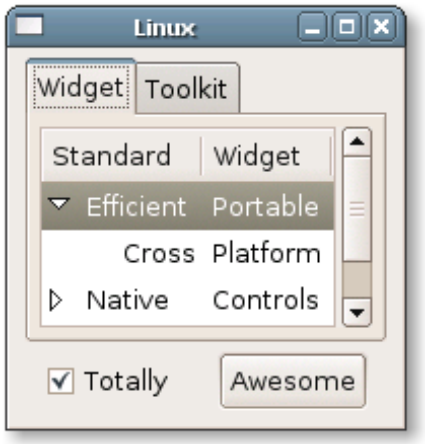

Abbildung 5.8 - SWT unter Linux

Abbildung 5.9 - SWT unter Mac OS X

[49] vgl.: SWT/JFace in Action, S.19

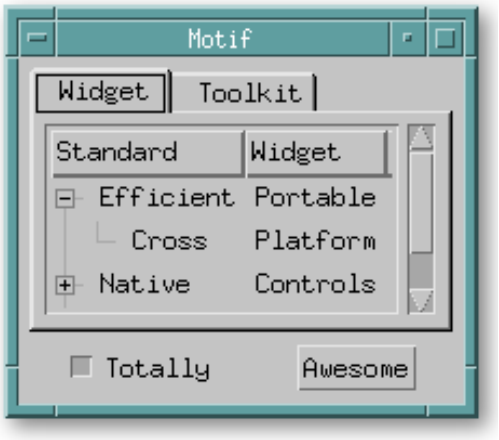

Abbildung 5.10 - SWT unter Motif

Abbildung 5.11 - SWT unter Photon

Die in den Abbildungen dargestellten GUI-Elemente sind in der Shell eingebettet und dienen der Interaktion mit der Applikation. Solche GUI-Elemente werden in SWT auch *Widgets* genannt, wie im nächsten Kapitel ausführlicher beschrieben wird. Diese Widgets und alle damit verbundenen Eigenschaften und Funktionen, werden in der Shell verwaltet. Die folgende Grafik soll die Interaktion von Betriebssystem, Display, Shell und Widgets verdeutlichen.[50]

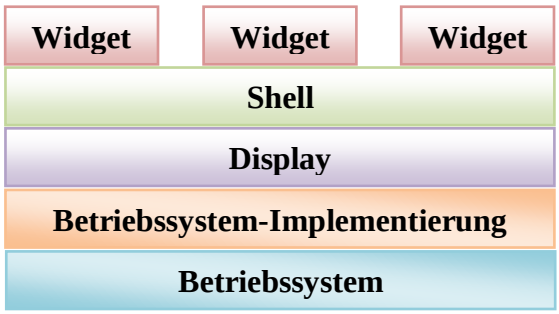

Abbildung 5.12 - SWT Struktur

[50] vgl.: SWT/JFace in Action, S.19

5.3.3 GUI-Elemente (Widgets)

Wie bereits angesprochen werden GUI-Elemente in SWT, die eine Interaktion mit der Applikation ermöglichen, auch *Widgets* genannt. Dabei gibt es normale Widget-Klassen und sogenannte „Control"-Klassen, die zusätzliche Konfigurationsoptionen wie beispielsweise das Setzen von Größe und Position erlauben.[51]

Die folgende Grafik zeigt die Hierarchie der Widget-Klassen.

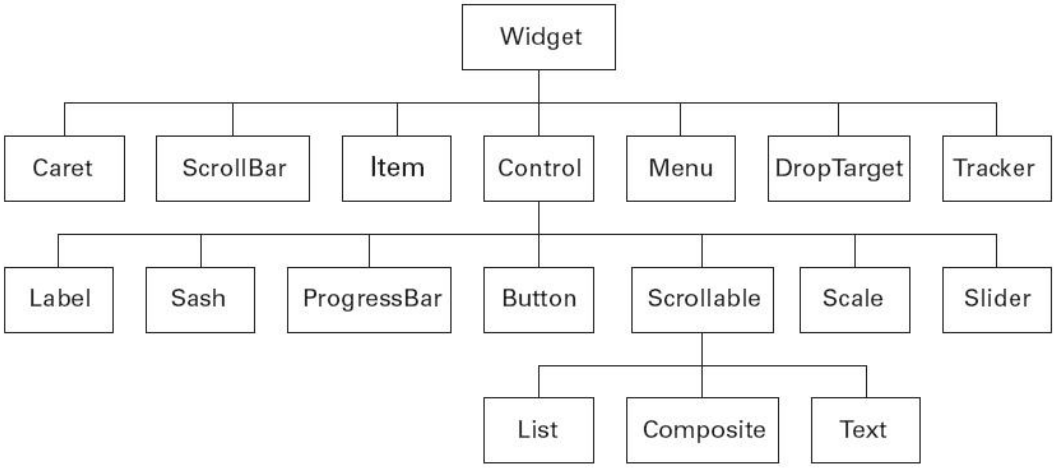

Abbildung 5.13 - Widget Klassenhierarchie

Die meisten dieser Klassen sind in ähnlicher Form auch in Swing enthalten und bieten eine ähnliche Funktionalität. Da die grafischen Benutzeroberflächen der meisten Betriebssysteme ähnlich sind (Buttons, Labels, Textfelder, Listen, etc.), können die Eigenschaften und Funktionsweisen auch als entsprechend ähnlich angenommen werden. So wird der Benutzer mit der Funktion eines Buttons oder eines Textfeldes in der Regel vertraut sein, und damit entsprechend umgehen können. Der Entwickler muss sich somit weniger Gedanken um die betriebssystemspezifischen Eigenschaften der GUI-Elemente machen.

Wie in Abbildung 5.13 dargestellt, bietet SWT eine große Anzahl an GUI-Elementen, die der Entwickler in seinen Applikationen verwenden und konfigurieren kann.

[51] vgl.: SWT/JFace in Action, S.28ff

Die folgenden Abbildungen zeigen einige dieser GUI-Elemente auf verschiedenen Betriebssystemen.

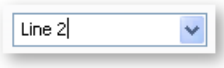

Abbildung 5.14 - Listen

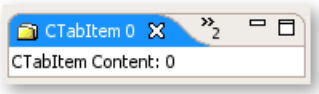

Abbildung 5.15 - CTabFolder

Abbildung 5.16 - Calendar

Abbildung 5.17 - ExpandBar

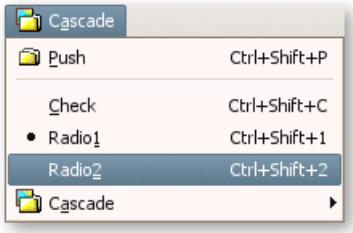

Abbildung 5.18 - Menu

Abbildung 5.19 - StyledText

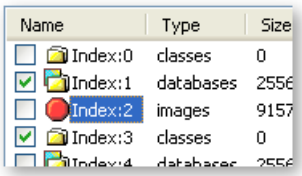

Abbildung 5.20 - Table

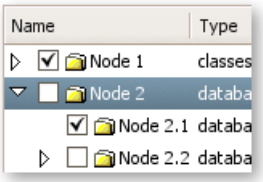

Abbildung 5.21 - Tree

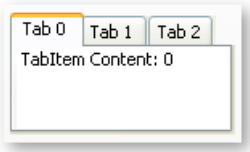

Abbildung 5.22 - TabFolder

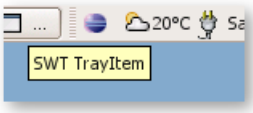

Abbildung 5.23 - Tray

Abbildung 5.24 - Button

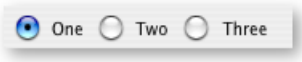

Abbildung 5.25 - ButtonGroup

Neben den eben dargestellten gibt es noch weitere nützliche GUI-Elemente, die der Entwickler in seinen Applikationen einsetzen kann. So gibt es zum Beispiel eine Reihe von Widgets, welche die Darstellung von 2D oder 3D Grafiken erlauben. Das *Simple Graph Widget* ermöglicht beispielsweise die grafische Darstellung von Graphen mit diversen Konfigurationsmöglichkeiten.[52]

Für die Entwicklung von Applikationen unter Eclipse ist sowohl das Standard Widget Toolkit, als auch *JFace* relevant, das im Folgenden näher erläutert werden soll.

5.4 JFace

Das Eclipse Team beschreibt *JFace* als „Ein UI Toolkit, das Klassen für das Entwickeln von UI Features bereitstellt, die mühsam zu implementieren sein können“[53]. Dabei wurde JFace als GUI-Toolkit entwickelt, um eine Abstraktionsebene über SWT zu bieten. JFace beinhaltet eine Reihe von Klassen, die wiederum SWT-Code kapseln und dem Entwickler einen einfacheren Zugang zu den GUI-Elementen bieten. Es gibt zudem für viele Standardaufgaben wie Schriftarten- oder Bildverwaltung bereits Implementierungen, die mit relativ geringem Aufwand verwendet werden können.[54]

Dabei gibt es verschieden Arten von JFace-Komponenten, die sich in vier Gruppen einteilen lassen:[55]

- *Viewers*
- *Actions* und *Contributions*
- Schriftarten- und Bildverwaltungen
- *Dialogs* und *Wizards*

Viewers repräsentieren eine Art der Informationsaufbereitung und –Darstellung. Es können Daten von Elementen in unterschiedlicher Art und Weise aufbereitet und

[52] vgl.: Advanced Widgets for Eclipse
[53] Advanced Widgets for Eclipse
[54] vgl.: SWT/JFace in Action, S.3f
[55] SWT/JFace in Action, S.20

dargestellt werden.[56] Häufig verwendete Viewer sind *TreeViewer* oder *TableViewer*, die Informationen in einer Baumstruktur oder tabellarisch anzeigen können.

Actions und *Contributions* stellen eine Abstraktionsebene über Events, also Ereignisse in der Benutzeroberfläche, dar. Wenn beispielsweise dieselben Events öfter auftreten, so kann eine Action erstellt werden, die von den Event-auslösenden Komponenten ausgeführt wird.[57]

Schriftarten- und Bildverwaltungen dienen der effizienten Verteilung von Systemressourcen. Schriften und Bilder können eine große Menge an Systemressourcen verbrauchen, wenn sie nicht korrekt verwaltet werden. Mit JFace muss sich der Entwickler weniger um die korrekte Verteilung und Verwaltung von Systemressourcen kümmern, sofern er die verfügbaren Verwaltungsmechanismen einsetzt.[58]

Dialogs und *Wizards* sind Komponenten zur Interaktion mit der Applikation. Die Dialoge bedienen sich dabei der SWT-Fenster und erweitern diese um zusätzliche Funktionen wie detailliertere Fehlerdarstellung oder erweiterte Informations-Darstellungen. Wizards sind erweiterte Dialoge, die den Benutzer durch einen Prozess führen, wie beispielsweise das Installieren einer Software.[59]

Wie bereits eingehend erwähnt, wurde JFace für die Eclipse Plattform entwickelt, die sich einer Vielzahl der von JFace zur Verfügung gestellten Elemente bedient. Für die Entwicklung von Client-Applikation unter Eclipse ist JFace auch für den Programmierer von großer Bedeutung. Das nächste Kapitel beschäftigt sich daher umfassend mit den Grundlagen sowie der Client-Programmierung unter Eclipse.

[56] vgl.: SWT/JFace in Action, S.20
[57] vgl.: SWT/JFace in Action, S.21
[58] vgl.: SWT/JFace in Action, S.21
[59] vgl.: SWT/JFace in Action, S.21

5.5 Eclipse

Die Definition der Eclipse Foundation für Eclipse ist sehr offen formuliert und lautet: „Eclipse ist eine Open Source Community, deren Projekte sich auf die Bereitstellung einer erweiterbaren Entwicklungsplattform und die Erstellung von Applikations-Frameworks zur Softwareentwicklung konzentrieren.“[60] Dabei steht Eclipse nicht als Anwendung oder Entwicklungsumgebung im Vordergrund, sondern als Zusammenschluss mehrerer Softwareunternehmen zu einer Community.[61]

Die Eclipse Foundation ist eine Non-Profit Organisation, die sich zum Ziel gesetzt hat, die Entwicklungen für die Eclipse Plattform und die Community zu koordinieren. Sie stellt für Eclipse diverse Projekte bereit und bietet den Entwicklern rund um Eclipse Dienste zur organisierten Zusammenarbeit an.[62]

5.5.1 Geschichte

Im November 1998 begann IBM mit der Entwicklung einer Plattform für Java-Entwicklungswerkzeuge. Zunächst waren die potentiellen Investoren zurückhaltend bei der finanziellen Unterstützung des Projekts, worauf im November 2001 die Software unter eine Open-Source Lizenz gestellt wurde, um die Verbreitung zu beschleunigen und weitere Entwicklungen auch außerhalb von IBM zu ermöglichen. IBM sowie die Software-Unternehmen Borland, MERANT, QNX Software Systems, Rational Software, Red Hat, SuSE, TogetherSoft und Webgain [63] gründeten das „Eclipse Konsortium“ sowie die Plattform „eclipse.org“. Die Mitglieder sollten Eclipse intern verwenden, es wirtschaftlich vorantreiben sowie jeweils ein Produkt auf Basis von Eclipse auf den Markt bringen. Die Akzeptanz war jedoch zunächst eher gering, da vielen das Projekt als von IBM zu sehr kontrolliert erschien. Aus diesem Grund wurde im Februar 2004 die Eclipse Foundation gegründet, die von nun an die Aufsicht über das Open-Source-Projekt haben sollte. Eclipse hat seitdem diverse strategische Mitglieder hinzubekommen, die zum einen Entwicklungsarbeit leisten und zum anderen das

[60] Eclipse – Die Plattform, S.13
[61] vgl.: Eclipse – Die Plattform, S.13
[62] vgl.: About the Eclipse Foundation
[63] About the Eclipse Foundation

wirtschaftliche Wachstum und die Verbreitung von Eclipse fördern.[64] Aufgrund dieser Unterstützung ist Eclipse heute eine der führenden Java Entwicklungsumgebungen und besitzt Marktanteile in Europa, dem Mittleren Osten, Afrika, Asien und Nordamerika.[65]

5.5.2 Dienste der Eclipse Foundation

Wie bereits erwähnt, ist die Eclipse Foundation für die Koordination der Community sowie der einzelnen Projekte zuständig. Dabei bietet sie diverse Dienste an, die sich wie folgt darstellen lassen und im Folgenden näher erläutert werden:[66]

- Schaffung von IT-Infrastruktur
- Management von geistigem Eigentum
- Koordination des Entwicklungsprozesses
- Entwicklung eines Ökosystems

5.5.2.1 Schaffung von IT-Infrastruktur

Die Eclipse Foundation stellt den Entwicklern Werkzeuge zur leichteren Zusammenarbeit, wie Code-Management-Systeme (CVS, SVN), Bug-Report-Datenbanken (Bugzilla), Mailing-Listen, Newsgroups sowie diverse Webservices zur Verfügung.

5.5.2.2 Management von geistigem Eigentum

Hier steht vor allem die Lizenzierung von Eclipse-Software im Vordergrund. Jede Software, die nicht kommerziell für die Eclipse Plattform verfasst worden ist, steht unter der Eclipse Public License (EPL). Diese Lizenz erlaubt es, auch kommerzielle Software auf Basis von Eclipse zu entwickeln und zu vertreiben. Alle Entwicklungen für die Eclipse Plattform durchlaufen einen Prozess der Überprüfung auf Lizenz-Kollisionen und Code-Herkunft.

---

[64] vgl.: A brief history of Eclipse
[65] vgl.: Eclipse Becomes the Dominant Java IDE
[66] About the Eclipse Foundation

5.5.2.3 Koordination des Entwicklungsprozesses

Die Mitglieder der Eclipse Foundation unterstützen die beteiligten Entwickler bei der Ein- und Durchführung des Eclipse-Entwicklungsprozesses. Dabei werden beispielsweise Community-Reviews und andere Veranstaltungen abgehalten, um die Zusammenarbeit der Projekt-Teams zu verbessern.

5.5.2.4 Entwicklung eines Ökosystems

Die Eclipse Foundation ist auch für die Koordination von Marketing und die Schaffung eines erweiterten Ökosystems für die Eclipse-Projekte zuständig. Dazu zählen unter anderem die Verbreitung von Eclipse-basierter Software, das Anbieten von Schulungen und Support oder das Herausgeben von Magazinen und Literatur. Desweiteren werden jährliche Community-Konferenzen wie die „EclipseCon“ oder die „Eclipse Summit Europe“ abgehalten, um die Community zu stärken und weiter voranzutreiben.

5.5.3 Die Eclipse Plattform

Neben der Eclipse-Community und Eclipse Foundation ist Eclipse eine Plattform zum Entwickeln von Software. Dabei gibt es unterschiedliche Pakete für diverse Anforderungen, wie Plug-In-, Java-, C/C++- oder Web-Entwicklung. Diese Pakete beinhalten initial unterschiedliche Eclipse-Projekte und sind somit auf bestimmte Anforderungen zugeschnitten.[67]

Die Eclipse Plattform ist dabei die unterste Ebene für die Erstellung von Software und bietet dafür notwendige Funktionalitäten. Auf ihr lassen sich Entwicklungsumgebungen programmieren, wie beispielsweise das *Eclipse Software Development Kit*, das sich wiederum aus mehreren Projekten zusammensetzt. Es können aber auch Anwendungen programmiert werden, die nichts mit Softwareentwicklung zu tun haben. Für solche Anwendungen kann die *Rich Client Platform* (*RCP*, siehe 5.5.5) verwendet werden, die einen Teil der Eclipse Plattform darstellt.[68] Eclipse wird deshalb auch häufig als „Universelle Plattform“[69] bezeichnet. Eclipse ist die Basis für alle Komponenten, die darauf aufsetzen. Dabei verwendet sie einen Mechanismus zum Verwalten von

[67] vgl.: Eclipse Download Overview
[68] vgl.: Eclipse Plattform Technical Overview
[69] Eclipse: A Platform Becomes an Open-Source Woodstock

Modulen, die auch *Plug-Ins* genannt werden. Die Architektur setzt auf dem OSGi-Framework auf, das die Modularisierung spezifiziert und erleichtert.[70] Ein Plug-In stellt Funktionen zur Verfügung, die, wenn das Plug-In geladen ist, innerhalb der Eclipse-Oberfläche verwendet werden können. Ein Plug-In wird in Java programmiert und üblicherweise in Form von Java-Archiv-Dateien (*JAR*), zusammen mit zusätzlichen Steuer- und Konfigurationsdateien, Bildern oder sonstigen erforderlichen Inhalten, bereitgestellt. Jedes Plug-In beinhaltet ein Manifest, in dem die Abhängigkeiten zu anderen Plug-Ins beschrieben werden. Plug-Ins können auch sogenannte „Extension Points“ zur Verfügung stellen, an die andere Plug-Ins anknüpfen können. Vice versa kann ein Plug-In über die Extension Points eines anderen Plug-Ins an dieses anknüpfen. Die Informationen zu den Extension Points werden in einer XML-basierten Datei („plugin.xml“) definiert und von Eclipse beim Start ausgelesen. Dabei werden die Abhängigkeiten und Beziehungen verwaltet und die Plug-Ins je nach Bedarf aktiviert oder deaktiviert.[71]

Grundsätzlich haben alle Funktionen, auch das Laden und Verwalten von Plug-Ins eine entsprechende Implementierung. So werden beim Start des Eclipse Software Development Kits beispielsweise folgende Plug-Ins geladen:[72]

- *Runtime-Core (Plug-Ins: org.eclipse.core.runtime.*, org.eclipse.osgi.*):* Komponente zum Laden und Verwalten von Plug-Ins
- *Workspace-Core (Plug-Ins: org.eclipse.core.resources):* Komponente für das Verwalten von Projekten sowie den Zugriff auf das Dateisystem
- *SWT (Plug-Ins: org.eclipse.swt.*)*
- *JFace (Plug-Ins: org.eclipse.jface*
- *Plug-In Development Environment (PDE, Plug-Ins: org.eclipse.pde.*):* Komponente zum Erstellen von Plug-Ins
- *Workbench (Plug-Ins: org.eclipse.ui.*):* Komponente für die grafische Benutzeroberfläche von Eclipse

[70] Eclipse Plattform Technical Overview
[71] Eclipse Plattform Technical Overview
[72] Eclipse – Die Plattform, S.411

Dies ist nur ein kleiner Teil der Plug-Ins, die bei einer Standard-Installation von Eclipse enthalten sind, stellen aber die Kernkomponenten dar.

Die folgende Abbildung zeigt die Eclipse Plattform Architektur sowie deren Erweiterungsmöglichkeiten.

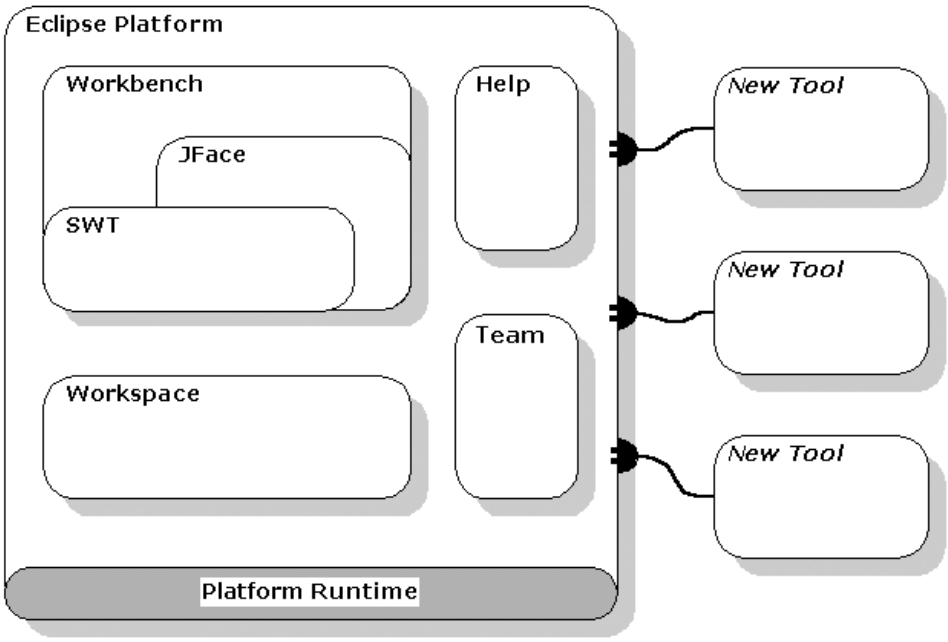

Abbildung 5.26 - Eclipse Plattform Architektur

Für die Aktualisierung der Software ist der Update-Manager, der in die Eclipse-Plattform integriert ist, zuständig. Über den Update-Manager können neue Plug-Ins oder Gruppen von zusammengehörigen Plug-Ins, sogenannten „Features“, installiert oder vorhandene upgedatet werden.

5.5.4 Das Eclipse User Interface (Workbench)

Wie bereits beschrieben, wird auch die Benutzerschnittstelle über ein Plug-In (org.eclipse.ui.*) zur Verfügung gestellt. Dieses Plug-In wird auch *Workbench* genannt und ermöglicht die grafische Interaktion des Benutzers mit den verfügbaren Plug-Ins. Der Workbench setzt auf dem Standard Widget Toolkit (siehe 5.3) und JFace (siehe 5.4) auf und bietet diverse Möglichkeiten der Interaktion mit der Oberfläche. Dabei besteht

der Workbench aus *Views*, *Editoren* und *Perspektiven*, die im weiteren Verlauf näher behandelt werden.[73]

Der Workbench präsentiert sich dem Benutzer sehr modular und ist weitreichend konfigurierbar. Die folgende Abbildung zeigt die Java-Ansicht des Workbenchs:

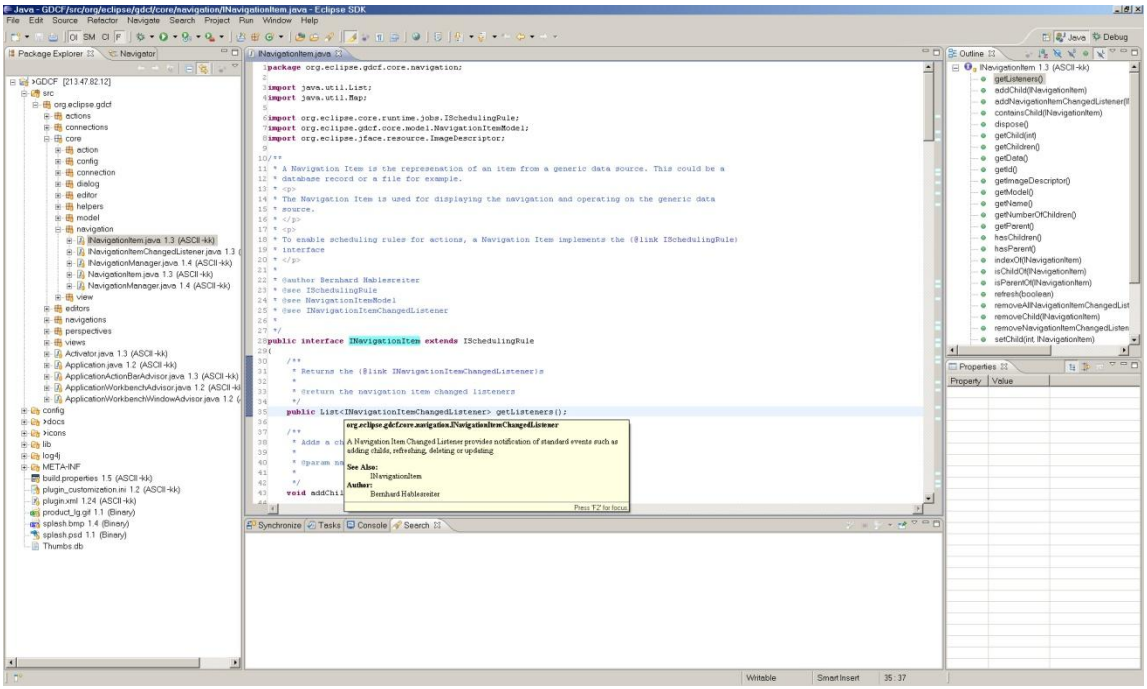

Abbildung 5.27 - Der Workbench

Auf der linken Seite in Abbildung 5.27 ist die Navigation zu sehen, in der Mitte eine geöffnete Java-Datei, rechts und unten jeweils diverse Informations-Fenster. Dabei sind die Elemente, die um die Mitte angeordnet sind, sogenannte „Views“ (siehe 5.5.4.1). Das Hauptfenster stellt einen Editor dar (siehe 5.5.4.2), mit dessen Hilfe Elemente bearbeitet und gespeichert werden können. Das gesamte Layout der Applikation wird über eine Perspektive (siehe 5.5.4.3) definiert, die weitreichend konfiguriert werden kann. Neben Views, Editoren und Perspektiven gibt es noch eine Shortcut-Leiste, eine Menü- und Tool-Leiste sowie eine Statuszeile.

[73] vgl.: Eclipse Plattform Technical Overview

Die folgende Abbildung soll dies verdeutlichen:

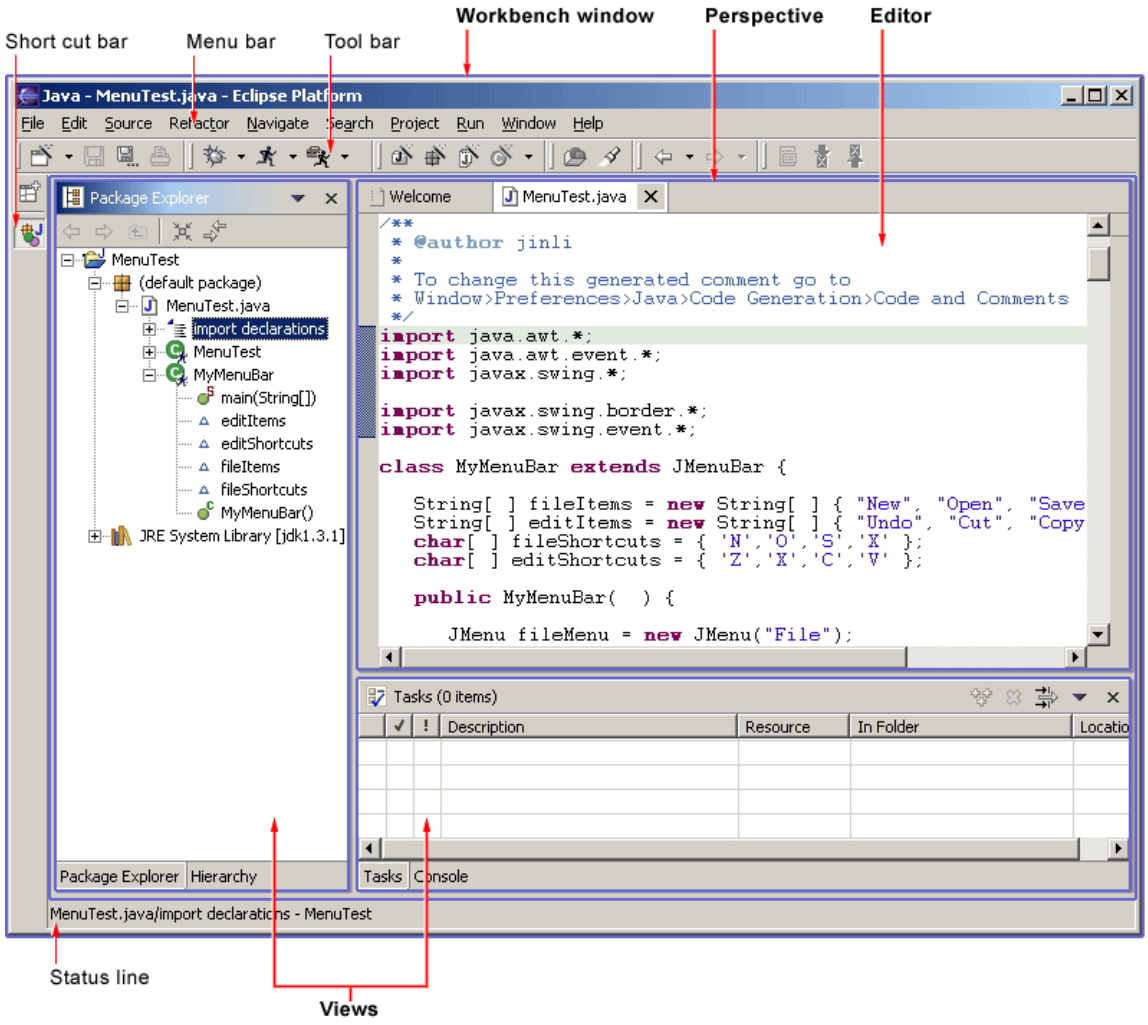

Abbildung 5.28 - Komponenten des Workbenchs

Innerhalb des Workbenchs lassen sich alle Elemente per Drag & Drop verschieben, in der Größe anpassen oder ausblenden. Zudem lassen sich einige Elemente in die Fußleiste minimieren und bei Bedarf wieder in den Vordergrund holen.

Dabei folgt der Workbench den Eclipse-Richtlinien für grafische Benutzeroberflächen, die bereits in Kapitel 3 erläutert worden sind.

Da Views, Editoren und Perspektiven für die Entwicklung von Client-Applikation unter Eclipse von großer Bedeutung sind, werden diese nachfolgend ausführlicher beschrieben.

5.5.4.1 Views

Ein View ist im Workbench eingebettet und bietet eine Sicht auf ein oder mehrere Objekte, die im Workbench verwaltet werden. So können Views beispielweise Informationen zu einer gerade geöffneten Datei enthalten oder die Attribute einer Java-Klasse auflisten, die in einem anderen View ausgewählt ist. Die Eclipse Plattform bietet eine Reihe von Standard-Views, wie den Navigator- Outline, Properties- oder Console-View.

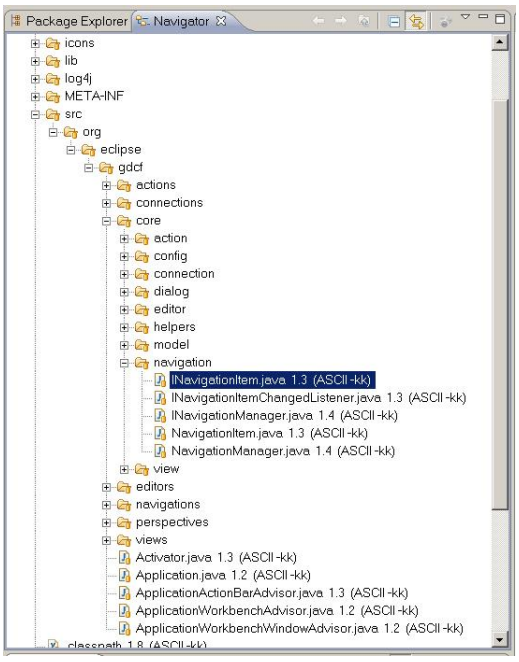

Abbildung 5.29 - Navigator View

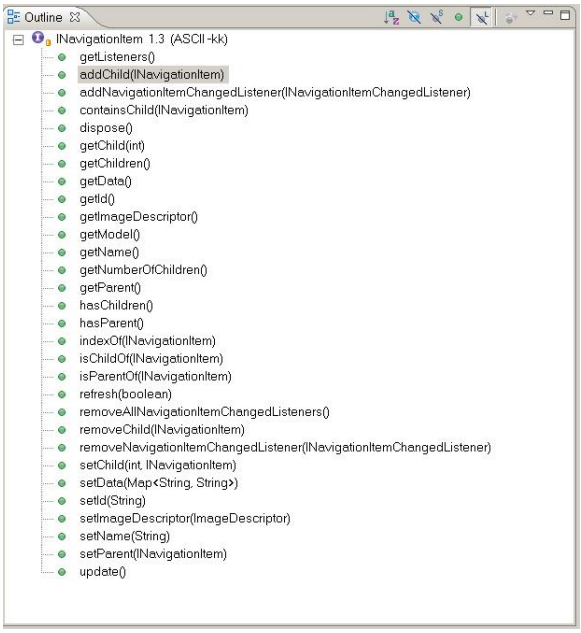

Abbildung 5.30 - Outline View

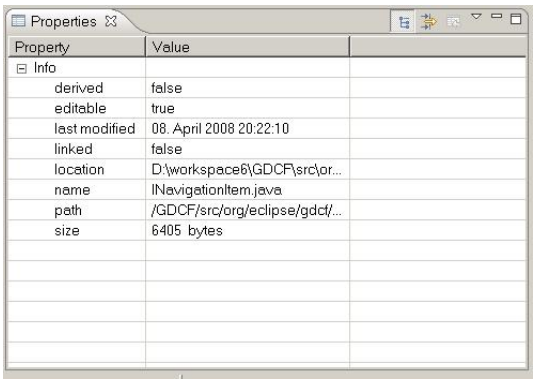

Abbildung 5.31 - Properties View

Dabei können in einem Workbench beliebig viele Views angeordnet, per Drag & Drop verschoben oder ausgeblendet werden. Sie können entweder angrenzend an andere Views oder als Reiter in einem Arbeitsbereich des Workbenchs angeordnet werden.

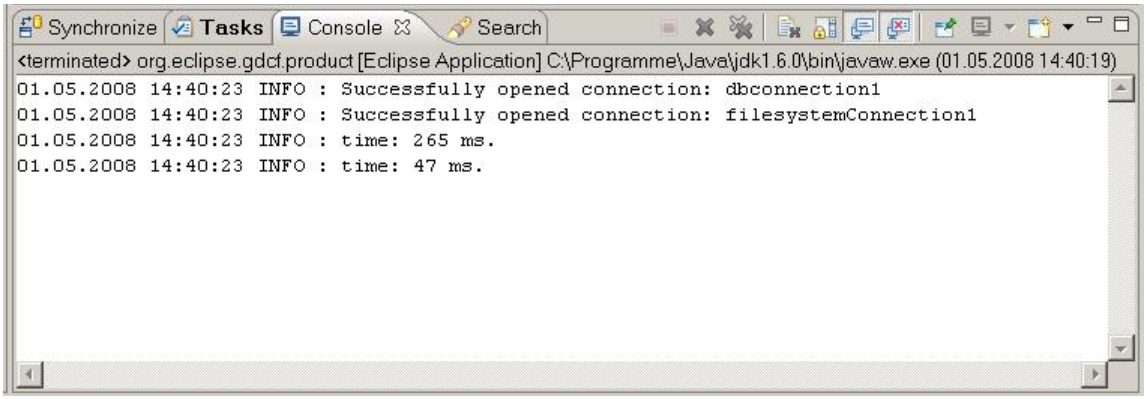

Abbildung 5.32 - Console View

In Abbildung 5.32 ist die Zusammenfassung mehrerer Views in einem Arbeitsbereich des Workbenchs zu sehen. Dabei ist nur der Console-View sichtbar, während die anderen inaktiv sind und bei Bedarf aktiviert werden können. Die Möglichkeiten der Anordnung und Gruppierung von Views machen den Workbench damit äußerst flexibel.

5.5.4.2 Editoren

Editoren erlauben dem Benutzer das Öffnen, Editieren und Speichern von Objekten.[74] Zu bearbeitende Objekte durchlaufen dabei immer denselben Prozess von Öffnen, Speichern und Schließen. Dabei können Editoren und Views miteinander verknüpft werden, wobei in den Views beispielsweise Informationen zu den gerade geöffneten Objekten angezeigt werden können. Die Eclipse Plattform bietet einen Standard-Editor zum Bearbeiten von Text-Dateien, alle weiteren Editoren können über Plug-Ins eingebunden werden.[75]

Die folgenden Abbildungen zeigen Editoren für Java- sowie XML-Dateien, mit Syntax-Highlighting, Darstellung von Klassenhierarchien und viele weitere Funktionen.

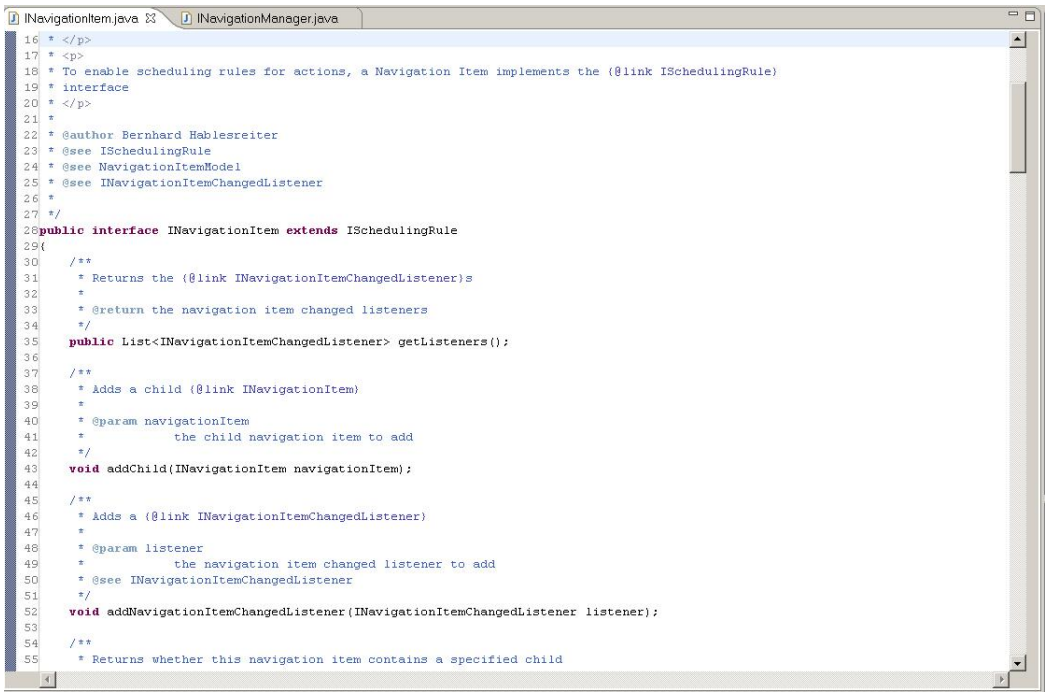

Abbildung 5.33 - Java Editor

[74] Eclipse Plattform Technical Overview
[75] vgl.: Eclipse Plattform Technical Overview

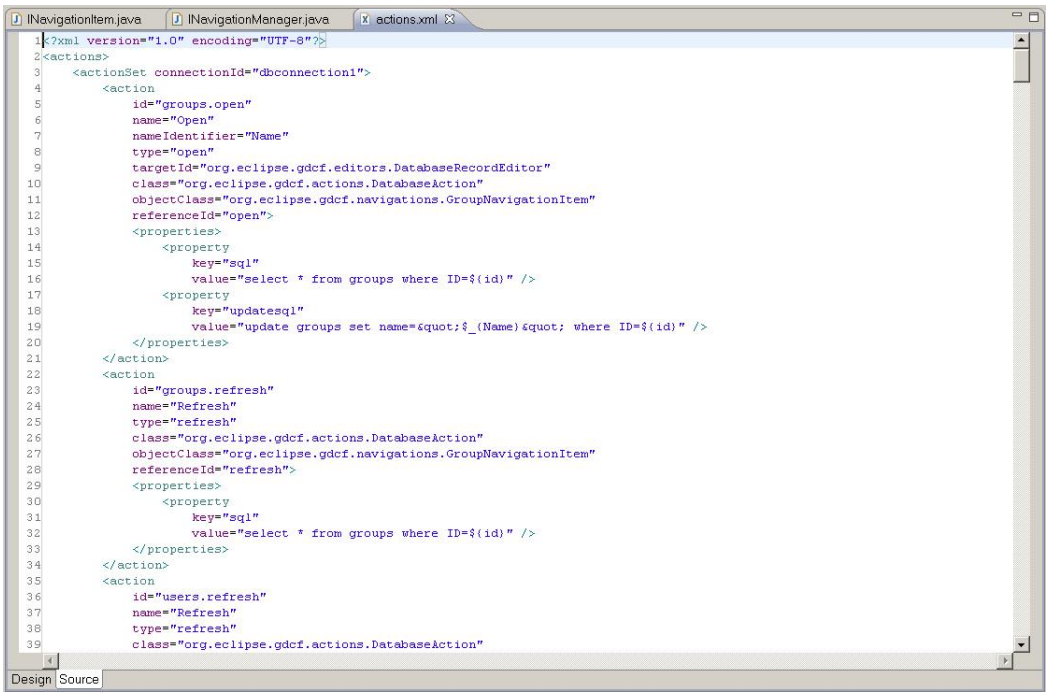

Abbildung 5.34 - XML Editor

5.5.4.3 Perspektiven

Perspektiven stellen eine weitere nützliche Funktion des Workbenchs dar. In Eclipse hängt das Erscheinungsbild des Workbenchs primär von der Perspektive ab, die gerade ausgewählt ist.[76] In Perspektiven können Views, deren Anordnung sowie Menüs und deren Einträge zu Gruppen zusammengefasst werden. Perspektiven werden üblicherweise dann eingesetzt, wenn der Aufgabenbereich thematisch strukturiert werden soll. Ein gutes Beispiel dafür sind die Perspektiven „Java“ und „Debug“ für das Eclipse Software Development Kit. In der Java-Perspektive (Abbildung 5.27) können Java-Dateien bearbeitet werden, wobei diverse Views Informationen zu den Objekten bieten und den Entwickler bei der Arbeit unterstützen. Die Debug-Perspektive (Abbildung 5.35) hingegen erlaubt dem Benutzer das Debuggen seiner Programme, wobei es hier eine Reihe von Views gibt, die den Debug-Prozess unterstützen, wie die Darstellung der gerade ausgeführten Programme oder der Variablen der einzelnen Objekte.

[76] Eclipse – Die Plattform, S.19

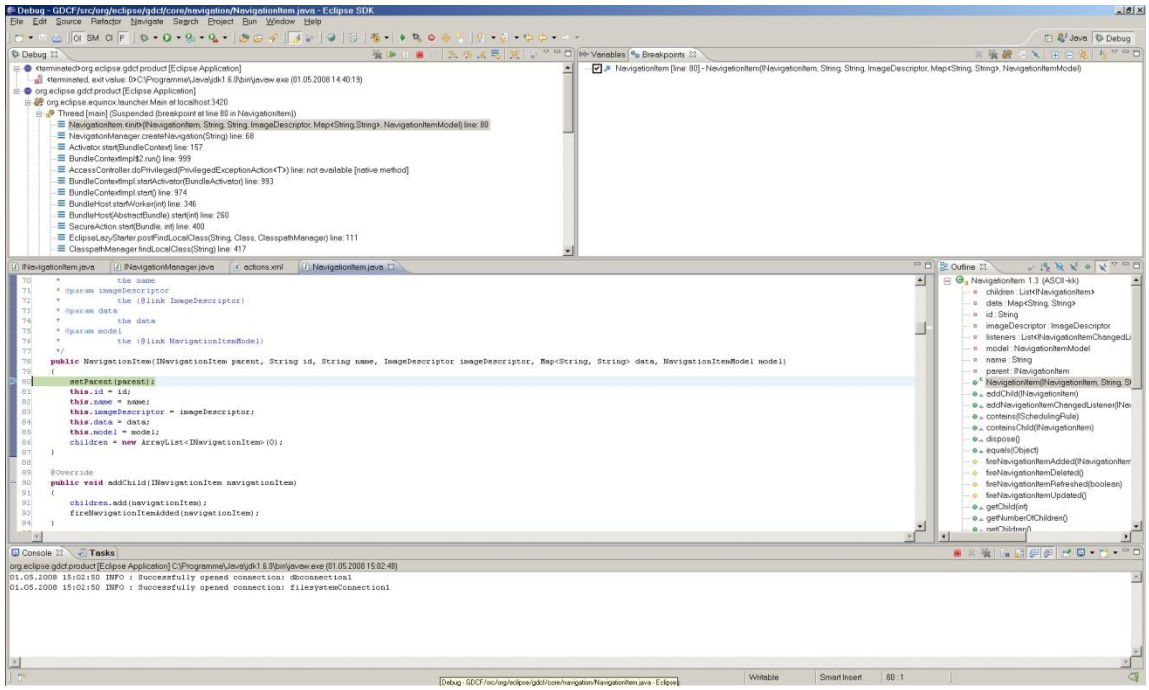

Abbildung 5.35 - Debug-Perspektive

5.5.5 Rich Client Platform

Wie in Kapitel 5.5.3 bereits erwähnt, lassen sich mit Eclipse auch Anwendungen entwickeln, deren Aufgaben unabhängig von Softwareentwicklung sind. Das bedeutet, es können beliebige Anwendungen auf Basis der Eclipse Plattform ausgearbeitet werden. Solche Anwendungen werden *Rich Clients* genannt, weshalb die Entwicklung solcher Anwendungen als *Rich Client Development* bezeichnet wird. Dabei bietet Eclipse die Möglichkeit, die Plug-In-Architektur auch für derartige Anwendungen zu verwenden.[77]

Die Eclipse Rich Client Platform ist seit Version 3.0 für die Entwicklung von Anwendungen abseits der Erstellung von Entwicklungsumgebungen verfügbar. Davor waren bestimmte Entwicklungsumgebungs-spezifischen Komponenten, wie das „Projects"-Menü fester Bestandteil des Workbenchs.[78]

[77] vgl.: Eclipse – Die Plattform, S.636
[78] Eclipse – Die Plattform, S.645

Bei der Entwicklung von Client-Applikationen kann nun das Workbench-Fenster mit allen Funktionen verwendet und modifiziert werden. Es können somit Views, Editoren, Menüs, Werkzeugleisten und Perspektiven erstellt und in die Applikationen integriert werden. Der Workbench stellt damit die Basis der grafischen Applikation dar, in der alle erforderlichen Elemente eingebettet werden.

Die Eclipse Rich Client Platform ist bereits für zahlreiche Open-Source-, aber auch für kommerzielle Softwareprojekte eingesetzt worden. Die folgenden Abbildungen zeigen sowohl eine Open-Source-, als auch eine kommerzielle Anwendung, die auf Basis der Eclipse Rich Client Platform entwickelt worden sind.

Abbildung 5.36 zeigt die Open-Source-Anwendung „Eclipse Trader“, die für den online Aktienhandel entwickelt worden ist. Abbildung 5.37 zeigt den „Actuate BIRT Report Designer“, der das einfache Erstellen von Reports ermöglicht.

Abbildung 5.36 - Eclipse Trader

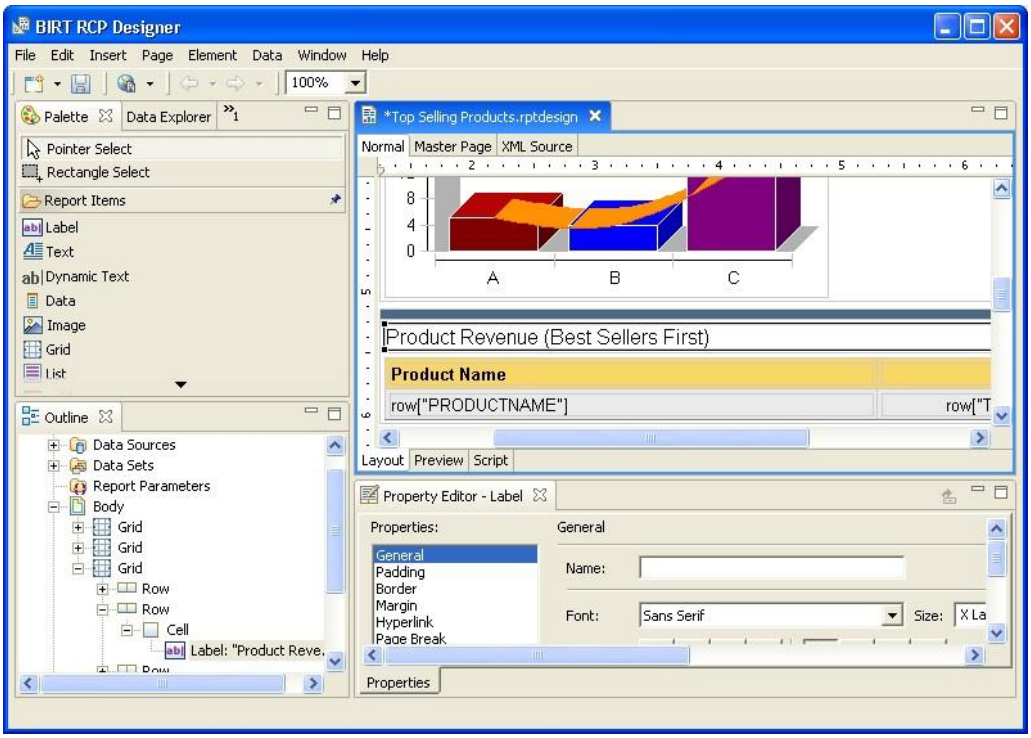

Abbildung 5.37 - Actuate BIRT Report Designer

5.5.6 Relevante Bereiche für das zu entwickelnde Framework

Für das entwickelte Framework sind vor allem die Rich Client Platform und die Workbench-Architektur relevant. Die Rich Client Platform ist ein essentieller Bestandteil, da Applikationen, die mit dem Framework erstellt werden sollen, grundsätzlich viele verschiedene Aufgabenbereiche abdecken können. Wie bereits in Kapitel 5.5.5 erläutert, ist die Rich Client Platform zum Entwickeln von Anwendungen abseits von Entwicklungsumgebungen geeignet, wie es auch mit dem Framework ermöglicht werden soll.

Bevor das entwickelte Framework im Detail erläutert wird, sollen zunächst bereits vorhandene Ansätze diskutiert werden, die Ähnlichkeiten mit den Anforderungen an das Framework aufweisen.

6 Analyse vorhandener Ansätze

Dieses Kapitel beschäftigt sich mit der Analyse bereits vorhandener Ansätze, die Ähnlichkeiten oder Überschneidungen zu dem in Kapitel 3 beschriebenen Framework aufweisen. Desweiteren sollen die Vor- und Nachteile der analysierten Ansätze abgehandelt werden.

Es werden hier zwei Ansätze beschrieben, welche die technischen Grundlagen (siehe 3.8) mit dem Framework teilen, also in Java programmiert worden sind und es ermöglichen, Client-Applikationen auf Basis von Eclipse zu erstellen.

6.1 Das Common Navigator Framework

Das *Common Navigator Framework (CNF)* bietet die Möglichkeit, einen Navigations-View zu erstellen und mit Inhalten zu versehen. Dabei können Menüs, Aktionen, Filter, Sortierfunktionen sowie Drag & Drop-Aktionen eingefügt werden. Das Framework wurde von IBM im Zuge der Entwicklungen des „Web Tools Projects“ (WTP) der Eclipse-Community hinzugefügt und ist seit der Version 3.2 verfügbar. Dabei verwenden bereits einige Komponenten diverser Projekte, wie das „Web Tools Project“ oder das „Data Tools Project“ (DTP) die Funktionalität des Common Navigator Frameworks.[79]

6.1.1 Architektur

Das CNF ist fest in die Eclipse Architektur integriert und erlaubt den Zugriff auf die Funktionen über Eclipse-typische Mittel wie Extension Point-Definitionen sowie XML-Konfigurationen. Dabei steht an oberster Stelle der Navigator - ein View, der Inhalte darstellen und verwalten kann. Die Inhalte sind dabei von Java-Klassen aufzubereiten und in entsprechender Form an den Navigator zu übergeben. Weitere Funktionen wie

[79] Common Navigator Framework for Platform/UI in 3.2

Menüs, Aktionen oder Sortierfunktionen können über Schnittstellen eingebunden werden.[80]

Die folgende Abbildung zeigt den Aufbau des CNF mit den diversen Schnittstellen zu verfügbaren Komponenten.

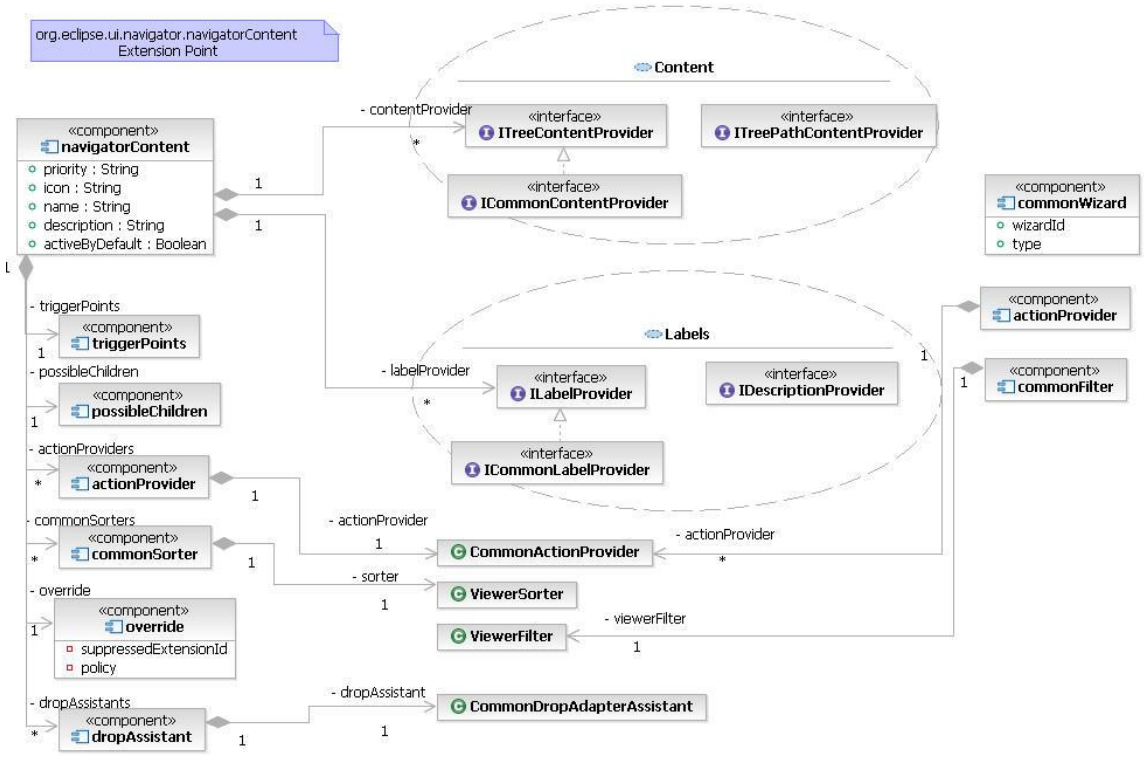

Abbildung 6.1 - Common Navigator Framework

6.1.2 Funktionalität

Das CNF bietet dem Entwickler die Möglichkeit, einen beliebigen Inhalt in einem Navigations-View darzustellen und zu verwalten. Dabei können Inhalte in einen bestehenden View zu anderen, beispielsweise Ressourcen-gebundenen Objekten wie Java-Dateien, hinzugefügt und speziell behandelt werden.[81]

Die folgende Abbildung zeigt die Integration von Ressourcen-ungebundenen Objekten mit Java-Dateien, die Ressourcen-gebunden sind.

[80] vgl.: Common Navigator Framework Presentation, S.10
[81] vgl.: Common Navigator Framework Presentation, S.7ff

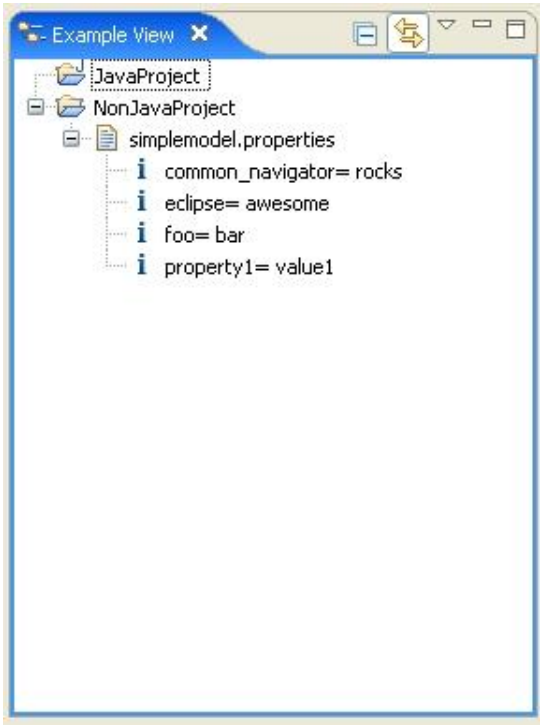

Abbildung 6.2 - Navigator Inhalt

Das CNF war ursprünglich für den Einsatz in der Eclipse-Entwicklungsumgebung vorgesehen, es ist jedoch auch möglich, es für die Rich Client Platform zu verwenden, wie die folgende Abbildung zeigt:

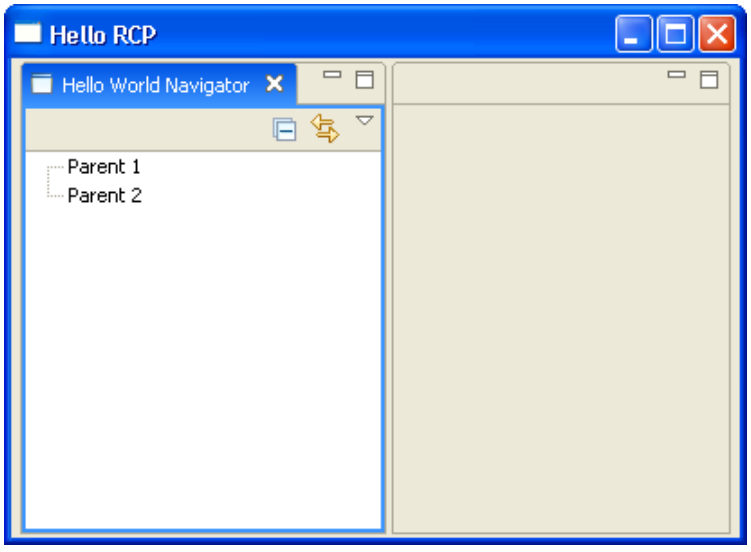

Abbildung 6.3 - Common Navigator Framework für RCP

Neben Inhalten können auch Aktionen definiert werden. Diese werden üblicherweise über Menüeinträge aufgerufen, die ebenfalls konfiguriert werden können. So lassen sich beispielsweise Aktionen zum Löschen oder Anlegen von Objekten definieren, wie die folgende Abbildung verdeutlicht:

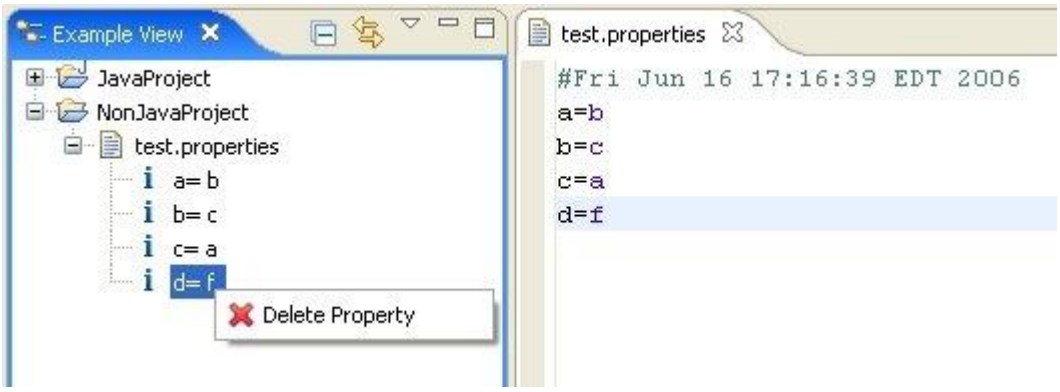

Abbildung 6.4 - Common Navigator Framework – Aktionen

Desweitern können zum Verschieben oder Öffnen von Objekten beispielsweise Drag & Drop-Aktionen sowie Objekt-Filter für einen Navigations-View konfiguriert werden.

6.1.3 Vor- und Nachteile

Der Hauptvorteil des CNF ist die vollständige Integration in die Eclipse-Architektur sowie die Konfiguration des Navigation-Views in der jeweiligen Plug-In-Konfigurationsdatei. Zudem lassen sich Inhalte zu bestehenden Views aus anderen Quellen hinzufügen und unabhängig von diesen verwalten. Die Möglichkeit, Menüs, Aktionen, Filter oder Drag & Drop-Ereignisse zu definieren, gehört ebenfalls zu den Stärken des Frameworks.

Der Nachteil liegt in der sehr aufwändigen und unübersichtlichen Konfiguration der Inhalte und Funktionen. Selbst für simple Applikationen müssen sehr viele Konfigurationen vorgenommen werden, die sich in XML noch erschwerend darstellen können.[82]

[82] vgl.: Common Navigator Tutorial 1

Da das Framework für die Darstellung und Verwaltung von Objekten konzipiert worden ist, gibt es auch keine Möglichkeit der Anbindung von Datenquellen. Das bedeutet, dass der Programmierer die Verwaltung und Verarbeitung der Quelldaten vornehmen muss; Das Framework dient lediglich der Darstellung.

6.2 Jalcedo

Jalcedo ist eine Open-Source Entwicklungsumgebung zum Erstellen von Rich Client Applikationen für Datenbankbasierte Client/Server-Systeme.[83]

Zum Erstellen einer Applikation in Jalcedo sind drei Schritte erforderlich:[84]

1. Erstellung von Java Entity Klassen auf Basis des Java Persistence API[85]
2. Generierung der Client- und Server-Applikation sowie des Codes für die Kommunikation
3. Anforderungsspezifische Anpassung und Überarbeitung des generierten Codes

Jalcedo besitzt eine Reihe von Eclipse Plug-Ins, die in die Applikation integriert werden können.

6.2.1 Architektur

Jalcedo besteht aus drei Komponenten, die jeweils für einen Teilschritt der Erstellung einer Client/Server-Applikation zuständig sind. Die erste Komponente deckt die Erstellung und Verarbeitung von Java Entity Klassen auf Basis des Java Persistence APIs ab. Die zweite Komponente generiert den Source-Code der Client- und Server-Applikation und erstellt das User-Interface. Die dritte Komponente erlaubt die Nachbearbeitung und Anpassung der erstellten Applikationen.

---

[83] Jalcedo Presentation
[84] Rapid developing a Rich Client/Server Application based RCP (JALCEDO)
[85] Java Persistence API

Die folgende Abbildung zeigt die Architektur einer erstellten Client/Server-Applikation:

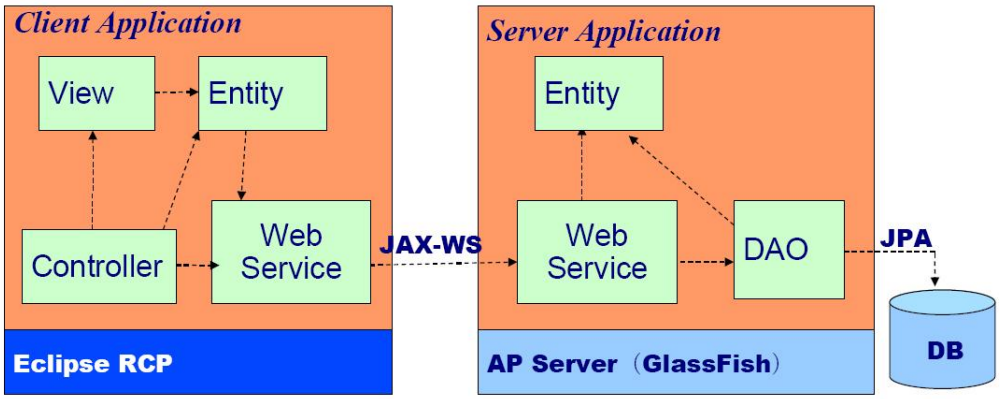

Abbildung 6.5 - Jalcedo Client/Server-Applikation

6.2.2 Funktionalität

Jalcedo bietet Funktionalitäten zum Erstellen der Client/Server-Architektur mit Hilfe des Java Persistence API. Dabei können Datenbanktabellen automatisch generiert und in die Applikation integriert werden. Die Kommunikation erfolgt über eine SOAP-Schnittstelle.[86] Für die Integration mit Eclipse bietet Jalcedo eine Reihe von Editoren und Werkzeugen zum Erstellen der Entity-Klassen sowie der Architektur.[87]

6.2.3 Vor- und Nachteile

Jalcedo ist als Set von Plug-Ins entwickelt worden und lässt sich so einfach in die Eclipse-Umgebung integrieren. In Bereichen, wo Client/Server-Applikationen erforderlich sind, ist Jalcedo einsetzbar. Der Einsatz von Standard-Technologien wie das Java Persistence API oder SOAP machen die Applikationen flexibel und erlauben die Kopplung mit bestehenden Systemen. Das Projekt befindet sich noch in der Entwicklung, weshalb die Funktionen noch nicht gänzlich ausgeschöpft werden können. Desweiteren müssen Navigatoren und Views selbst programmiert und an das Datenmodell angepasst werden. Jalcedo ist für das Zusammenarbeiten mit Datenbanken konzipiert und bietet daher auch keine Möglichkeit der Anbindung anderer Datenquellen.

---

[86] W3C Soap
[87] vgl.: Jalcedo Presentation

7 Das Generic Data Control Framework (GDCF)

Das *Generic Data Control Framework (GDCF)* stellt die im Rahmen dieser Arbeit erstellte Software dar, mit deren Hilfe Client-Applikationen mit Datenquellen-Anbindung unter Eclipse erstellt werden können.

Die Anforderungen an das Framework sind bereits in Kapitel 3 definiert worden und werden im Laufe dieses Kapitels noch weiter betrachtet.

Der Name „Generic Data Control Framework“ zielt auf die generische Architektur sowie die Datensteuerung und –Kontrolle in der Applikation ab. Wie in Abschnitt 3.3 definiert, muss das Framework generisch sein und alle geeigneten Datenquellen anbinden können. Desweiteren ist definiert, dass alle Aktionen, welche die Daten betreffen, ebenfalls generisch implementiert sein müssen, um eine höchstmögliche Flexibilität zu gewährleisten.

Dieses Kapitel beschäftigt sich eingehend mit der Architektur sowie den Funktionen von GDCF, wobei auch beispielhaft die Erstellung einer einfachen Applikation beschrieben wird.

7.1 Kurzbeschreibung des Frameworks

GDCF bietet dem Entwickler die Möglichkeit, Client-Applikationen unter Eclipse mit geringem Entwicklungsaufwand zu erstellen. Dabei können diverse Datenquellen, wie beispielsweise Datenbanken oder das Dateisystem angebunden und in generischer Art und Weise in die Applikation integriert werden. Das Framework bietet diverse Schnittstellen zur Kommunikation mit den Datenquellen sowie zur Verwaltung der Daten.

Für die Darstellung sowie Bearbeitung stellt das Framework diverse Standard-Implementierungen bereit, die beliebig erweitert oder ersetzt werden können. Es demonstriert mit diesen Standard-Implementierungen diverse Vorgehensweisen zum

Entwickeln von Rich Client Applikationen unter Eclipse und gibt dem Entwickler einen Einblick das komplexe Thema.

Der Entwicklungsprozess einer Client-Applikation mit Hilfe von GDCF gliedert sich in vier Schritte, die auch im weiteren Verlauf dieses Kapitels ausführlich erläutert werden:

1. Definition der Verbindungen (*Connections*)
2. Definition der Navigationen (*Navigations*)
3. Definition der Aktionen (*Actions*)
4. (Anpassen der Applikation an eigene Anforderungen)

Dabei sind die ersten drei Schritte verpflichtend, da sie die Hauptbestandteile der Applikation - die Datenanbindung - darstellen. Der vierte und letzte Schritt ist optional, da nicht jede Applikation an eigene Anforderungen und Bedürfnisse angepasst werden muss – hier ist die Standard-Implementierung eventuell ausreichend.

7.2 Architektur

GDCF stellt eine Abstraktionsschicht von Elementen der Eclipse Rich Client Platform und einer beliebigen Datenquelle dar. Das Framework dient dabei als Schnittstelle zwischen Datenquelle und GUI-Elementen der Eclipse RCP und transferiert die entsprechenden Objekte und Ereignisse zwischen diesen beiden Schichten. Die Übersetzung und Verwaltung der Objekte und Ereignisse wird dabei in GDCF von diversen Managern übernommen.

GDCF gliedert sich neben anderen Eclipse Plug-Ins unterhalb der Eclipse Platform und der Eclipse RCP und oberhalb der Datenquelle ein.

Die folgende Grafik zeigt diese Eingliederung von GDCF in die Eclipse Architektur:

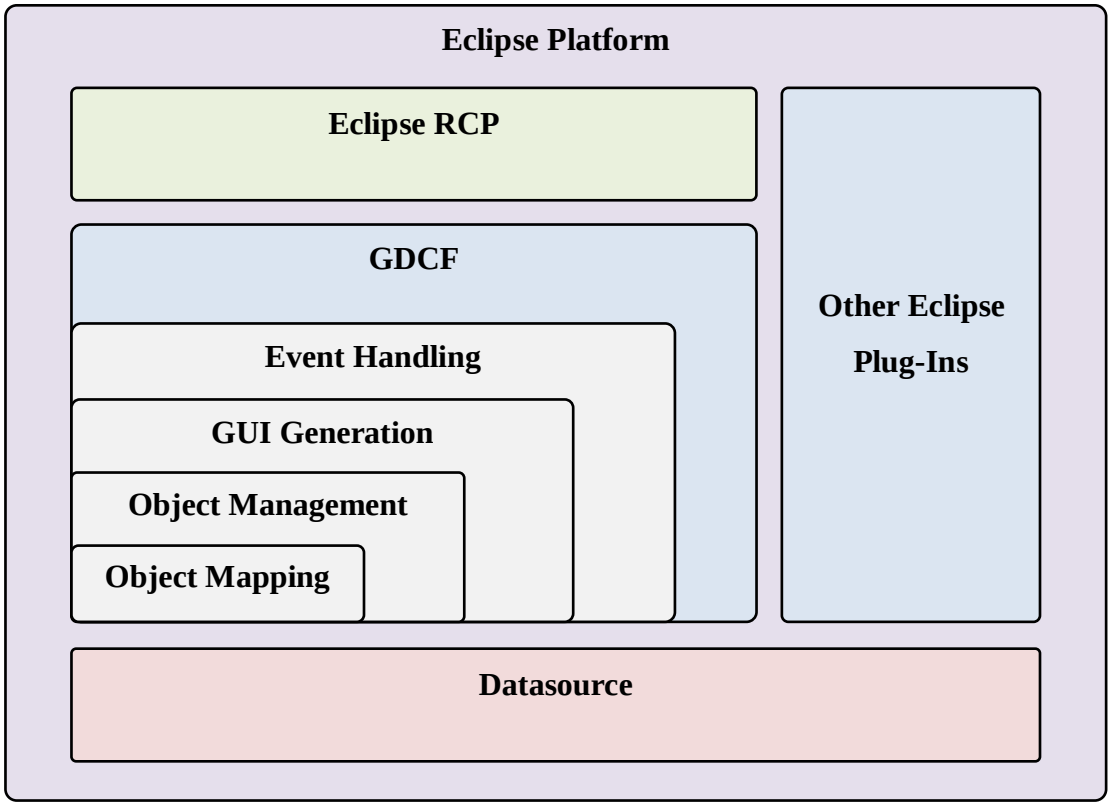

Abbildung 7.1 - GDCF Architektur

Die Übersetzung von Objekten der Datenquellen und Elementen der Eclipse RCP richtet sich grob an den Abläufen der Erstellung einer Client-Applikation aus, wie sie im letzten Kapitel beschrieben worden ist. Die Definition der Verbindungen, Navigationen sowie Aktionen erfolgt in XML-Konfigurationsdateien, welche eingelesen und in entsprechende Modelle übersetzt werden. Die Modelle sind Java-Container-Klassen, deren Variablen eine Repräsentation der XML-Konfiguration darstellen. Mit Hilfe der Modelle ist es möglich, Datenquellen anzuknüpfen und die eingehenden Daten entsprechend aufzubereiten, damit diese später verarbeitet werden können. Die Modelle stellen damit die Grundlage der einzelnen Module zum Management von Connections, Navigations und Actions dar. Für jedes dieser Module ist ein Manager zuständig, der ein Konfigurationsmodell entgegennimmt und das Modell konkretisiert, also die tatsächlichen Objekte, wie beispielsweise Connections oder Navigations, erzeugt.

Der Ablauf der Initialisierung einer Applikation lässt sich wie folgt darstellen:

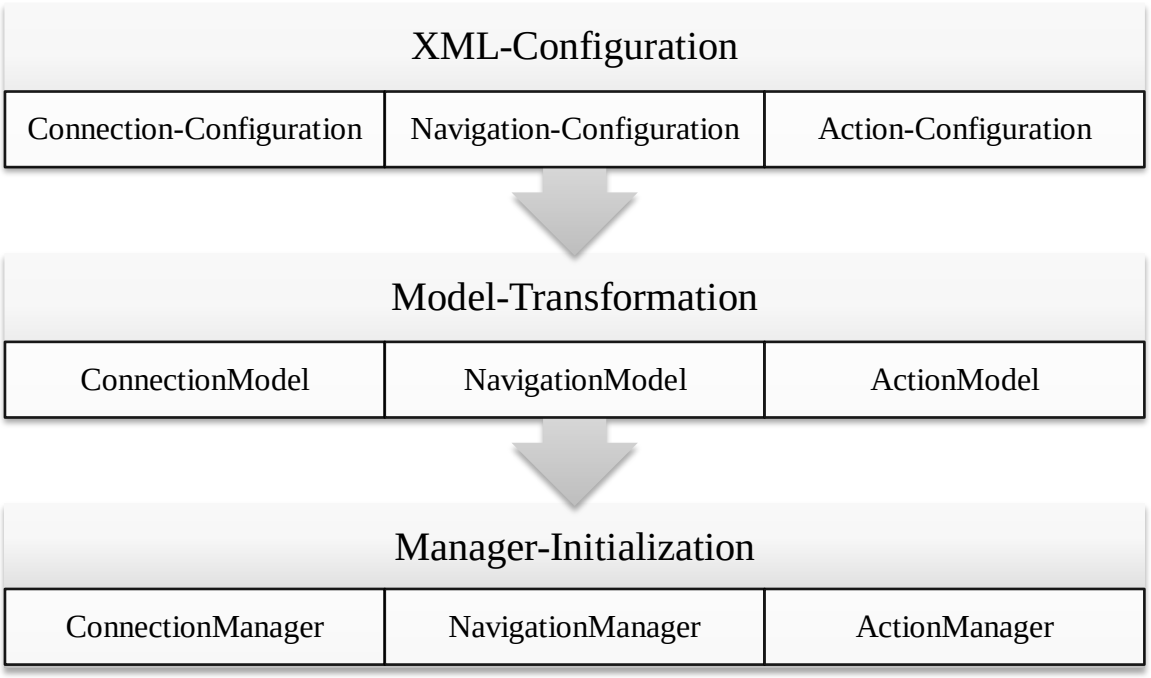

Abbildung 7.2 - GDCF Initialisierungsablauf

Die einzelnen Manager verwalten die für ihren Bereich relevanten Objekte und definieren Schnittstellen für die Kommunikation innerhalb der Applikation. So verwaltet ein Connection-Manager Connection-Objekte und ein Navigation-Manager Navigation-Objekte. Die gemanagten Objekte sind essentieller Bestandteil des Frameworks und werden im Folgenden näher erläutert.

7.2.1 Connections

Eine *Connection* ist die abstrakte Repräsentation einer Verbindung zu einer Datenquelle. Sie stellt eine Schnittstelle zu Datenquellen dar und ist ein Java-Interface. Ein Interface definiert Methoden für die Kommunikation mit der Außenwelt, die alle Klassen, die dieses Interface implementieren, beinhalten müssen. Das Interface gibt dabei die Funktionen der Methoden vor; Die zu implementierenden Klassen müssen diese Funktionen dann dementsprechend vorsehen.[88]

[88] vgl.: Java Interfaces

Das Connection Interface (*IConnection*) stellt unter anderem Methoden zum Öffnen und Schließen der Verbindung sowie zum Ausführen von Aktionen (*Actions*) bereit.

Abbildung 7.3 – IConnection/IResult-Klassendiagramme

Jede Klasse, die vom Typ *IConnection* ist, muss beispielsweise Methoden zum Öffnen, Schließen und Aufrechterhalten der Verbindung vorsehen. Innerhalb des Frameworks wird niemals mit konkreten Implementierungen gearbeitet, sofern sie generische Bereiche, wie eben Datenquellen, betreffen. So wird im Quellcode immer nur das Interface angesprochen. Die tatsächliche Implementierung ist Aufgabe des Entwicklers, wobei mindestens die Vorgaben des Interfaces erfüllt werden müssen. Diese Art der Abstraktion macht die generische Verwendung von Datenquellen überhaupt erst möglich und erfordert im Vorfeld enormen Designaufwand.

Da eine Connection die Schnittstelle zu einer Datenquelle repräsentiert, müssen über diese auch Transaktionen stattfinden können. Dies wird über die Methode *execute()* realisiert. Sie nimmt ein Action-Objekt (siehe 7.2.3) entgegen, extrahiert die relevanten Informationen und führt die Aktion für die Verbindung aus. Eine solche Action könnte beispielsweise eine SQL-Select-Anweisung auf einer Datenbank oder das Löschen einer Datei auf dem Dateisystem sein. Das Resultat der Action wird in einem eigenen Objekt zurückgegeben, das ebenfalls ein Interface (*IResult*) implementieren muss. Für eine

Datenbankverbindung muss so zum Beispiel ein entsprechendes Objekt vom Typ eines Datenbank-Resultats zurückgegeben werden. Die Connection kann dieses Resultat dann an die Applikation weiterleiten, die dann beispielsweise Darstellungen vornimmt oder dem Benutzer Rückmeldung zu der Action liefert. Das Interface *IResult* leitet sich von dem Eclipse-Interface *IStatus* ab. Ein Status repräsentiert das Ergebnis einer Operation und definiert diverse Status-Typen wie Fehler oder Warnungen und beinhaltet Informationen für den Benutzer wie Nachrichten oder Fehlermeldungen.[89]

Connections werden von einem Connection-Manager verwaltet, über den die Connections geöffnet, geschlossen oder zurückgegeben werden können. Auch der Connection-Manager ist als Interface definiert (*IConnectionManager*), besitzt aber eine funktionsfähige Basis-Implementierung (*ConnectionManager*). Das hat den Vorteil, dass Entwickler eigene Manager-Klassen programmieren und für ihre Applikationen verwenden können, ohne den restlichen Code des Frameworks anpassen zu müssen.

7.2.2 Navigations

Die Aufbereitung und Erstellung der Navigation sind die komplexesten Aufgaben des Frameworks. Die Navigation wird, wie bereits erwähnt, in einer XML-Konfigurationsdatei definiert. Das Ergebnis wird in einem Navigations-View (*NavigationView*) dargestellt, der die Daten in einer Baumstruktur (*Tree*) verwaltet. Das bedeutet, dass Elemente hierarchisch angeordnet und angezeigt werden können. So lassen sich beispielsweise Ordnerstrukturen eines Dateisystems oder 1:n-Beziehungen zwischen Objekten darstellen.

Eine Navigation ergibt sich immer aus Daten, die über eine Connection (siehe 7.2.1) geladen werden. Dazu muss in der Konfiguration eine Action (siehe 7.2.3) definiert werden, die als Resultat eine Datenstruktur zurückliefert, welche in eine Navigation übergeführt werden kann. Das könnte beispielsweise eine SQL-Select-Anweisung sein, in der eine Menge an Daten aus einer Datenbank geladen wird. Die Transformation der Daten zu Navigations-Elementen, in GDCF *NavigationItems* genannt, erfolgt ebenfalls in einer Manager-Klasse (*NavigationManager*). Diese Manager-Klasse ist, wie auch der

[89] IStatus (Eclipse Plattform API Specification)

Connection-Manager, als Interface (*INavigationManager*) gestaltet, was die Erweiterbarkeit der Funktionalität sowie die Anpassung an spezielle Anforderungen durch den Entwickler ermöglicht.

Die folgende Abbildung zeigt den Prozess von der Konfiguration zum NavigationItem.

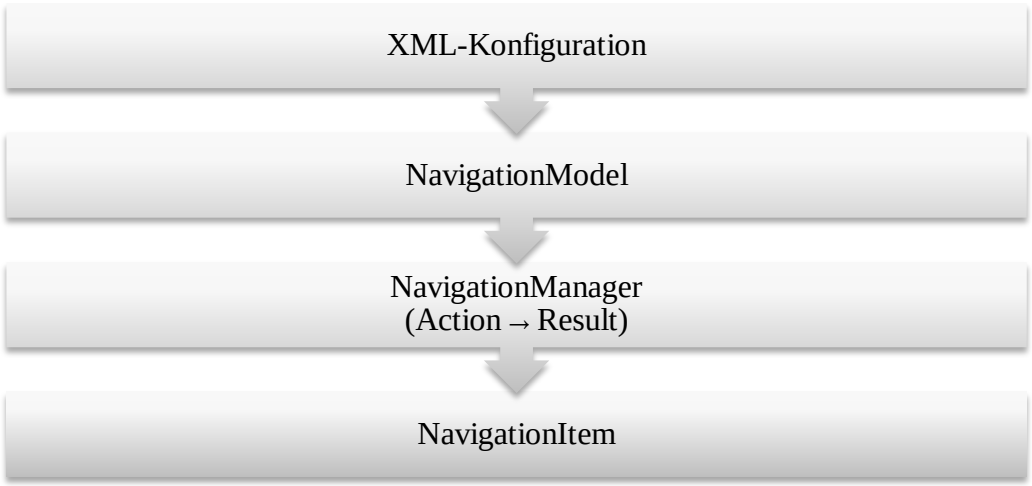

Abbildung 7.4 - Navigations-Erstellungs-Prozess

Die Konfiguration der Navigation kann hierarchisch aufgebaut werden. Das bedeutet, dass Beziehungen zwischen NavigationItems definiert werden können. Das muss in den Aktionen entsprechend berücksichtigt werden, da beim Erstellen der Navigation diese Beziehungen durch die Konfiguration auf die Elemente übertragen werden. Ein Beispiel wären zwei Datenbanktabellen mit einer 1:n-Beziehung. Dabei kann ein Element der ersten Tabelle beliebig viele „Kinder“ der zweiten Tabelle besitzen. Eine solche Abhängigkeit muss bei der Aktion zum Erstellen der Navigation definiert werden.

7.2.2.1 NavigationItems

Ist die Navigationsstruktur definiert und für das Framework übersetzbar, so ist das Ergebnis eine Menge von NavigationItems, die, je nach Hierarchiedefinition, miteinander in Verbindung stehen. Ein NavigationItem ist zentraler Bestandteil von GDCF, da fast alle Komponenten des Frameworks damit operieren.

Jeder NavigationItem muss das Interface *INavigationItem* implementieren, welches alle Kommunikationen und Aktionen mit den Elementen festlegt.

Die folgende Abbildung zeigt das Klassendiagramm des Interfaces:

Abbildung 7.5 - INavigationItem-Klassendiagramm

Die meisten der angeführten Methoden dienen der Verwaltung der Hierarchie. Es können Elemente hinzugefügt, gelöscht oder zurückgegeben werden. Diese Methoden werden beispielsweise verwendet, um dem Navigationsbaum neue Elemente hinzuzufügen. In diesem Fall würde ein Kind-Element (*Child*) an ein bestehendes Vater-Element (*Parent*) angehängt werden.

Jeder NavigationItem besitzt eine ID, die sich gewöhnlich aus dem Klassennamen und einer eindeutigen Bezeichnung, zum Beispiel der ID aus der Datenbanktabelle, zusammensetzt. Diese ID wird für die Verwaltung und Ansprechbarkeit des

NavigationItems verwendet. Neben der ID besitzt jeder NavigationItem einen Namen sowie ein Icon, mit denen er im Navigationsbaum repräsentiert wird.

Zusätzlich sind Methoden zum Entfernen, Updaten oder Aktualisieren der Elemente vorgesehen, die an entsprechende Actions geknüpft sein müssen.

7.2.3 Actions

Die Darstellung der NavigationItems alleine ist für eine Applikation in der Regel zu wenig. Um mit den Elementen arbeiten zu können, müssen *Actions* definiert werden. Eine Action ist in GDCF die Definition einer Aufgabe, die erledigt werden soll. Eine Action ist ebenso abstrakt wie eine Connection oder ein NavigationItem. Actions sind daher nur Hüllen um konkrete Befehle, die über eine Connection ausgeführt werden können. Neben dem Befehl besitzt eine Action auch Metainformationen, wie Art oder Typ. So unterscheidet sich eine Action zum Löschen eines Elements von der zum Öffnen desselben, oder eine Action zum Umbenennen von einer zum Aktualisieren. Diese Typen sind fester Bestandteil von GDCF und bieten Basisfunktionalitäten für gängige Aufgaben. Wie bereits in Abschnitt 3.3.3 als Anforderung definiert, beinhaltet das Framework folgende standardmäßige Aktionstypen:

- Öffnen
- Bearbeiten
- Speichern
- Anlegen
- Löschen
- Umbenennen
- Aktualisieren

Der Grund für die Definition dieser Typen ist die damit verbundene unterschiedliche Reaktion. Wenn der Benutzer beispielsweise das Objekt eines NavigationItems öffnet, so soll ein neuer Editor erscheinen, der die Bearbeitung des Objekts ermöglicht. Wenn er das Objekt jedoch umbenennt, so muss ein Dialog zum Umbenennen erscheinen. Dasselbe gilt auch für alle anderen Typen, die jeweils unterschiedliche Reaktionen

erfordern. Für jede Action kann somit der entsprechende Typ definiert werden. Wird kein Typ angegeben, so wird die Action ohne weitere Behandlungen ausgeführt. Das garantiert, dass auch Actions, die unabhängig von diesen Typen sind, in die Applikation integriert und ausgeführt werden können.

Neben den Typen gibt es noch die Unterscheidung nach dem Modus der Action. So gibt es Actions, die Daten aus der Quelle abziehen und solche, die Daten in eine Quelle schreiben beziehungsweise updaten. Das lässt sich ebenfalls an dem Beispiel der Datenbank verdeutlichen: Werden lediglich Daten ausgelesen, so muss eine Abfrage auf der Datenbank ausgeführt werden. Sollen jedoch Daten verändert oder hinzugefügt werden, muss ein Update vorgenommen werden.

Die Definition der Actions wird ebenfalls in einer XML-Konfigurationsdatei vorgenommen, wo auch entsprechenden Befehle, die über die Connection ausgeführt werden sollen, definiert werden. Eine Action muss immer konkret zu einer Connection zugeordnet werden, das heißt sie muss ebenfalls vom Entwickler implementiert werden. Die Kommunikation definiert das Interface *IGenericAction*, das von dem Eclipse-internen Interface *IAction* abgeleitet ist.

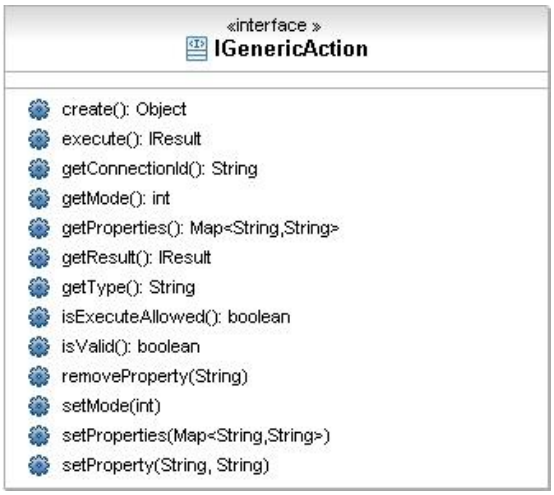

Abbildung 7.6 - IGenericAction-Klassendiagramm

Eine Action besitzt diverse Metainformationen sowie die Fähigkeit, den Befehl zum Ausführen über die Connection zu erstellen - dies erfolgt über die Methode *create()*.

Diese Methode erzeugt ein beliebiges Objekt, das von der korrespondierenden Connection-Klasse verarbeitet wird. Für das Beispiel der Datenbankanbindung könnte dies ein SQL-Kommando sein, das über JDBC an die Datenbank geschickt wird. Hier bleibt es grundsätzlich dem Entwickler überlassen, welche Form des Datenaustauschs verwendet wird. Ebenso könnte ein spezielles Objekt übermittelt werden, das beispielsweise zusätzliche, sicherheitsrelevante Funktionen bietet und vor der Ausführung noch diverse Überprüfungen vornimmt. Das Interface legt hier keine Implementierungsvorgaben fest und erlaubt die individuelle Gestaltung der relevanten Abläufe.

Die Ausführung einer IGenericAction übernimmt der *GenericActionHandler*, eine Klasse, die anhand der ID eines NavigationItems sowie der Action-ID die korrekte Action erzeugt und ausführt. Diese Klasse kann auch für Actions verwendet werden, die nicht von GDCF, sondern über einen Extension Point der Eclipse Plattform eingefügt worden sind. Das bietet den Vorteil, dass Actions außerhalb von GDCF hinzugefügt, aber generisch in der XML-Konfiguration definiert werden können. Dieser Ablauf wird in Abschnitt 7.3 anhand eines Beispiels veranschaulicht.

Für die Verwaltung von Actions ist die Klasse *ActionManager* zuständig. Dieser ist ebenfalls in einem Interface (*IActionManager*) definiert und kann vom Entwickler selbst implementiert oder erweitert werden.

7.2.4 Views

Views können grundsätzlich vom Entwickler hinzugefügt werden und müssen daher keinen Vorgaben folgen. Anders ist dies bei Navigations-Views. Diese müssen das Interface *INavigationView* implementieren, das diverse Methoden zur Verwaltung von NavigationItems definiert. Ein Navigations-View kann dabei Funktionen wie zum Beispiel Filterung, Sortierung oder Synchronisation mit Editoren bereitstellen. Die Standard-Implementierung des INavigationView-Interfaces ist der *NavigationView*. Dieser erlaubt es, Elemente zu filtern, mit dem Editor zu synchronisieren und definiert ein Kontextmenü, das sich auf Basis der definierten Actions automatisch erstellt. Dieses beinhaltet beispielsweise das Aktualisieren der Navigation oder das Erweitern (*expand*)

oder Reduzieren (*collapse*) einzelner NavigationItems. Wenn zusätzliche Funktionalität erforderlich ist, kann diese Klasse entweder ersetzt oder erweitert werden.

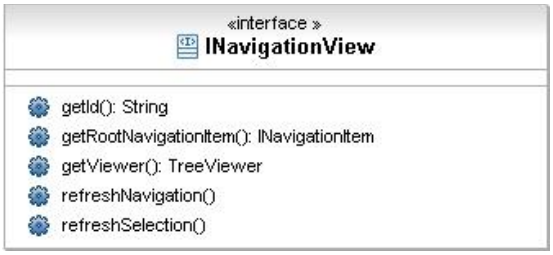

Abbildung 7.7 - NavigationView-Klassendiagramm

7.2.5 Editoren

Die Implementierung von Editoren ist von GDCF nicht ausformuliert. Das bedeutet, dass grundsätzlich jede Art von Editor, ohne Berücksichtigung von GDCF-Spezifika, implementiert werden kann. Wichtig ist jedoch die Form, in der die Daten an den Editor geliefert werden. In Eclipse werden Daten in Form eines *IEditorInput*-Objekts an den Editor geliefert. Dabei ist ein IEditorInput-Objekt eine Beschreibung des zu editierenden Elements, wie beispielsweise ein Dateiname. [90] In GDCF werden Elemente als *GenericEditorInput*-Objekte übergeben, die speziell auf NavigationItems ausgerichtet sind. So kann über einen GenericEditorInput der in dem korrespondierenden View selektiere NavigationItem angesprochen werden.

Die folgende Abbildung zeigt das Klassendiagramm des *IGenericEditorInput*-Interfaces:

Abbildung 7.8 - GenericEditorInput-Klassendiagramm

Dabei muss der Editor mit dem GenericEditorInput umgehen können und entsprechend programmiert werden. Der Editor kann dann direkt auf den selektierten NavigationItem aus dem entsprechenden NavigationView zugreifen und damit arbeiten. Wenn der

[90] IEditorInput (Eclipse Plattform API Specification)

NavigationItem gespeichert werden soll, muss der Editor dies entsprechend handhaben und über die Manager-Klasse eine Action generieren und ausführen.

7.3 Funktionalität und Implementierung

Dieses Kapitel beschäftigt sich umfassend mit den Funktionalitäten von GDCF sowie den Konfigurations- und Erweiterungsmöglichkeiten. Desweiteren wird die Implementierung einer einfachen Applikation beschrieben, welche die wichtigsten Funktionen des Frameworks nutzt und zur Verfügung stellt.

Das Beispiel umfasst die Erstellung von zwei Connections, der Navigations und der Actions auf die Elemente. Dabei werden die wichtigsten Funktionen von GDCF sowie die Funktionsweisen der Views, Editoren und Perspektiven ausführlich beschrieben.

7.3.1 Anforderungen an die Beispiel-Applikation

Es soll eine Client-Applikation erstellt werden, welche das Verwalten von Firmen (*Companies*), Managern (*Managers*) und Angestellten (*Employees*) ermöglichen soll. Dabei soll eine Company mehrere Manager und diese wiederum mehrere Employees unterstellt haben können.

Als Hauptdatenquelle dient eine Datenbank mit Tabellen und Relationen zu den jeweiligen Elementen. Als Zusatzdatenquelle wird das Dateisystem angebunden, um neue, anzukaufende Firmen und Notizen zu diesen Firmen verwalten zu können.

Für eine Firma soll ein Name definiert werden können, für Manager und Angestellte sind die Eingabe von Vor- und Nachname erforderlich. Die einzelnen Objekte sollen geöffnet, bearbeitet, umbenannt und gelöscht werden können. Zudem soll es möglich sein, neue Objekte anzulegen.

7.3.1.1 Anmerkungen

Das Beispiel demonstriert somit die Verwendung mehrerer Connections und Navigations und beschreibt eine Reihe von Actions, die das Arbeiten mit den Elementen ermöglicht. Hier soll aber darauf hingewiesen werden, dass aufgrund der beschränkten

Demonstrationsmöglichkeiten in dieser Arbeit, das Beispiel eher einfach gehalten worden ist und dass GDCF auch für das Erstellen von komplexeren Applikationen geeignet ist.

7.3.2 Oberfläche der Applikation

Wie bereits in Kapitel 5.5.4 angemerkt, basieren GDCF-Applikationen auf der Struktur des Eclipse Workbenchs. Das bedeutet, dass Menüs, Toolbars, Views und Editoren verwendet werden können.

Die folgende Abbildung zeigt die mögliche Oberflächen-Struktur einer solchen Applikation:

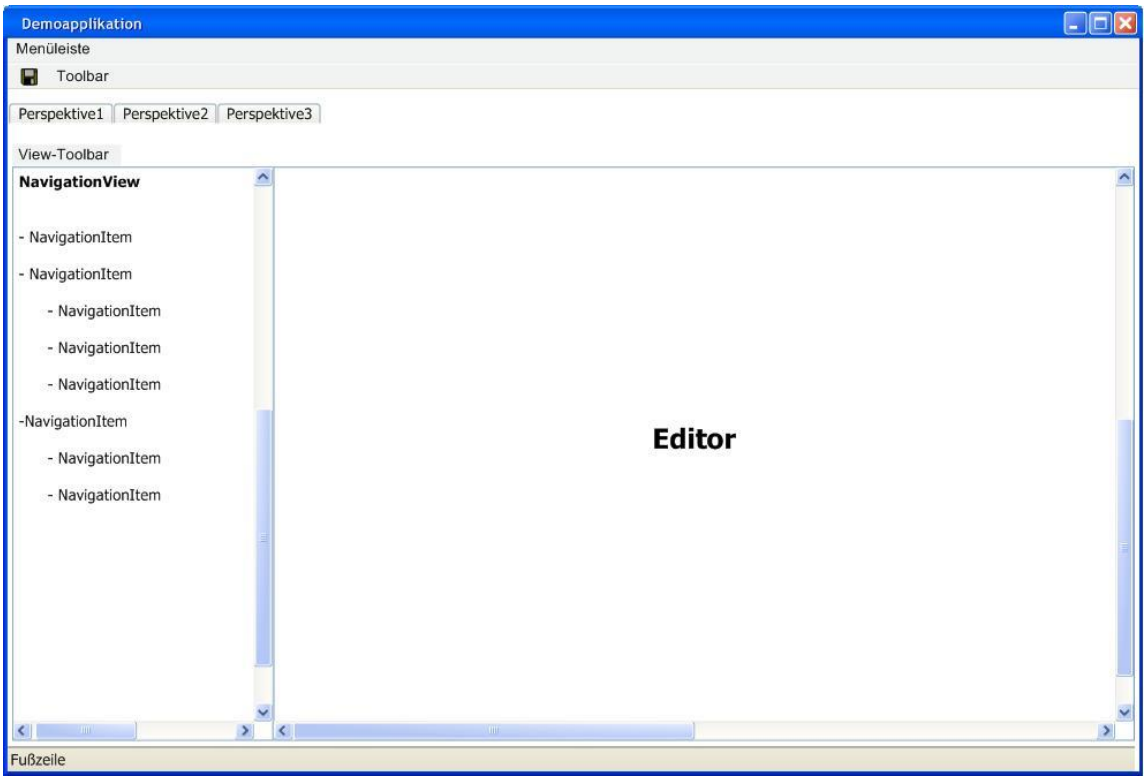

Abbildung 7.9 - Applikations-Oberfläche

Dabei können die einzelnen Elemente auch ausgeblendet und die Views beliebig positioniert werden, beispielsweise auf der rechten Seite.

Die Reiter für die Perspektiven erlauben es, zwischen den verschiedenen Navigationen zu wechseln. So lässt sich beispielsweise von der Datenbank- auf die Dateisystem-Navigation umschalten, um auf deren Elemente zugreifen zu können. Der Editorbereich bleibt dabei immer Bestandteil der Applikation und kann Elemente von verschiedenen NavigationViews darstellen.

7.3.3 Ablauf der Applikations-Erstellung

Um eine Client-Applikation mit GCDF erstellen zu können, müssen folgende Implementierungsschritte, die teilweise bereits in Kapitel 7.1 beschrieben worden sind, in der Reihenfolge durchgeführt werden:

1. Definition der Connections in einer XML-Konfigurationsdatei
2. Definition der Navigations und NavigationItems unter Berücksichtigung deren Relationen
3. Definition der Actions auf die NavigationItems
    a. Definition der Basis-Aktionen
    b. (Definition zusätzlicher Aktionen über das Eclipse Plug-In-Development)
4. Implementierung und Konfiguration der Editoren
5. Anpassen des Workbenchs
6. Anpassen der NavigationViews
7. Anpassen der Menüs
8. Anpassen der Toolbars
9. (Implementierung eigener Funktionalität)

Im Folgenden werden diese Schritte sowie die Konfigurationsmöglichkeiten in GDCF näher erläutert.

7.3.4 Definition der Connections

Der erste Schritt zur Erstellung der Applikation ist die Definition der Connections, welche die Schnittstellen zu den Datenquellen darstellen. Diese müssen grundsätzlich vom Entwickler implementiert werden, da die Anbindung von Datenquellen in GDCF sehr abstrakt gehalten ist. Klassen müssen dabei das Interface *IConnection* implementieren, das bereits in Kapitel 7.2.1 eingehend beschrieben worden ist.

Im Falle der Datenbank-Anbindung könnte diese Klasse beispielsweise *DatabaseConnection* heißen und mit SQL-Kommandos umgehen können. Diese Klasse ist bereits Bestandteil der Beispielimplementierungen des Frameworks und erlaubt, auf sehr einfache Weise, das Absetzen von SQL-Kommandos auf eine Datenbank über JDBC.

Für die Dateisystem-Anbindung gibt es ebenfalls eine Beispielklasse (*FileSystemConnection*), die es erlaubt, gängige Operationen auf Dateien und Ordner auszuführen.

Connections müssen in einer eigenen XML-Konfigurationsdatei definiert werden, die von GDCF ausgelesen und in entsprechende Modelle übersetzt wird. Eine Connection hat immer eine eindeutige ID sowie eine beliebige Anzahl an Eigenschaften (*Properties*). Jedes Property-Element hat einen Schlüssel und einen Wert. GDCF gibt dabei keine Vorgaben für Schlüssel und Werte an – diese müssen von der implementierenden Connection-Klasse verarbeitet werden. Eine DatabaseConnection erwartet beispielsweise folgende Parameter:

driver	Der Datenbanktreiber
connectionurl	Die URL zur Anbindung der Datenbank
username	Der Benutzername
password	Das Passwort

Tabelle 7.1 - DatabaseConnection Parameter

Diese Parameter sind für die Erstellung einer Datenbankverbindung über JDBC erforderlich und werden von der Klasse DatabaseConnection verarbeitet.

Die XML-Konfiguration für die DatabaseConnection mit einer beispielhaften MySQL-Datenbank könnte wie folgt aussehen:

```
<connection
      id="databaseConnection"
      connectionClass="DatabaseConnection">
      <properties>
            <property key="driver" value="com.mysql.jdbc.Driver" />
            <property
                  key="connectionurl"
                  value="jdbc:mysql://localhost:3306/test" />
            <property
                  key="username"
                  value="testuser" />
            <property
                  key="password"
                  value="test" />
      </properties>
</connection>
```

Die angegebene ID wird verwendet, um die Connection jederzeit identifizieren und aufrufen zu können – sie muss daher einmalig sein. Das Attribut *connectionClass* gibt an, welche Connection-Klasse von GDCF für die Verbindung instanziiert werden soll.

Die Definition der Dateisystem-Connection ist einfacher, hier muss lediglich eine eindeutige ID angegeben werden:

```
<connection
      id="fileSystemConnection"
      connectionClass="FileSystemConnection">
      <properties/>
</connection>
```

Der Entwickler hat hier die Möglichkeit, zusätzliche Properties zu spezifizieren und eigene Connection-Klassen zu implementieren, die diese Properties entsprechend verarbeiten können – die oben dargestellten Konfigurationen dienen lediglich der Veranschaulichung der Konfigurationsmöglichkeiten.

7.3.5 Definition der Navigations

Das Definieren der Navigation ist ein sehr komplexer Part und erfordert große Aufmerksamkeit. Wie bereits in Kapitel 7.2.2 beschrieben, besteht eine Navigation aus mehreren NavigationItems, die eine Repräsentation der Datenelemente aus der Datenquelle sind. Aufgrund der Verwendung einer Baumstruktur für die Darstellung, können komplexe Hierarchien abgebildet werden – bei einer tabellarischen Darstellung wäre dies hingegen nicht möglich. Wie auch bei Connections werden die Navigations in einer XML-Konfigurationsdatei definiert und von GDCF in ein Java-Modell übersetzt. Dieses Modell ist ebenfalls hierarchisch aufgebaut und kann somit einfache Eltern-Kind-Beziehungen abbilden.

Für das Beispiel werden für die Navigation drei Tabellen benötigt, welche die in Abschnitt 7.3.1 definierten Anforderungen erfüllen müssen. Es müssen somit die Tabellen „Companies“, „Manager“ und „Employees“ erstellt und deren Attribute festgelegt werden. Zudem ist definiert worden, dass ein Employee immer einem Manager untersteht, ein Manager aber mehrere Employees unterstellt haben kann. Desweiteren gehört ein Manager immer zu einer Company, wobei eine Company aber mehrere Manager angestellt haben kann. Es ergeben sich also zwei klassische 1:n-Relationen, die sich wie folgt darstellen lassen:

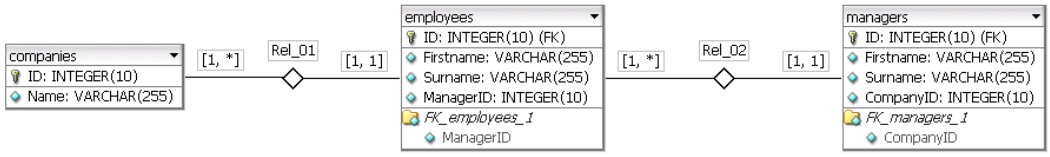

Abbildung 7.10 - ER-Modell der Beispiel-Applikation

Dieses Modell kann in ähnlicher Form auf GDCF übertragen werden, was im Folgenden genauer erläutert wird.

In der XML-Konfigurationsdatei muss zunächst eine neue Navigation definiert werden.

```
<navigation
      id="CompanyNavigationView"
      class="CompanyNavigationView"
      name="Companies"
      connectionId="databaseConnection">
```

Die ID dient der Identifikation der Navigation und muss einmalig sein. Das Attribut *class* gibt den NavigationView an, der die Navigation beinhalten soll. Dies muss der Pfad zu einer Java-Klasse sein, die von *INavigationView* abgeleitet ist. Diese wird von GDCF instanziiert und übernimmt die Kontrolle über die erstellten NavigationItems. Der Name gibt an, welchen Titel der NavigationView auf der Oberfläche erhalten soll. Die *connectionId* referenziert auf die entsprechende Connection-Klasse, welche an die Datenquelle angebunden ist und die entsprechenden Daten zum Erstellen der NavigationItems liefert.

Der nächste Schritt besteht in der Definition der obersten Hierarchieebene – den Companies. Die Konfiguration des NavigationItems für Companies lautet wie folgt:

```
<navigationItem
      class="Company"
      id="${id}"
      name="${name}"
      group="company"
      icon="icons/building.png">
      <populateAction class="DatabaseAction">
            <properties>
                  <property
                        key="sql"
                        value="select * from companies" />
            </properties>
      </populateAction>
```

```
<itemData>
            <data
                  key="id"
                  value="${id}" />
      </itemData>
</navigationItem>
```

Diese Konfiguration ist für die Darstellung der NavigationItems essentiell und bedarf näherer Erläuterungen. Im Folgenden werden daher die Attribute und Elemente sowie deren Funktionsweisen genauer erörtert.

7.3.5.1 Das Attribut „class“

Das *class*-Attribut spezifiziert den NavigationItem - eine Klasse, die das INavigationItem-Interface implementiert. Es sollte für jeden NavigationItem eine Klasse erstellt werden, da die Instanz für diverse Überprüfungen und Generierung von Actions verwendet wird. Die Klasse kann von der Basis-Implementierung des INavigationItem-Interfaces, der Klasse *NavigationItem,* abgeleitet werden – so müssen keine weiteren Implementierungen vorgenommen werden. GDCF erstellt für jeden NavigationItem eine Instanz der spezifizierten Klasse.

7.3.5.2 Das Attribut „id“

Das *id*-Attribut setzt die ID für den jeweiligen NavigationItem fest. Da in dem Modell des NavigationItems keine expliziten Ausprägungen oder konkrete Instanziierungen bekannt sind, aber jeder NavigationItem eine einmalige ID besitzt, muss es generisch festgelegt werden. Das bedeutet, dass jeder NavigationItem bei der Erstellung eine eindeutige und einmalige ID erhalten muss. Die ID wird in der Regel aus den Datensätzen der Datenquelle herausgelesen und kann beispielsweise die ID einer Datenbanktabelle oder ein Dateiname sein. In GDCF lassen sich solche Daten mit der Syntax *${Feldname}* aus der Datenquelle extrahieren. Im Beispiel wird als ID die korrespondierende ID aus der Tabelle *Companies* angegeben. Damit versucht GDCF beim Erstellen des NavigationItems aus dem aktuellen Datensatz diese ID zu extrahieren. Die Datenstruktur ist Aufgrund des *IResult*-Interfaces, das bereits in Abschnitt 7.2.1 beschrieben worden ist, immer gleich zu interpretieren. Damit ist es

möglich, auf generische Weise spezifische Felder aus Datenquellen zu lesen. Da jedoch IDs von Datenbanktabellen oft über mehrere Tabellen hinweg gleich sind, sollte sich die ID eines NavigationItems, wie bei der Basis-Implementierung umgesetzt, aus dem Klassennamen sowie der ID zusammensetzen. Damit ist gewährleistet, dass auch über mehrere Tabellen hinweg gleiche IDs für die Identifizierung der NavigationItems verwendet werde können. Der Feldname gibt in diesem Fall den Spaltennamen der Datenbanktabelle an. Hier könnte auch jeder andere, zur Identifikation geeignete, Feldname angegeben werden.

7.3.5.3 Das Attribut „name“

Das *name*-Attribut steuert die Darstellung des NavigationItems auf der Oberfläche. Der Name wird im NavigationView sowie in den Editoren angezeigt und kann ebenfalls mit der in 7.3.5.2 beschriebenen Syntax definiert werden. Hier können auch mehrere Felder kombiniert werden, wie etwa durch die Schreibweise:

```
name="${firstname} ${surname}"
```

In diesem Fall würde sich der Name aus den Feldern *firstname* und *surname* zusammensetzen. Es können auch beliebige Zeichenfolgen definiert oder Zeichenfolgen mit Feldwerten kombiniert werden, zum Beispiel:

```
name="Company: ${name}"
```

7.3.5.4 Das Attribut „group“

Das *group*-Attribut ist für die Darstellung von Hierarchien relevant. Jedem NavigationItem kann eine Gruppe zugewiesen werden. Wie im weiteren Verlauf dieses Kapitels noch beschrieben wird, dient diese Gruppe zur Referenzierung auf ein übergeordnetes Objekt in der Navigation. Hier empfiehlt sich die Angabe eines sprechenden Namens, wie etwa „company“ für dieses Beispiel.

7.3.5.5 Das Attribut „icon“

Über das *icon*-Attribut kann dem NavigationItem ein Icon zugewiesen werden. Dieses sollte 16x16 Pixel groß und für den NavigationItem passend sein. Im Beispiel wurde ein Icon mit einem Gebäude angegeben, wie auf den folgenden Screenshots zu sehen ist.

7.3.5.6 Das Element „populateAction“

Dieses Element ermöglicht die Abfrage der Daten aus der Datenquelle und eine Umlegung auf die Navigation. Eine *populateAction* ist eine Action, die für die Erstellung der Navigation zuständig ist. Hier muss die Action-Klasse definiert werden, die das Interface *IGenericAction* implementiert. In der oben dargestellten Konfiguration ist dies die Klasse *DatabaseAction*, die bereits für das Framework beispielhaft implementiert worden ist. Diese nimmt eine Reihe von Parametern entgegen, wie zum Beispiel eine SQL-Select-Anweisung und übersetzt diese in eine Form, welche die Datenquelle versteht. Die Connection-Klasse führt diese Action aus und liefert ein IResult-Objekt zurück. Aus diesem werden anschließend die Daten extrahiert und die Navigation aufgebaut.

Im Beispiel werden von der Tabelle „Companies“ alle Datensätze selektiert und zurückgegeben. Der NavigationManager (siehe 7.2.2) erzeugt aus diesen Datensätzen mit Hilfe des Modells aus der Konfiguration jeden einzelnen NavigationItem und weist ihm seine Daten zu. So werden aus einem Modell beliebig viele NavigationItems erzeugt.

7.3.5.7 Das Element „itemData“

Das Element *itemData* definiert zusätzliche Daten, die jeder NavigationItem bei der Initialisierung erhält. Die Daten bestehen immer jeweils aus einem Schlüssel und einem Wert. Mit der Syntax *${Feldname}* kann hier wieder ein Wert aus der Datenquelle zugewiesen werden. Dies ist vor allem für die Darstellung von Hierarchien relevant, wie im weitern Verlauf noch erklärt wird.

Wenn die Konfigurationen vorgenommen worden sind, kann die Applikation zum ersten Mal getestet werden. Es sollten nach dem Öffnen der Applikation nun so viele Companies dargestellt werden, wie in der Datenbanktabelle eingetragen sind.

Die folgende Abbildung zeigt die erste Ausgabe:

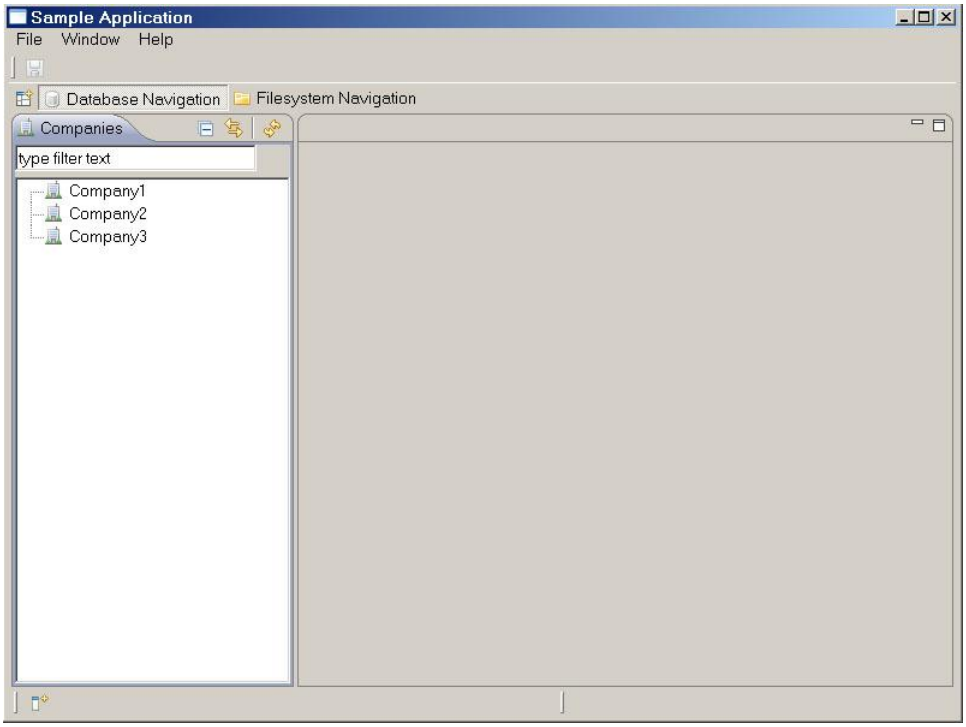

Abbildung 7.11 - Testapplikation 1

In der Datenbank befinden sich drei Einträge, die nun in der Navigation aufscheinen. Der Name ist dabei von der Spalte „name“ aus der Datenbanktabelle extrahiert worden. Das Icon stammt aus der Konfigurationsdatei.

Da nun die Companies angezeigt werden, sollen als nächstes die Manager in die Navigation aufgenommen werden. Dazu muss auch erstmals die Hierarchie berücksichtigt werden, da Manager Companies zugeordnet sein können.

Die Konfiguration der Manager-NavigationItems lautet wie folgt:

```
<navigationItem
      class="Manager"
      id="${id}"
      name="${firstname} ${surname}"
      group="manager"
      parentGroup="company"
```

```
      icon="icons/user_suit.png">
      <populateAction class="DatabaseAction">
            <properties>
                  <property
                        key="sql"
                        value="select * from managers where
                                companyId=${company.id}" />
            </properties>
      </populateAction>
      <itemData>
            <data
                  key="id"
                  value="${id}" />
      </itemData>
</navigationItem>
```

Diese Konfiguration unterscheidet sich aus der zuvor dargestellten durch zwei wesentliche Merkmale: Das Attribut *parentGroup* sowie das Element *populateAction*.

7.3.5.8 Das Attribut „parentGroup“

Mit Hilfe dieses Attributs können Hierarchien in dem Navigations-Modell abgebildet werden. Durch Definition des *parentGroup*-Attributs wird die direkte Zugehörigkeit eines NavigationItems zu einem anderen definiert. Dies wird zunächst auf dem Modell abgebildet und schließlich für die Erstellung der NavigationItems herangezogen. Im Beispiel ist definiert, dass ein Manager das Unterelement einer Company ist. Daher muss in der Manager-Konfiguration das parentGroup-Attribut auf „company“ gesetzt werden, da dort die Gruppe entsprechend definiert worden ist.

7.3.5.9 populateActions und Hierarchien

Wie in der Konfiguration ersichtlich, wird hier der *populateAction* ein SQL-Kommando übergeben, das die Zeichenfolge „${company.id}“ enthält. In GDCF können Actions auf Basis von bereits zuvor ausgeführten Befehlen erzeugt werden. In diesem Fall wird das SQL-Kommando übersetzt und alle Zeichenketten mit der Syntax *${Gruppe.Feldname}* durch entsprechende Werte ersetzt. Der NavigationManager erzeugt dabei zunächst das erste Company-Objekt und gleich anschließend alle weiteren Manager-Objekte, die dem

Company-Objekt unterstehen. Durch die Syntax kann dann auf Daten des übergeordneten Objekts zugegriffen werden, sofern diese explizit in der Konfiguration unter dem Element *itemData* spezifiziert worden sind. Aus diesem Grund ist es erforderlich, bei der Konfiguration der Company die ID explizit als itemData zu definieren. Dieser Zugriff ist auch über mehrere Hierarchieebenen möglich und gestattet desweiten die Übersetzung von itemData-Elementen, IDs und Namen.

So könnte für einen Manager beispielsweise folgender Name definiert werden:

```
name="${company.id} - ${firstname} ${surname}"
```

Die Darstellung des Namens könnte dann beispielsweise folgendermaßen aussehen:

„1 - Max Mustermann“

Diese flexible Technik erlaubt auch die Erstellung von komplexeren Abfragen und individuelleren Namensgebungen.

Die Definition der Employees gestaltet sich ebenso wie die der Manager, es wird lediglich die *group* und *parentGroup*, die *populateAction* sowie das *icon-Attribut* angepasst.

Nach erneutem Starten der Applikation und unter der Annahme beispielhafter Einträge in der Datenbank, ergibt sich folgende Navigation:

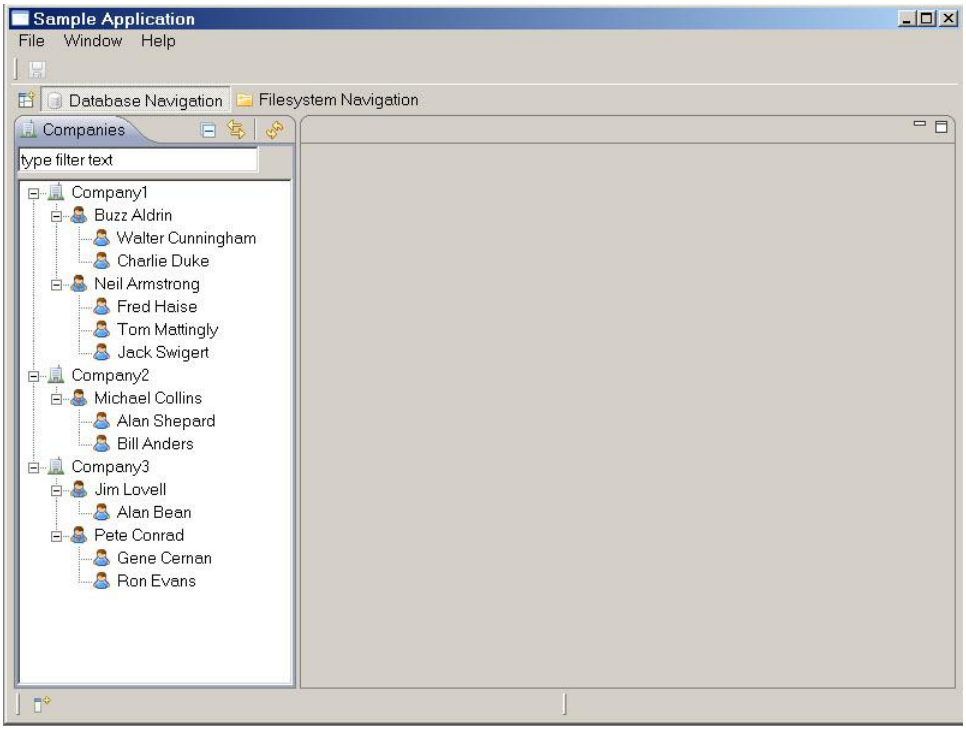

Abbildung 7.12 - Testapplikation 2

Die Anbindung des Dateisystems gestaltet sich ähnlich. In der Konfiguration wird ein neues *navigation*-Element definiert und die relevanten NavigationItems darunter eingetragen.

Als Connection muss die entsprechende Klasse (*FileSystemConnection*) angegeben werden. Die Actions müssen angepasst und auf *FileSystemActions* umgestellt werden. Diese sind zu Demonstrationszwecken für GDCF bereits implementiert worden.

Über die Auswahl der Perspektive kann schließlich diese Navigation angewählt werden und präsentiert sich dann wie folgt:

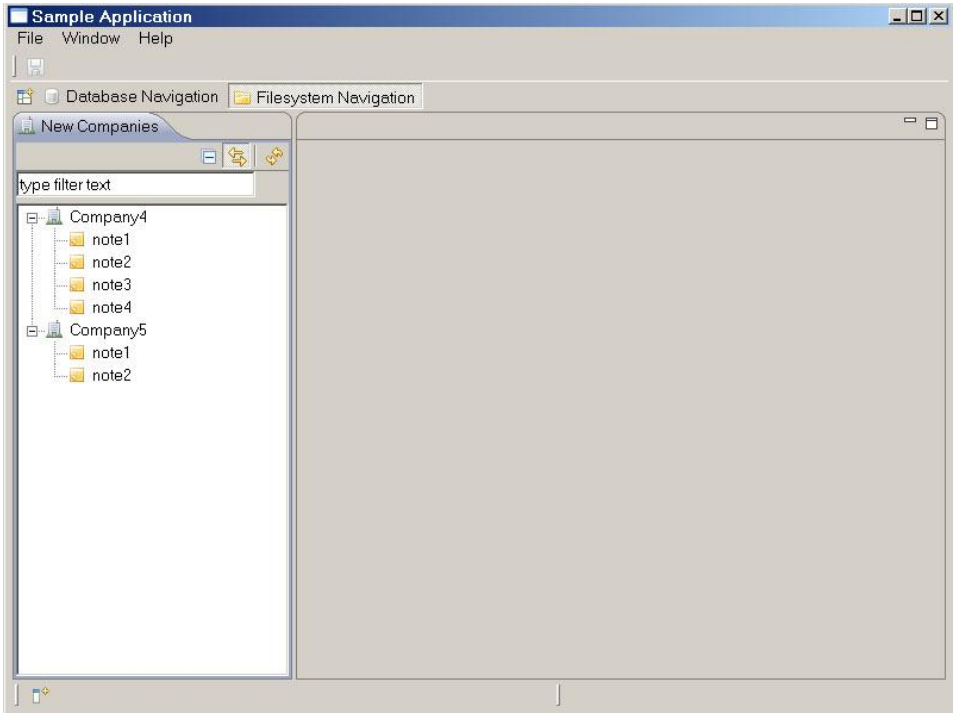

Abbildung 7.13 - Testapplikation 3

Da nun die Navigationen komplett sind, werden als nächstes Actions definiert, die ein Arbeiten mit den NavigationItems ermöglichen.

7.3.6 Definition von Actions

Actions stellen die Interaktion mit den NavigationItems dar und erlauben den Zugriff auf die darunter liegende Datenquelle. So können NavigationItems beispielsweise angelegt, geöffnet, bearbeitet und gespeichert, gelöscht, umbenannt oder aktualisiert werden.

Wie bereits in Abschnitt 7.2.3 beschrieben, müssen Actions in unterschiedliche Gruppen eingeteilt werden, da jede Art von Action anders behandelt werden muss. Wird beispielsweise ein NavigationItem geöffnet, so muss ein entsprechender Editor instanziiert werden, der die Daten des NavigationItems darstellen kann. Wird der NavigationItem gelöscht, so soll vorher ein Ja/Nein-Dialog zur Bestätigung angezeigt werden. Wenn der NavigationItem umbenannt wird, muss ebenfalls ein entsprechender Dialog erscheinen, der das Umbenennen ermöglicht.

7.3.6.1 Hintergrund

Für diverse Actions gibt es in GDCF vorgesehene Abläufe, die sich an die Actions von Eclipse anlehnen. Dort repräsentiert eine Action ein Kommando, das von einem Benutzer erteilt und üblicherweise über GUI-Elemente, wie Buttons oder Toolbars ausgelöst wird. Für Actions in Eclipse ist das Interface *IAction* vorgesehen, welches diverse Methoden zur Darstellung und Ausführung definiert. So hat jede Action eine eindeutige ID, einen Namen sowie ein Icon. Desweiteren definiert das Interface eine *run*-Methode, die zum Ausführen der Action dient. Diese Methode ist grundsätzlich an keinerlei Implementierungsvorschriften gebunden, es kann hier also jede Art von Code ausgeführt werden.[91] GDCF nutzt diesen Mechanismus und erweitert die Eclipse Action-Klasse um Funktionen auf generische Datenquellen. Diese Klasse dient zum Ausführen der definierten Actions unter Berücksichtigung der jeweiligen Art (Öffnen, Löschen, Umbenennen, etc.).

In Eclipse gibt es bereits eine Vielzahl an vordefinierten Actions, denen auch bereits Tastenkombinationen zugewiesen sind. So können Einträge beispielsweise mit der „Entfernen“-Taste gelöscht oder mit der „F5“-Taste aktualisiert werden, sofern sie mit

[91] vgl.: IAction (Eclipse Plattform API Specification)

der entsprechenden Action verknüpft sind.[92] Eclipse stellt diese als sogenannte *RetargetActions* zur Verfügung. Eine RetargetAction ist eine abstrakte Hülle um eine konkrete Action und erhält bei der Instanziierung eine ID, die mit der einer konkreten Action übereinstimmt. Diese muss im aktiven Part des Workbenchs bekannt sein, damit sie verfügbar wird - ansonsten ist die Action deaktiviert. RetargetActions haben den Vorteil, dass für immer gleichbleibende Aufgaben, wie eben die bereits genannten, nicht jedesmal der gesamte Code erneut geschrieben werden muss - es ist lediglich eine konkrete Action mit der RetargetAction zu verknüpfen.[93] GDCF verwendet diese RetargetActions und verknüpft sie intern automatisch mit den vom Benutzer definierten Actions.

Die Definition erfolgt wieder in einer XML-Konfigurationsdatei, die von GDCF ausgelesen und übersetzt wird. Aus jeder definierten Action wird ein Modell erstellt, das eine Java-Repräsentation der XML-Konfiguration ist.

Im Folgenden werden die Basis-Actions von GDCF genauer erläutert sowie die Beispielkonfigurationen erklärt.

7.3.6.2 Die „open"-Action (open-modify-save)

Die *open*-Action ermöglicht dem Benutzer, NavigationItems in einem Editor zu öffnen und, falls erwünscht, zu bearbeiten und zu speichern. Sie stellt somit einen wesentlichen Bestandteil für die Applikation dar und benötigt höheren Implementierungs- und Konfigurationsaufwand.

[92] vgl.: ActionFactory (Eclipse Plattform API Specification)
[93] vgl.: RegargetAction (Eclipse Plattform API Specification)

Der Prozess des Öffnens, Editierens und Speicherns erfordert das Zusammenspiel mehrerer Komponenten, wie die folgende Grafik verdeutlicht:

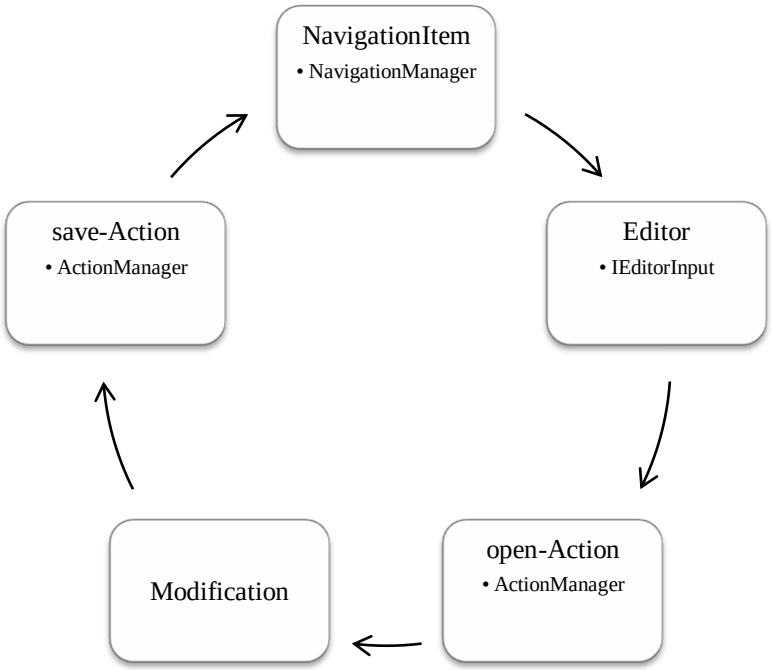

Abbildung 7.14 - Öffnen-Bearbeiten-Speichern Prozess

Zunächst muss der ausgewählte NavigationItem an den entsprechenden Editor übergeben werden. Dazu wird ein IGenericEditorInput-Objekt erzeugt (siehe 7.2.5) und an den Editor weitergereicht. Der Editor erzeugt anschließend über die Manager-Klasse eine entsprechende Action zur Anzeige der Daten und führt diese aus. Nachdem die Daten im Editor bearbeitet worden sind, muss der Editor eine weitere Action zum Zurückschreiben der Daten erzeugen. Nach erfolgreicher Ausführung müssen sich die geänderten Daten auch im NavigationView widerspiegeln. Hier spielt vor allem die Änderung des Namens eine Rolle, da der Name für die Anzeige im NavigationView verwendet wird.

Die Konfiguration einer open-Action für Companies könnte beispielsweise wie folgt aussehen:

```
<actionSet connectionId="databaseConnection">
     <action
           id="company.open"
           name="Open"
           nameIdentifier="Name"
           type="open"
           targetId="DatabaseRecordEditor"
           class="DatabaseAction"
           objectClass="Company"
           referenceId="open"
           icon="icons/building_go.png">
           <properties>
                 <property
                       key="sql"
                       value="select * from companies where ID=${id}"
/>
                 <property
                       key="updatesql"
                       value="update companies set
name=&quot;$_{Name}&quot; where ID=${id}" />
           </properties>
     </action>
</actionSet>
```

Das umschließende Element *actionSet* definiert die Hülle für eine beliebige Anzahl an Actions, die für dieselbe Connection definiert werden.

Unter dem *action*-Element wird die eigentliche Action mit den diversen Attributen festgelegt. Sie besitzt immer eine eindeutige und einmalige ID sowie einen Namen, der für die Anzeige im User Interface verwendet wird.

Das Attribut *type* legt den Typ der Action fest. Hier gibt es eine Reihe vordefinierter Werte wie „open“, „delete“, „rename“, etc. Dieses Attribut steuert auch die Handhabung

der Action in GDCF, da diese, wie bereits erwähnt, unterschiedliche Zyklen durchlaufen und auf die Ergebnisse unterschiedlich reagiert werden muss.

Das Attribut *targetId* definiert den aufzurufenden Editor und verweist auf einen eindeutigen Klassennamen.

Das *class*-Attribut legt fest, welche Action-Klasse für die Ausführung verwendet werden soll (siehe auch 7.3.5.6), wobei diese mit der im actionSet definierten Connection zusammenarbeiten muss.

Das Attribut *objectClass* ist für alle Actions relevant und definiert die Berechtigungen für NavigationItems. Hier muss der Klassenname der NavigationItems angegeben werden, für die diese Action gültig sein soll. Wird das Attribut nicht gesetzt, so gilt die Action für alle Elemente. Das bedeutet, dass die in dem Beispiel definierte Action nur für Companies anzuwenden ist, alle anderen Elemente besitzen eine solche Action nicht. Im NavigationView wird sie dann auch nur im Kontextmenü der Company-NavigationItems angezeigt.

Die *referenceId* gibt die zugehörige RetargetAction an, die für diesen Typ definiert ist.

Das *icon*-Attribut setzt das Icon für die Action fest, das im Kontextmenü, Menü oder Toolbar angezeigt werden soll. Wird kein Icon definiert, so erscheint lediglich der für das *name*-Attribut gesetzte Text.

Das Attribut *nameIdentifier* schließlich ist speziell für Actions vorgesehen, die im Laufe ihres Lebenszyklus eine Modifikation des Namens des NavigationItems vornehmen können. Wird der Name geändert, so müssen Editoren und NavigationViews diesen NavigationItem neu synchronisieren und den geänderten Namen entsprechend darstellen.

In Abschnitt 7.3.5.6 ist bereits das Erstellen von Actions beschrieben worden. Im Beispiel werden in der Action-Definition zwei SQL-Kommandos definiert. Eines zum Anzeigen und eines zum Speichern des NavigationItems. Beim ersten SQL-Kommando wird eine simple Select-Anweisung definiert, die nur einen Datensatz, nämlich den des

NavigationItems, zurückliefert und der vom Editor für die Anzeige verwendet werden kann. Das zweite SQL-Kommando wird für das Zurückschreiben des Datensatzes in die Datenquelle verwendet. Hier ist zu sehen, dass für das Update des Namens die Syntax „$_{Name}“ angegeben ist. Mit der Definition von *$_{Feldname}* ist es möglich, Daten aus Editoren zu extrahieren und in Anweisungen zu übersetzen. In diesem Fall würde das Feld „Name“ mit dem Wert des Editors überschrieben werden. Dieser muss beim Erzeugen der Action die Modifikation des Namens entsprechend berücksichtigen und dem ActionManager eine Feldliste mit Werten zur Verfügung stellen.

Nach erfolgreicher Definition der open-Action und dem Öffnen des Kontextmenüs eines Company-NavigationItems erscheint folgendes Menü:

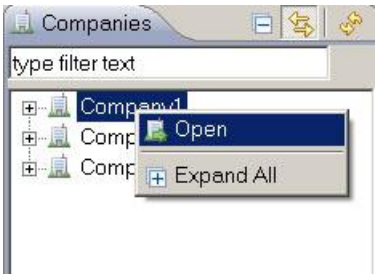

Abbildung 7.15 - open-Action 1

Beim Klicken auf den Eintrag wird der definierte Editor (der beispielhaft implementiere *DatabaseRecordEditor*) geöffnet und die open-Action ausgeführt.

Es ist auch möglich, die Elemente per Drag & Drop in den Editorbereich zu ziehen. Für jedes Element wird ein entsprechender Editor geöffnet, sofern eine open-Action vorhanden ist und ein Editor für dieses Element nicht bereits geöffnet ist. Wird auf den NavigationItem doppelgeklickt, so wird dieser ebenfalls im Editor angezeigt, es sei denn, er beinhaltet Subelemente. In diesem Fall wird der NavigationItem expandiert und die Subelemente werden angezeigt. Dies soll eine möglichst intuitive Bedienung ermöglichen.

Der Editorbereich stellt sich nach dem Öffnen der Elemente wie folgt dar:

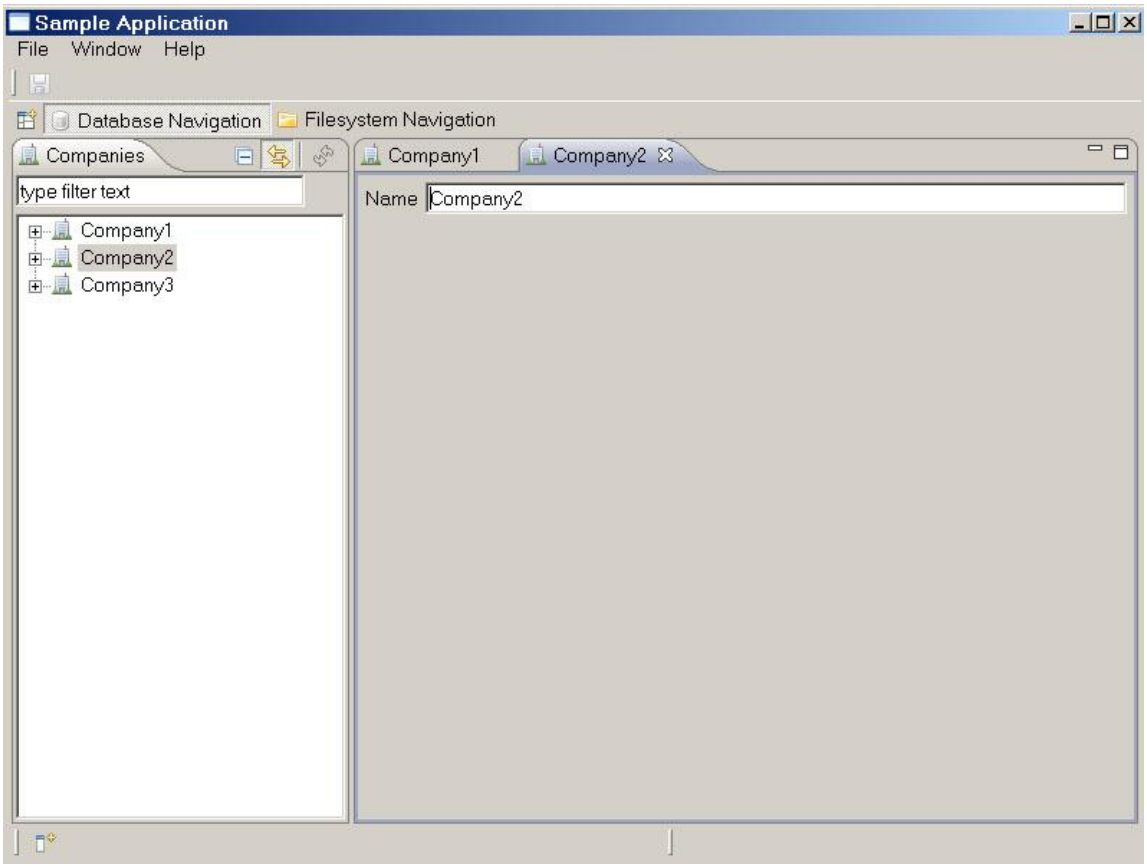

Abbildung 7.16 - open Action 2

Der DatabaseRecordEditor stellt die definierten Felder untereinander dar und bietet die Möglichkeit der Modifikation der Werte.

Bei Änderung des Namens und Klicken auf den Speichern-Button wird schließlich das zweite SQL-Kommando ausgeführt. Die Namensänderung erfordert anschließend eine Synchronisation des NavigationItems mit den Views und Editoren, in denen er angezeigt wird. So soll der Name im NavigationView sowie im Tab-Fenster des Editors geändert werden.

Diese Änderung zeigt die folgende Abbildung:

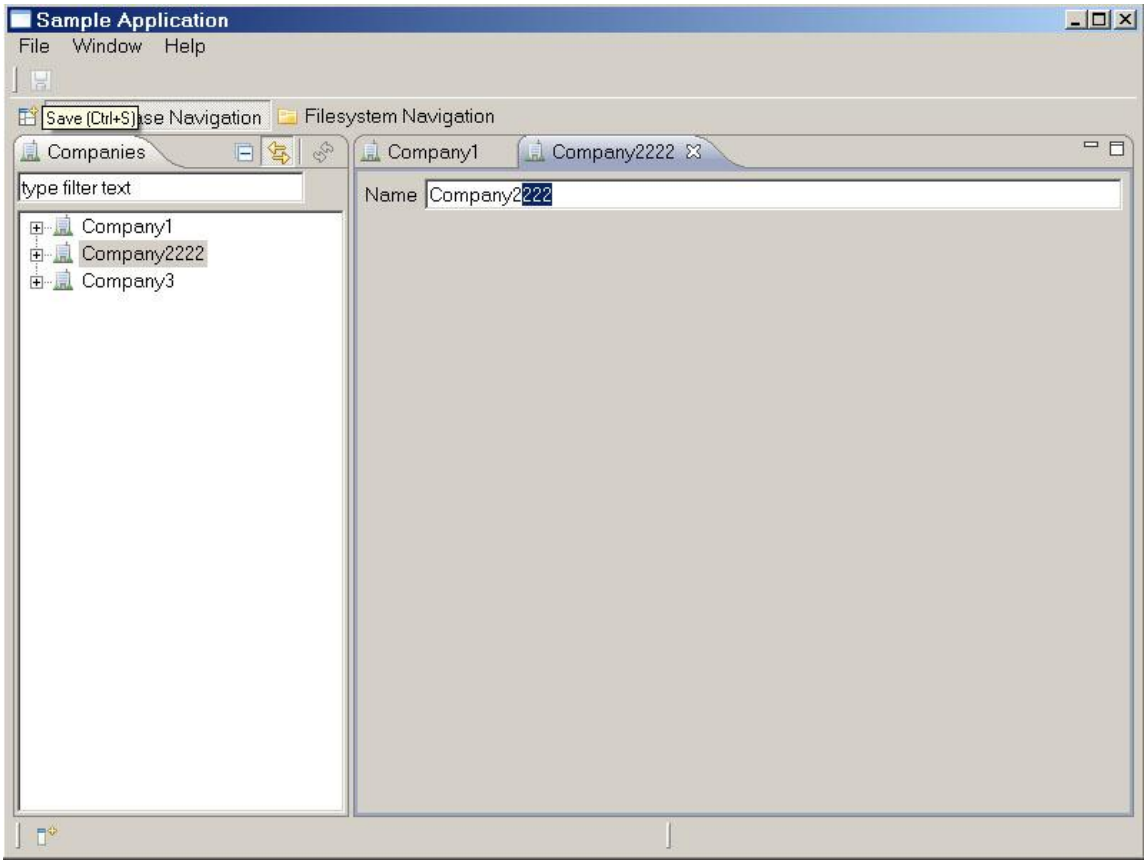

Abbildung 7.17 - open Action 3

7.3.6.3 Die „delete"-Action

Eine *delete*-Action dient zum Löschen eines NavigationItems. Wenn der Benutzer einen oder mehrere NavigationItems löschen will, so soll vorher ein Bestätigungsdialog erscheinen. Eine delete-Action ist als RetargetAction mit dem Shortcut „DEL" definiert. Nach Drücken dieser Taste wird die Action für den ausgewählten NavigationItem ausgeführt.

Die Konfiguration für das Beispiel könnte folgendermaßen aussehen:

```
<action
      id="company.delete"
      name="Delete"
      mode="update"
      type="delete"
      class="DatabaseAction"
      referenceId="delete"
      objectClass="Company"
      requireChildAction="true"
      showProgress="true"
      confirmation="true"
      confirmationText="Are you sure you want to delete the selected
company">
      <properties>
            <property
                  key="sql"
                  value="delete from companies where ID=${id}" />
      </properties>
</action>
```

Die bereits erläuterten Attribute sind hier ebenso anzuwenden wie die SQL-Kommandos. Zusätzliche Attribute sind *mode*, *requireChildAction*, *showProgress*, *confirmation* sowie *confirmationText*.

Das Attribut *mode* gibt an, um welche Art von Action es sich handelt. In GDCF sind *update* und *query* definiert. Für Datenbankverbindungen über JDBC beispielsweise müssen für Abfragen und Modifikationen der Daten unterschiedliche Methoden verwendet werden. Dieses Attribut legt fest, ob eine Modifikation der Daten zu erwarten ist, oder ob lediglich eine Abfrage erfolgt. Im Falle des Beispiels soll ein Eintrag gelöscht werden, was eine Modifikation der Daten darstellt, weshalb das Attribut auf „update“ gesetzt werden muss.

Das Attribut *requireChildAction* gibt an, ob die Action ausgeführt werden darf, wenn der NavigationItem Kinderelemente besitzt. Ist der Wert auf „true“ gesetzt, so werden alle Kinderelemente auf den Besitz einer Action mit demselben Typ hin untersucht. Besitzt eines der Kinderelemente keine delete-Action, so wird diese für den NavigationItem deaktiviert. Das hat den Grund, dass durch Löschen eines Eintrags eventuell Inkonsistenzen in der Datenquelle entstehen können. So würden beispielsweise alle Manager und Employees der Company plötzlich ohne Vaterelement existieren. Wenn alle Kinderelemente eine delete-Action besitzen, so wird für jedes einzelne Element diese Action ausgeführt. Das bedeutet, dass von der untersten bis zur obersten Ebene alle Elemente rekursiv entfernt werden. Dasselbe Prinzip gilt auch für das Löschen von Ordnern auf dem Dateisystem. Ein Ordner kann nicht gelöscht werden, wenn er Dateien enthält – es müssen zunächst alle Dateien und Subordner gelöscht werden, bevor der eigentliche Ordner entfernt werden kann.

Das *showProgress*-Attribut gibt an, ob beim Ausführen der Action ein Progress Bar mit Informationen zu den Elementen angezeigt werden soll. Gerade bei länger andauernden Actions möchte man den Benutzer eventuell über Fortschritt oder Status informieren. In Eclipse ist dies über das *Jobs API* möglich, welches das Abarbeiten von Aufgaben ermöglicht. Das Jobs API bietet diverse Möglichkeiten, Aufgaben in Warteschlangen zu organisieren und Abhängigkeiten zu definieren. Dabei gibt es sogenannte *System*- und *User-Jobs*, die sich vor allem durch ihre Darstellung unterscheiden. System-Jobs laufen im Hintergrund und geben dem Benutzer keine Rückmeldung oder Informationen zu den gerade ausgeführten Aufgaben. User-Jobs hingegen informieren den Benutzer mit Hilfe eines Fortschritt-Dialogs und eines speziellen *Progress-Views*.[94]

In GDCF werden Actions immer als Eclipse Jobs ausgeführt. Das hat den Vorteil, dass der Benutzer Informationen zu den Abläufen erhalten kann. Über das Attribut showProgress wird gesteuert, ob die Action als System- oder als User-Job ausgeführt werden soll.

[94] vgl.: The Eclipse Jobs API

Die beiden Attribute *confirmation* und *confirmationText* erlauben die Konfiguration von Dialogen zur Bestätigung von Actions. Das bedeutet, dass eine Action ausdrücklich bestätigt werden muss, bevor sie ausgeführt wird, wenn das confirmation-Attribut auf „true“ gesetzt wird. Der confirmationText ist optional und definiert den angezeigten Text im Bestätigungsdialog.

Die delete-Action stellt sich nach erfolgreicher Konfiguration in der Applikation wie folgt dar:

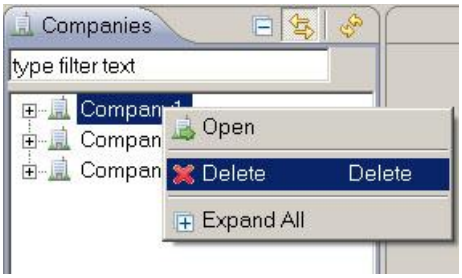

Abbildung 7.18 - delete Action 1

Nach dem Klicken auf die Schaltfläche erscheint der Dialog zur Bestätigung mit dem definierten Text.

Abbildung 7.19 - delete Action 2

Es ist auch möglich, mehrere NavigationItems zu löschen. Dazu müssen diese einfach selektiert und die Action ausgewählt werden. GDCF überprüft bei der Auswahl mehrerer NavigationItems auch die Berechtigungen zum Ausführen der Actions. So werden bei Auswahl mehrerer, unterschiedlicher NavigationItems nur jene Actions dargestellt, die alle explizit definieren. Dieser Ansatz ist bei den meisten Applikationen üblich, da sonst

Aktionen ausgeführt werden könnten, die von manchen selektieren Elementen nicht unterstützt werden, was eventuell in unerwartetem Verhalten der Applikation resultiert.

7.3.6.4 Die „rename"-Action

Eine *rename*-Action erlaubt das Umbenennen von NavigationItems und funktioniert im Prinzip wie die Änderung des Namens in einem Editor (siehe 7.3.6.2).

Die Konfiguration einer rename-Action könnte für das Beispiel folgendermaßen vorgenommen werden:

```
<action
      id="company.rename"
      name="Rename"
      nameIdentifier="Name"
      type="rename"
      mode="update"
      class="DatabaseAction"
      objectClass="Company"
      referenceId="rename"
      icon="icons/building.png">
      <properties>
            <property
                  key="sql"
                  value="update companies set
name=&quot;$_{Name}&quot; where ID=${id}" />
      </properties>
</action>
```

Die Konfiguration der SQL-Anweisung ist ident mit der Editor-Konfiguration. Lediglich die *type*-Deklaration sowie die *referenceId* auf die entsprechende Eclipse-RetargetAction sind unterschiedlich.

Nach Öffnen des Kontextmenüs und Auswählen der Action erhält der Benutzer einen Dialog zum Ändern des Namens.

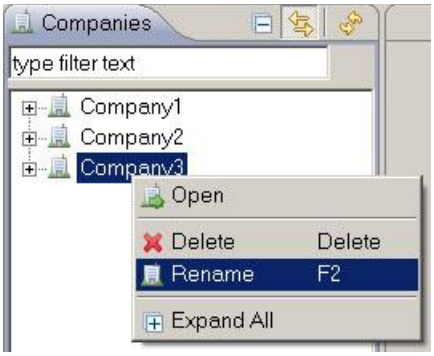

Abbildung 7.20 - rename Action 1

Abbildung 7.21 - rename Action 2

Die Änderung wird sofort auf die NavigationViews und Editoren übertragen.

7.3.6.5 Die „refresh"-Action

Eine *refresh*-Action dient zum Aktualisieren eines oder mehrerer NavigationItems. Dies ist beispielsweise dann erforderlich, wenn auf Änderungen in der Datenquelle reagiert werden soll.

```
<action
      id="company.refresh"
      name="Refresh"
      type="refresh"
      class="DatabaseAction"
      objectClass="Company"
      referenceId="refresh">
      <properties>
            <property
                  key="sql"
                  value="select * from companies where ID=${id}" />
      </properties>
</action>
```

Die Konfiguration gestaltet sich sehr einfach – für das Beispiel wird lediglich eine SQL-Select-Anweisung definiert. Durch Definition des *type*-Attributs führt GDCF automatisch eine Aktualisierung dieses NavigationItems durch. Als *referenceId* wird

ebenfalls „refresh“ angegeben, was Eclipse dazu veranlasst, dieser Action die Standard-Tastenkombination „F5“ für das Aktualisieren zuzuweisen.

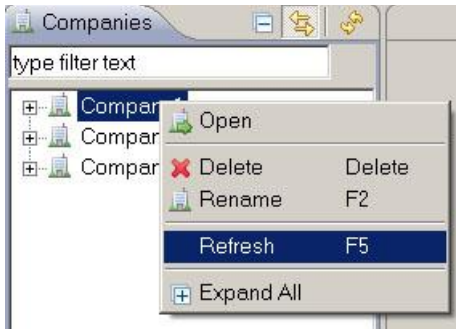

Abbildung 7.22 - refresh-Action 1

7.3.6.6 Die „new“-Action

Wenn der Benutzer ein neues Objekt anlegen können soll, so muss eine *new*-Action definiert werden. In Eclipse werden neue Objekte üblicherweise über einen sogenannten *Wizard* erzeugt. Ein Wizard ist ein Hilfs-Dialog, der den Benutzer durch einen Prozess leitet, vergleichbar mit einem Assistenten zum Installieren von Software.[95] In GDCF ist ein einfacher, beispielhaft implementierter Wizard vorhanden, mit dessen Hilfe neue Objekte auf generische Weise angelegt werden können.

Die Konfiguration einer new-Action könnte für das Anlegen eines Managers wie folgt aussehen:

```
<action
      id="manager.new"
      name="Manager"
      description="Add a new manager"
      type="new"
      mode="update"
      class="DatabaseAction"
      objectClass="Company"
      referenceId="new"
      icon="icons/user_suit.png">
      <properties>
```

[95] vgl.: IWizard (Eclipse Plattform API Specification)

```
            <property
                  key="sql"
                  value="insert into managers (firstname, surname,
companyId) values(&quot;${Firstname(default)}&quot;,
&quot;${Surname}&quot;, ${company.id})" />
      </properties>
</action>
```

Die einzelnen Bereiche der Konfiguration wurden bereits ausführlich behandelt, neu hinzu kommt das Attribut *description*. In einem Wizard können für jede Seite Beschreibungs-Texte eingefügt werden. Im Wizard der new-Action wird der hier spezifizierte Text auf der ersten Seite angezeigt.

Weiteres Augenmerk liegt auf der SQL-Anweisung. Hier werden zwei Feldnamen definiert, die beim Anlegen gesetzt werden sollen. Dabei ist beim Feldnamen „Firstname“ ein Standardwert eingetragen. Dieser wird mit der Syntax *${Feldname(Standardwert)}* gesetzt. Wenn der Benutzer beim Anlegen das Feld leer lässt, so wird dieser Standardwert aus der Konfiguration übernommen. In der SQL-Anweisung ist auch zu erkennen, dass die Company-ID auf den Wert der Company die gerade ausgewählt ist, gesetzt werden soll.

Neu anzulegende Objekte erscheinen in einem separaten Menü, wie die folgende Abbildung zeigt:

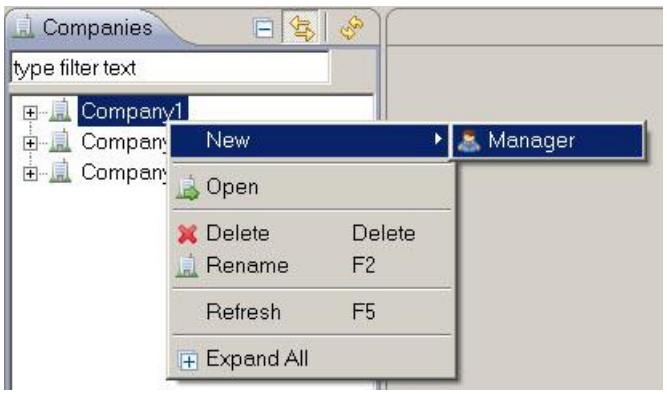

Abbildung 7.23 - new-Action 1

Nach Klicken auf die Schaltfläche „Manager“ öffnet sich der Wizard mit der konfigurierten Beschreibung sowie den zu setzenden Feldern.

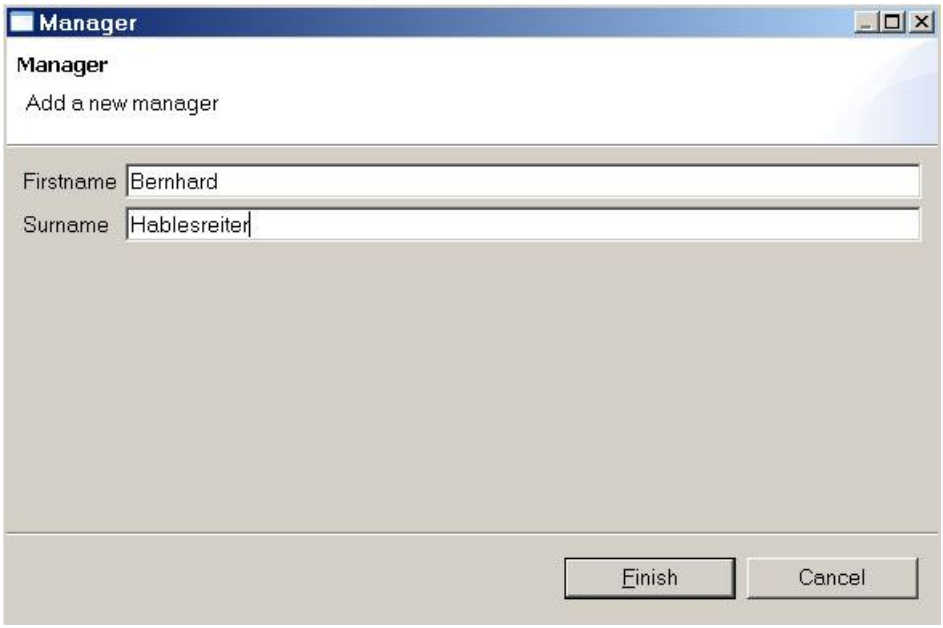

Abbildung 7.24 - new-Action 2

Der Wizard wird über den Button „Finish“ beendet und die Action wird mit den eingegebenen Daten ausgeführt. Das Ergebnis ist ein neu angelegter NavigationItem mit dem definierten Namen.

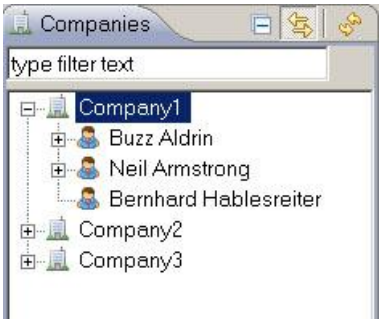

Abbildung 7.25 - new-Action 3

7.3.6.7 Definition von Actions über das Eclipse Plug-In-Development

Wie in Abschnitt 7.3.3 angesprochen, soll es auch möglich sein, neue Actions über die Eclipse-Werkzeuge des Plug-In-Developments erstellen zu können. Beispielhaft soll daher eine Action zum Anlegen von neuen Companies erstellt werden.

Die Konfiguration ist ähnlich wie in 7.3.6.6, definiert aber bei der referenceId eine spezielle Action-ID, die über die Eclipse-Schnittstellen eingebunden werden soll.

```
<action
      id="company.new"
      name="Company"
      description="Add a new company"
      type="new"
      mode="update"
      class="DatabaseAction"
      referenceId="newCompany"
      refreshFullNavigation="true"
      icon="icons/building_add.png">
      <properties>
            <property
                  key="sql"
                  value="insert into companies (name)
values(&quot;$_{Name}&quot;)" />
      </properties>
</action>
```

Die referenceId „newCompany“ zeigt auf die ID der zu erstellenden Action über die Eclipse-Werkzeuge. Die Konfiguration dieser Action muss in der *plugin.xml*-Datei erfolgen, die für die Client-Applikation angelegt worden ist.

```
<extension point="org.eclipse.ui.popupMenus">
      <viewerContribution id="viewerContribution"
            targetID="CompanyNavigationView">
             <action
                   class="org.eclipse.gdcf.core.action.GenericActionHa
ndler"
                  icon="icons/building_add.png"
                  id="newCompany"
                  label="Company"
                  menubarPath="new/newWizardShortlist">
            </action>
      </viewerContribution>
</extension>
```

Über den Extension-Point „popupMenus“ können per Plug-In-Konfiguration Menüeinträge in fremde Views eingebunden werden. Dies spezifiziert die targetID, welche hier auf den *CompanyNavigationView* zeigt. Bei der Action-Definition muss die Klasse angegeben werden, welche die Action später repräsentieren soll. Da alle Actions in GDCF generisch sind werden sie über den *GenericActionHandler* ausgeführt, der hier angegeben werden muss. Das *id*-Attribut muss denselben Wert wie in der Action-Konfiguration von GDCF haben, da die externe Action sonst nicht identifiziert werden kann. Das Attribut *menubarPath* gibt an, an welcher Position im Menü der Eintrag eingesetzt werden soll.

Ist die Konfiguration beendet, können neue Companies über die extern definierte Action angelegt werden.

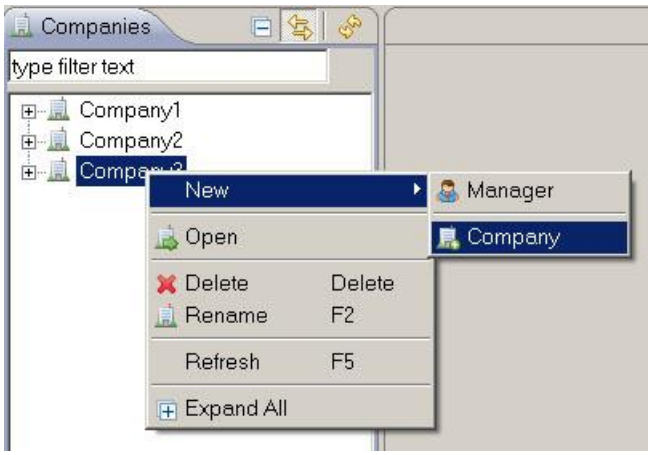

Abbildung 7.26 - new-Action über Plug-In-Development 1

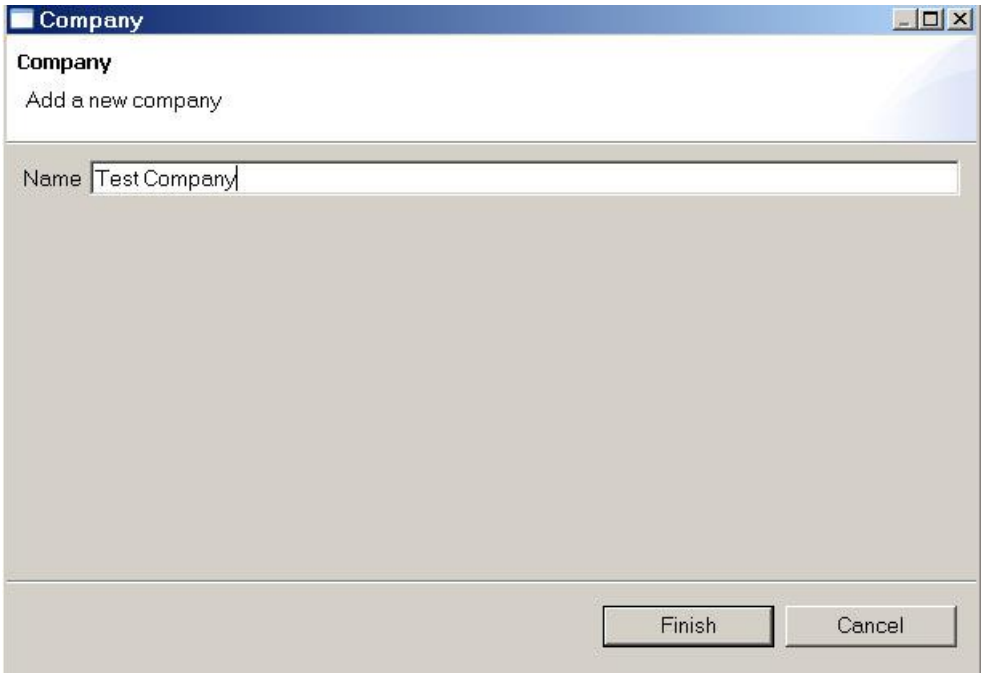

Abbildung 7.27 - new-Action über Plug-In-Development 2

Abbildung 7.28 - new-Action über Plug-In-Development 3

Da neu angelegte Elemente über die Connection-Schnittstelle geladen werden müssen, empfiehlt es sich, in der Konfiguration der Action das Attribut *refreshFullNavigation* auf „true“ zu setzen. Damit wird die Navigation aktualisiert und neu angelegte Elemente werden sofort angezeigt.

7.3.7 Zusätzliche Funktionen

Der Funktionsumfang der Views und Editoren kann durch die Erweiterung und Adaptierung der Basis-Implementierungen vergrößert werden. In der Regel werden Entwickler eigene Editoren programmieren, die auf die Anforderungen ihrer Applikationen zugeschnitten sind. Die basismäßig implementierten NavigationViews sind von der Funktionalität bereits recht ausgereift, können aber problemlos erweitert oder ersetzt werden. Die flexible Architektur von GDCF bietet dem Entwickler viel Freiraum bei der Umsetzung seiner Aufgaben.

NavigationViews bieten standardmäßig eine Refresh-Funktion, das Verlinken mit dem Editor sowie eine Filter- bzw. Suchfunktion.

Die Refresh-Funktion ist an den NavigationManager geknüpft und kann bei Bedarf die Navigation aktualisieren, um so auf Änderungen in der Datenquelle zu reagieren.

Abbildung 7.29 - NavigationView Refresh-Funktion

Die Funktion zum Verlinken des Editors mit dem NavigationView („Link with editor“) dient dazu, im Editor geöffnete NavigationItems im korrespondierenden View zu selektieren. Dies ist bei sehr vielen NavigationItems sinnvoll, da nach Klicken auf den entsprechenden Editor der NavigationItem automatisch selektiert wird. Dabei werden auch alle übergeordneten Elemente expandiert.

Abbildung 7.30 - NavigationView Editor-Verlinkung

Über die Filter-Funktion können Elemente im NavigationView schneller gefunden werden. Hier können Teile des Namens eingegeben werden, wodurch der NavigationView nur diese Elemente anzeigt. Wenn die gefundenen Objekte in einer

tieferen Hierarchie als der obersten platziert sind, so werden die Pfade zu diesen Elementen automatisch expandiert.

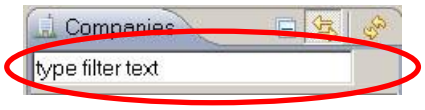

Abbildung 7.31 - NavigationView Filter-Funktion

7.4 Erfüllung der Anforderungen an GDCF

In Kapitel 3 wurden die Anforderungen an das Framework detailliert ausformuliert. Dabei wurden die Anforderungen auch im Laufe der Arbeit weiter ausgeleuchtet und deren Erfüllung bei der Beschreibung des Frameworks teilweise angesprochen. Dieser Abschnitt soll die Anforderungen noch einmal in Bezug auf das entwickelte Generic Data Control Framework behandeln und deren Erfüllung oder Nichterfüllung aufzeigen.

An das zu entwickelnde Framework wurden folgende Anforderungen erstellt, die im weiteren Verlauf genauer betrachtet werden:

- Verständlichkeit
- Bedienbarkeit
- Generalisierung
- Flexibilität
- Erweiterbarkeit
- Integrierbarkeit
- Performance

7.4.1 Verständlichkeit

In Abschnitt 3.1 wurde die Verständlichkeit in Bezug auf GDCF sowie die damit zu erstellenden Applikationen definiert.

7.4.1.1 Verständlichkeit der Software-Architektur und des Programmcodes

Die Architektur von GDCF ist bereits eingehend behandelt worden. Die einzelnen Aufgabenbereiche des Frameworks sind klar getrennt und die Funktionen mittels Interfaces eindeutig definiert. Die Basisimplementierungen zeigen dabei die Vorgehensweise bei der Entwicklung eigener Komponenten und Erweiterungen.

Der Programmcode des Frameworks lehnt sich an den Eclipse-Code-Style an und hält sich an die Eclipse-Namens-Konventionen. Desweiteren ist der gesamte Code mittels JavaDoc dokumentiert und jederzeit einsehbar. Dies ist besonders wichtig, um Erweiterungen oder eigene Implementierungen vorzunehmen. Die JavaDoc deckt dabei die Erläuterungen der Klassen sowie einzelnen Methoden ab und lehnt sich an die Eclipse-Dokumentation an, um Unklarheiten zu vermeiden und Einarbeitungszeiten gering zu halten.

7.4.1.2 Verständlichkeit der Client-Applikation

Bei der Verständlichkeit der Client-Applikation muss zwischen den grundlegenden Funktionen der Applikationen und den zusätzlichen, vom Entwickler erstellten, Funktionen unterschieden werden. Die in Abschnitt 7.3 beschriebenen Funktionen lehnen sich stark an die von Eclipse zur Verfügung gestellten an und sollen möglichst einfach verstanden werden. Da hier auf Standard-Aktionen wie Einfügen, Löschen, Aktualisieren, etc. gesetzt wird, kann davon ausgegangen werden, dass Benutzer der Applikation diese Funktionen verstehen und anwenden können. Die gesamte Oberfläche ist in Navigation und Bearbeitung der Elemente gegliedert, wie dies auch bei den meisten Webseiten und vielen Programmen der Fall ist. Wenn der Benutzer nicht ganz unerfahren ist, so kann das Verstehen der Oberfläche der Applikation ebenfalls angenommen werden.

Anders die individuell erstellen Funktionen – hier liegt die Verantwortung beim Entwickler, lediglich die Art und Weise der Einbettung der Funktionen kann von GDCF beeinflusst werden.

7.4.2 Bedienbarkeit

Die Bedienbarkeit ist ebenfalls in zwei Bereiche unterteilt worden, die Entwickler und Anwender betreffen.

7.4.2.1 Bedienbarkeit des Frameworks

Bei GDCF wird vollständig auf Standards und Open-Source, wie XML, Java und Eclipse gesetzt. Das erlaubt es, eventuell bereits vorhandenes Know-how des

Entwicklers in die Programmierung einfließen zu lassen. Er hat auch die Möglichkeit, durch die ausführlichen Dokumentationen der Basistechnologien seinen Wissenstand entsprechend zu erweitern. Da GDCF vollständig auf Eclipse basiert, können Client-Applikationen auch einfach in der Eclipse-Entwicklungsumgebung programmiert werden. Der Erstellungs-Prozess lässt sich somit weiter vereinfachen.

7.4.2.2 Bedienbarkeit der Client-Applikation

Die Bedienbarkeit der Client-Applikation ist bereits in Abschnitt 3.2.2 erläutert worden. Da die mit Hilfe von GDCF erstellen Applikationen auf Eclipse aufbauen und die Workbench-Architektur nutzen, lassen sich Teile der Bedienbarkeit auf diese abwälzen. Wie auch bereits in Abschnitt 7.4.1.2 angesprochen, bedient sich Eclipse der üblichen Fenster- und Funktionsanordnung mit Views, Editoren, Menüs, etc. Die Applikationen sind dabei selbsterklärend und weisen Ähnlichkeiten zu Standard-Software auf.

7.4.3 Generalisierung

Einer der Hauptanforderungen an das Framework wurde in Abschnitt 3.3 mit der Generalisierung formuliert. Dabei sollten alle datenbezogenen Abläufe möglichst generisch und abstrakt programmiert werden.

Wie bereits bei der Beschreibung der Architektur von GDCF in Abschnitt 7.2 erläutert worden ist, setzt das Framework auf eine vollständige Abstraktion aller daten- und anforderungsspezifischen Funktionen. Dies umfasst die Verwaltung und Erstellung von Verbindungen zu Datenquellen (*Connections*), Datenobjekten (*Navigations* und *NavigationItems*) sowie Aktionen auf diese Datenobjekte (*Actions*). GDCF definiert dazu eine Reihe von Interfaces, die vom Entwickler implementiert werden müssen. Die Funktionen sind dabei vollständig ausdefiniert und müssen nur entsprechend umgesetzt werden.

7.4.4 Flexibilität

Hier wurde definiert, dass das Framework in Bezug auf die anzubindenden Datenquellen sowie die Gestaltung der Benutzeroberfläche flexibel sein muss.

GDCF bietet dem Entwickler eine sehr abstrakte Sicht auf Datenquellen, Datenobjekte sowie Aktionen, wie auch in Abschnitt 3.3 definiert worden ist. Dabei hat der Entwickler einen großen Freiraum bei der Implementierung neuer Datenquellen, da in Bezug auf Transaktionen und Datenaustausch keinerlei Implementierungsvorgaben gemacht worden sind. Es obliegt somit dem Entwickler, den Datenaustausch abzuwickeln und die transferierten Daten zu verarbeiten.

Auch die Erstellung von Navigationen ist abstrakt gehalten und kann jederzeit durch Erweiterungen modifiziert werden. Dasselbe gilt für die Verwaltung und Erstellung von Aktionen. In GDCF sind alle Manager-Klassen als Interface gestaltet und besitzen eine Basis-Implementierung. Die Flexibilität ist somit insofern gegeben, dass der Entwickler entweder die Basis-Implementierungen verwenden, modifizieren oder durch eigene, anforderungsspezifische ersetzen kann.

Die Flexibilität der Gestaltung der Benutzeroberflächen ist auf die SWT-Komponenten sowie den Eclipse-Workbench beschränkt. Anforderungen, die darüberhinaus gehen können daher nicht erfüllt werden. Dies gilt auch für die Gestaltung der Navigations-Views, Editoren, Menüs und Toolbars. Hier können alle, von SWT und Eclipse zur Verfügung gestellten Komponenten verwendet werden. Durch die Ersetzbarkeit der GUI-Implementierungen von GDCF ist die Flexibilität bei der Gestaltung der Benutzeroberflächen jedenfalls gegeben.

7.4.5 Erweiterbarkeit

Die Erweiterbarkeit von GDCF ist in den letzen beiden Abschnitten bereits beschrieben worden. Hier spielen die Ersetzbarkeit der Basis-Implementierungen sowie die Definition der Interfaces entscheidende Rollen.

Desweiteren wurde in Abschnitt 7.3.6.7 beschrieben, wie Aktionen über die Eclipse-eigenen Werkzeuge in die Applikationen eingebunden werden können. Diese Vorgehensweise kann auch auf alle weiteren Möglichkeiten der Erweiterung und Anknüpfung, die das Eclipse-Framework bietet, angewandt werden.

Die Erweiterbarkeit von GCDF ist somit ebenfalls weitreichend sichergestellt.

7.4.6 Integrierbarkeit

Unter dem Punkt der Integrierbarkeit wurden Anforderungen auf Framework- sowie auf Applikations-Seite definiert.

7.4.6.1 Integrierbarkeit des Frameworks

Die Integrierbarkeit des Frameworks ergibt sich vor allem durch die Gestaltung als Eclipse-Plug-In. Damit kann das Plug-In in jeder Applikation genutzt werden und fügt sich nahtlos in die Eclipse-Architektur ein. Dadurch ist es auch möglich, Applikationen über die Eclipse-eigenen Werkzeuge weiterzuentwickeln.

7.4.6.2 Integrierbarkeit der Client-Applikation

Die Integrierbarkeit der Client-Applikation ist durch die Verwendung der GDCF-Module in anderen Applikationen gegeben. Diese können ebenfalls Teile von GDCF verwenden oder auf die Komponenten der erstellten Client-Applikationen, wie beispielsweise Navigations-Elemente, zugreifen.

7.4.7 Performance

Die Anforderungen an die Performance wurden in die Bereiche Skalierbarkeit, Speicherauslastung, Design und Coding sowie Logging aufgeteilt.

7.4.7.1 Skalierbarkeit

Bei der Entwicklung von GDCF wurde auf eine größtmögliche Skalierbarkeit der Applikationen geachtet, was vor allem für die Datenobjekte eine wichtige Rolle spielt. Das Management von Datenobjekten wurde dabei unter Verwendung geeigneter Datenstrukturen skalierbar und performant implementiert. Da jedoch Manager-Klassen sowie Datenquellen-Anbindungen abstrakt definiert worden sind und der Entwickler diese Implementierungen selbst vornehmen muss, obliegt ihm auch die performante Programmierung. Die Basis-Implementierungen können hier jedoch als Ansatzpunkt herangezogen werden.

7.4.7.2 Speicherauslastung

Für die Gewährleistung der geringen Speicherauslastung wurde bei der Entwicklung des Frameworks auf ein effizientes Speichermanagement gesetzt. So sind die angebundenen Datenobjekte als *NavigationItems* implementiert, welche sehr leichtgewichtig sind. Sie besitzen lediglich eine ID, einen Namen und zusätzliche Meta-Informationen. Das Ziel ist es, erst bei der Bearbeitung der Datenobjekte die vollständigen Daten zu laden. Damit kann der erforderliche Speicherplatz gering gehalten werden. Desweiteren wurde auf eine effiziente Behandlung und Entfernung von nicht mehr benötigten Daten gesetzt. Dies ist bei der Entwicklung von SWT-Anwendungen von besonderer Bedeutung, da die Speicherverwaltung teilweise selbst vorgenommen werden muss.[96]

Dasselbe gilt für die Verwendung von Icons. So werden für Aktionen und Navigations-Elemente keine konkreten Bilder, sondern sogenannte ImageDescriptor-Objekte verwendet. Diese stellen eine Referenz eines Bildes dar und werden nur bei Bedarf geladen.[97] Das ermöglicht eine effiziente Speicherausnutzung für alle Icon-Elemente, die im Framework eingesetzt werden.

7.4.7.3 Design und Coding

GDCF ist sehr modular aufgebaut und unter Verwendung objektorientierter Konzepte umgesetzt worden, was eine bessere Skalier- und Wartbarkeit des Codes ermöglicht.

Bei der Auswahl von Datenstrukturen wurde auf anforderungsspezifische und effiziente Strukturen gesetzt. So sind je nach Aufgabenbereich Listen, Maps und Container-Klassen eingesetzt worden, die für die jeweiligen Aufgaben geeignet sind.

7.4.7.4 Logging

GDCF bietet dem Benutzer das Logging-System *SLF4J* an. Dieses ist ein generisches Logging-System, das diverse Logging-Implementierungen, wie log4j oder Java Logging unterstützt und dabei sehr performant ist.[98]

---

[96] vgl.: SWT: Managing Operating System Resources
[97] vgl.: ImageDescriptor (Eclipse Plattform API Specification)
[98] vgl.: SLF4J FAQ

7.5 Einschränkungen von GDCF

Trotz der hohen Flexibilität, die GDCF dem Entwickler bietet, gibt es Einschränkungen bei der Verwendung des Frameworks. Die Erstellung von Client-Applikationen ist zunächst an die Limitationen der Eclipse Plattform sowie SWT gebunden. Das bedeutet, dass die von Eclipse und SWT zur Verfügung gestellten Mittel eingesetzt werden können - Anforderungen darüberhinaus können jedoch nicht erfüllt werden.

Eine weitere Einschränkung ist durch die Gestaltung der Schnittstellen von GDCF gegeben. Diese sind zwar sehr abstrakt definiert, beschränken die Anbindung von Datenquellen und Datenobjekten jedoch auf jene, die den Anforderungen des Frameworks entsprechen. So müssen die anzubindenden Datenquellen zumindest Daten liefern können, die von der Struktur her in eine Navigation übersetzt werden können. Dabei kann die Navigation hierarchisch aufgebaut werden und erlaubt die Repräsentation von 1:1- sowie 1:n-Beziehungen. Das bedeutet, dass n:n-Relationen in den Navigations-Views nicht abgebildet werden können. Hier müssten spezielle Views und Manager-Klassen programmiert werden, um diese Anforderungen zu erfüllen. In Navigations-Views kann auch immer nur jeweils eine Datenquelle aufgenommen werden. Wenn Daten von mehreren Datenquellen stammen, so muss für jede Datenquelle ein Navigations-View definiert werden.

Das Framework ist vor allem für kleine bis mittelgroße Applikationen erstellt worden und bietet auch diverse Erweiterungs- und Integrationsmöglichkeiten. Für große und komplexe Applikationen ist es jedoch weniger gut geeignet, da hier der Aufwand für die Anbindung von Datenquellen und Erstellung von Aktionen auf Datenobjekte, relativ zum Gesamtaufwand, in der Regel gering ist. Hier empfiehlt sich der Einsatz der Eclipse Plug-In-Development-Werkzeuge und die selbständige Anbindung der Datenquellen. Der Vorteil liegt in der flexibleren und anforderungsspezifischeren Definition der Schnittstellen sowie der Anzeige der Daten. GDCF kann aber auch bei größeren Applikationen, beispielsweise für einzelne Module, eingesetzt werden.

7.6 Technische Erläuterungen

In diesem Abschnitt werden diverse technische Aspekte von GDCF, wie Fehlerbehandlung, Logging, technische Anforderungen sowie Metriken erläutert.

7.6.1 Fehlerbehandlung

In GDCF gibt es grundsätzlich zwei Arten von Fehlern: Fehler in der internen Struktur, wie der Erstellung der Navigation oder der Aktionen, und Fehler aus den Datenquellen, die dem Benutzer angezeigt werden sollen. GDCF leitet alle Fehler zum einen an einen Logging-Mechanismus (siehe 7.6.2) und zum anderen an den Benutzer weiter, sofern es sich um einen gravierenden Fehler handelt, wie beispielsweise einen Verbindungsfehler.

Für Fehler, die von der Datenquelle ausgelöst und über eine Connection (siehe 7.2.1) übermittelt werden, verwendet GDCF denselben Mechanismus wie Eclipse. Das Result-Objekt (*IResult*) der Transaktion einer Connection beinhaltet auch die Eclipse Standardschnittelle für Status-Objekte (*IStatus*). Über dieses Status-Objekt können der Ausgang einer Transaktion wie beispielsweise Fehler, Warnungen, sowie weitere Informationen transportiert werden. GDCF knüpft an diese Status-Objekte an und präsentiert dem Benutzer die übermittelten Fehler bzw. Nachrichten in einem eigenen Dialog.

Die folgende Abbildung zeigt den Ausgang einer fehlgeschlagenen Aktion, die versucht, ein Vaterobjekt aus der Navigation der Datenbankverbindung, aus dem in Abschnitt 7.3 beschriebenen Beispiel, zu löschen. Dies ist aufgrund der Definition von Fremdschlüsseln in der Datenbank jedoch nicht möglich.

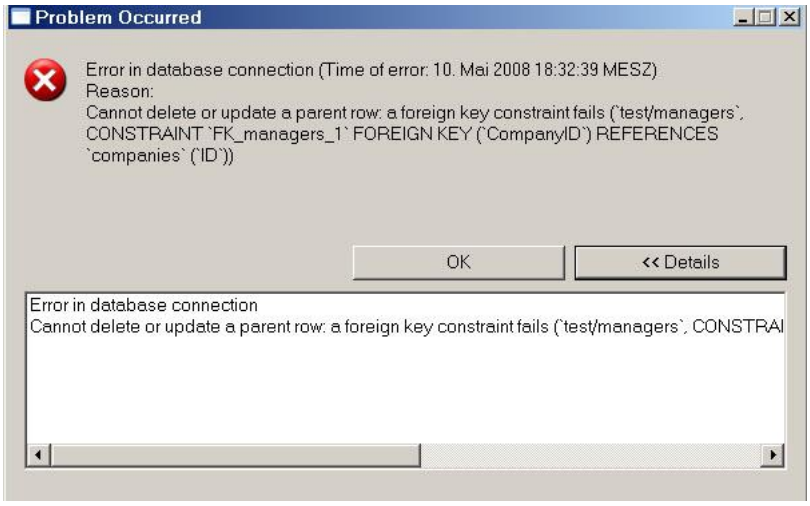

Abbildung 7.32 - Fehlerdarstellung in GDCF: Datenbankfehler

In diesem Beispiel wurde in der Connection-Klasse definiert, dass die von JDBC geworfene Exception direkt an den Benutzer weitergegeben wird. Hier wäre auch die Darstellung eines weniger detaillierten oder weniger technischen Fehlers vorstellbar. Die Verarbeitung der Fehler obliegt somit der Connection-Klasse und damit dem Entwickler.

7.6.2 Logging

Wie bereits erwähnt verfügt GDCF über den Logging-Mechanismus *SLF4J* (siehe 7.4.7.4). GDCF implementiert diesen Logging-Mechanismus in allen Manager-Klassen sowie in den Navigations-spezifischen- und Connection-Klassen. SLF4J ist lediglich eine Schnittstelle zu anderen Logging-Mechanismen und legt damit die Auswahl in die Hände des Entwicklers. Das hat den Vorteil, dass bereits vorhandene Logging-Mechanismen einfach in die Applikation eingebunden werden können. Das Logging selbst muss ebenfalls vom Entwickler vorgenommen werden und ist lediglich in den Basis-Implementierungen beispielhaft umgesetzt.

7.6.3 Anforderungen

GDCF wurde auf Java 5.0 und 6.0 sowie für Eclipse 3.3 (Europa) entwickelt. Das Framework kann auf allen Betriebssystemen eingesetzt werden, unter denen auch Eclipse funktioniert. Die Hardware-Anforderungen hängen stark von der Komplexität

und dem Umfang der erstellten Applikationen ab, da hier mehr oder weniger Festplatten- sowie Arbeitsspeicher und auch Prozessorgeschwindigkeit erforderlich sein können.

7.6.4 Umfang und Aufwand der Implementierung

Der Umfang und Aufwand der Implementierung wurde für GDCF mit Hilfe des Eclipse-Plug-Ins *Metrics*[99] in der Version 1.3.6 ermittelt. Dabei wurden Java typische Objekte sowie deren *Lines of Code* gezählt. Diese Lines of Code geben die Anzahl der Netto-Codezeilen ohne Kommentare, Leerzeilen und einfachen Klammern an.

Die folgende Tabelle zeigt die Anzahl der Java Objekte sowie die Gesamtanzahl der Codezeilen:

Einheit	Anzahl
Klassen	62
Interfaces	13
Methoden	387
Packages	19
Netto Lines of Code (LoC)	5080

Tabelle 7.2 – Metriken

Bei Gegenüberstellung der Aufwände von GDCF-Core-Komponenten und Beispiel-Implementierungen lässt sich die Intention von GDCF gut ablesen. Die Implementierungen der Core-Komponenten umfassen hauptsächlich die Abwicklung von Navigationen und Actions und beinhalten die Schnittstellen. Editoren, Connections und Results werden nur als Interface definiert. Die Beispiel-Implementierungen wiederum zeigen das genaue Gegenteil: Hier werden hauptsächlich Connections, Results und Editoren implementiert, da diese per Definition datenquellenabhängig sind und individuelle Implementierungen erfordern. Die Abläufe, die an die Datenquellen gebunden sind, sind jedoch Bestandteil der Core-Komponenten und auch dort implementiert, was sich in den Lines of Code widerspiegelt.

[99] Metrics

Die folgende Tabelle zeigt die Gegenüberstellung der Lines of Code von den Core-Komponenten und den Beispiel-Implementierungen:

Komponente	LoC Core-Implementierung	LoC Beispiel-Implementierung
Action	989	53
View	711	22
Navigation	645	65
Connection	124	294
Result	40	97
Editor	35	415
Dialog	385	-
Model	347	-
Config	280	-
EditorInput	133	-
Helpers	81	-
Perspective	-	28
Basis-Klassen	-	336
Summe	**3770**	**1310**

Tabelle 7.3 – Gegenüberstellung Lines of Code

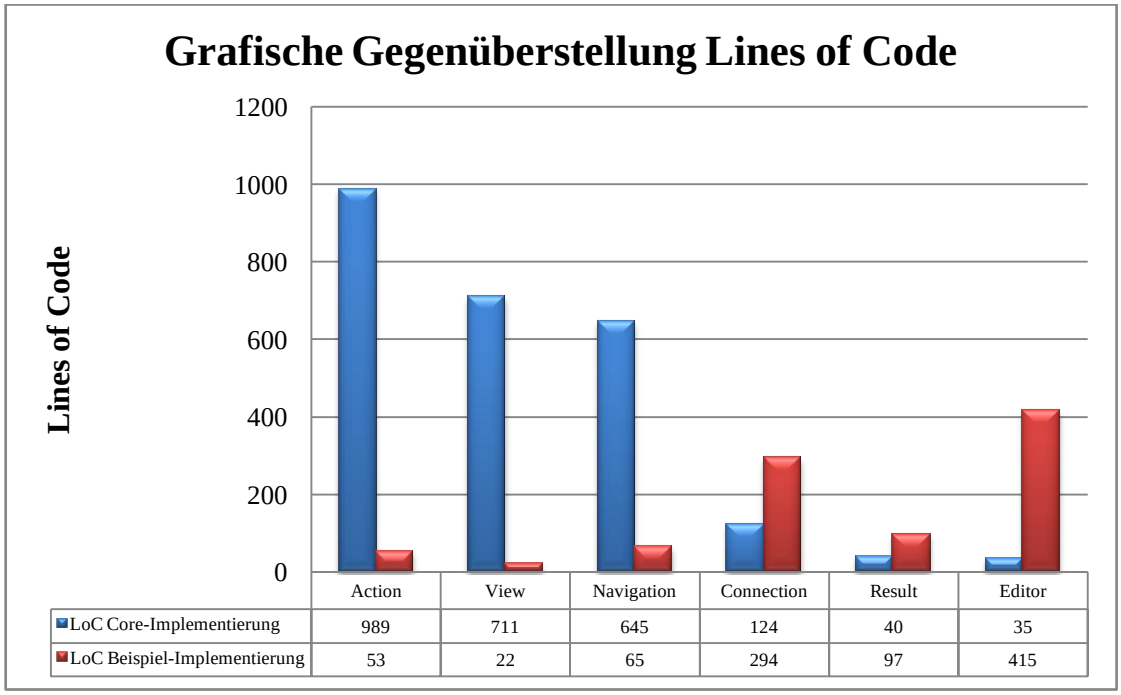

Abbildung 7.33 - Grafische Gegenüberstellung Lines of Code

In der folgenden Abbildung ist die Gegenüberstellung der gesamten Lines of Code zu sehen:

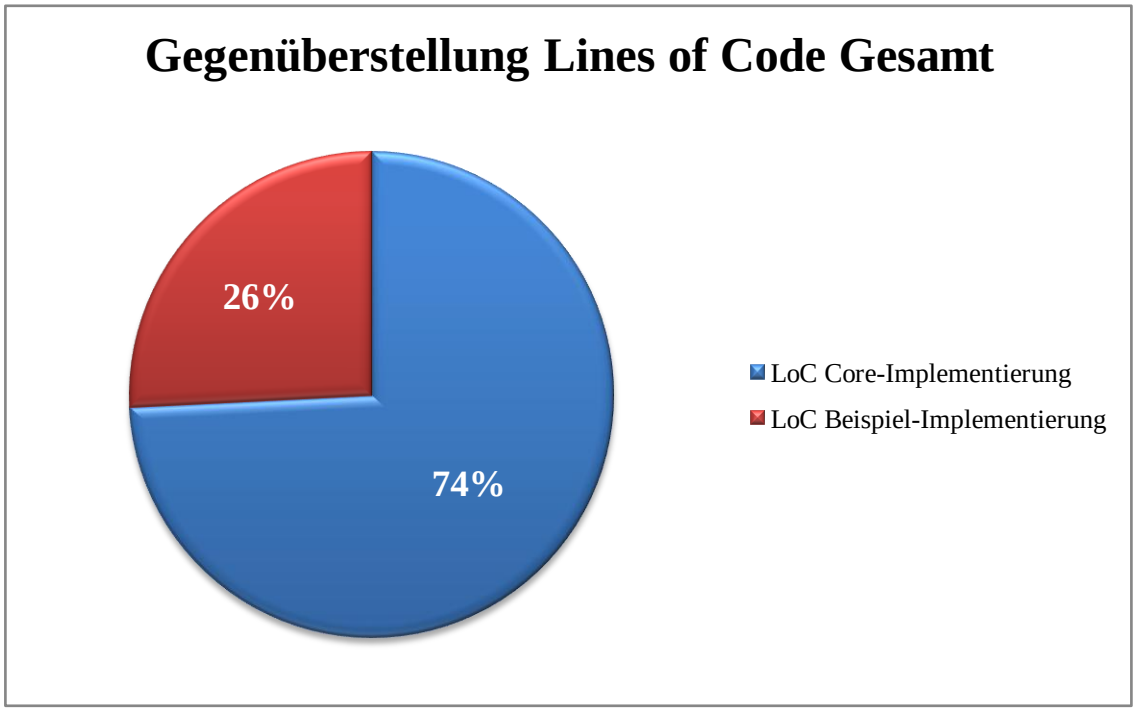

Abbildung 7.34 - Gegenüberstellung Lines of Code Gesamt

Hier muss angemerkt werden, dass die Beispiel-Implementierungen keinerlei Anspruch auf Vollständigkeit erheben. Sie sind zwar komplett und voll funktionsfähig, jedoch könnte eine Benutzer-Implementierung wesentlich aufwändiger sein und mehr Codezeilen benötigen. Mit diesen Beispiel-Implementierungen ist es aber durchaus möglich, ansprechende Applikationen ohne großen Programmieraufwand zu erstellen. Zudem sind die zu implementierenden Funktionen genau vorgegeben und müssen nicht neu konzipiert werden. Sie müssen lediglich für die entsprechenden Anforderungen ausprogrammiert werden, was den Entwicklungsaufwand deutlich reduziert.

8 Zusammenfassung

Wie in der Arbeit ausführlich gezeigt worden ist, erfordert die Erstellung von Client-Applikationen unter Eclipse in der Regel viel Know-how und Entwicklungszeit. GDCF ist in der Lage, dem Entwickler einen Großteil der Entwicklungszeit abzunehmen, da lediglich die Benutzerschnittstellen angepasst sowie die Datenanbindungen vorgenommen werden müssen. Der Entwicklungsaufwand kann sich im besten Fall auf die Definition der XML-Dateien beschränken, ohne dass dabei eine Zeile Java-Code geschrieben werden muss. Diese Anforderung war auch ein wichtiger Aspekt beim Design des Frameworks. Ziel war es vor allem, die Entwicklungszeit zu verkürzen und dem Entwickler für einfache Applikationen einen schnellen Lösungsweg zu bieten.

Es wurde auch gezeigt, dass es diverse Möglichkeiten der Erweiterung und Adaptierung von GDCF gibt, um Applikationen an spezielle Anforderungen anzupassen. Desweiteren wurde verdeutlicht, dass es einfach möglich ist, die Applikationen mit Eclipse-eigenen Werkzeugen (Plug-In Development-Tools) weiterzuentwickeln oder daran anzuknüpfen.

Mit diesen Erfüllungen der Anforderungen wurde die erste These, nämlich die Möglichkeit der einfachen Erstellung von Client-Applikationen unter Eclipse mit Anbindung beliebiger Datenquellen, validiert.

Die zweite These richtete sich auf die einfache Erlernbarkeit des Eclipse Frameworks mit Hilfe der in Kapitel 4 definierten Methoden. Unter Berücksichtigung der in Abschnitt 4.1.2 festgelegten Eingangsvoraussetzungen sind die Komponenten dieser Arbeit mit Bedienungsteil, Code-Dokumentation und Beispiel-Applikation geeignet, erfahrenen Java-Entwicklern die in Abschnitt 4.2 formulierten Inhalte zu vermitteln. Durch die Sammlung gängiger Funktionen und Vorgehensweisen bieten GDCF und diese Arbeit ein Entwickler-Werkzeug zur einfachen Erlernung des Eclipse-Frameworks beziehungsweise der Client-Applikations-Entwicklung unter Eclipse, womit auch die zweite These bewiesen ist.

Zusammenfassend lässt sich feststellen, dass es möglich ist, auf einfache Art und Weise Client-Applikationen auf Basis von Eclipse zu erstellen, ohne dabei zu tief in das

komplexe Thema einsehen zu müssen oder entsprechendes Know-how zu besitzen. Darüberhinaus ist gezeigt worden, dass das entwickelte Framework geeignet ist, die Eclipse Plattform und Client-Entwicklung besser und einfacher verstehen, und die implementierten Funktionen in anderen Softwareprojekten einsetzten zu können.

9 Literaturverzeichnis

9.1 Offline-Quellen

- R. Bull, C. Best, M. Storey: Advanced widgets for Eclipse, ACM, Vancouver, Eclipse Technology Exchange (ETX) 2004, Seiten: 6-11
- D. Hunter, A. Watt, J. Rafter, J. Duckett, D. Ayers, N. Chase, J. Fawcett, T. Gaven, B. Patterson: *Beginning XML*, 3rd Edition, Wiley Publishing, Inc, Indianapolis, 2004
- M. Elder: Common Navigator Framework Presentation, IBM Rational Software, Durham, USA, 2006
- K. Brüssau, O. Widder, H. Brückner, M. Lippert, M. Lübken, B. Schwartz-Reinken, L. Wunderlich: *Eclipse – Die Plattform*, entwickler.press, Frankfurt, 2006
- A. Wolfe: Eclipse: A Platform Becomes an Open-Source Woodstock, ACM, Vancouver, ACM Queue, Ausgabe 8, November 2003, Seiten: 14-16
- D. Geer: Eclipse becomes the dominant Java IDE, IEEE Computer, Ausgabe 7, Juli 2005, Seiten: 16-18
- K. Ohgata: *Jalcedo Presentation*, NEC Soft Ltd., Tokyo, 2007
- S. Wilson, J. Kesselman: *Java Platform Performance - Strategies and Tactics*, Addison-Wesley Longman Publishing Co., Inc., Boston, USA, 2000
- N. Meyers: *Java Programming on Linux*, Waite Group Press, Indianapolis, 2000
- M. Scarpino, S. Holder, S. Ng, L. Mihalkovic: *SWT/JFace in Action*, Manning Publications Co., Greenwich, 2005

9.2 Online-Quellen

- *A brief history of Eclipse*
  http://www.ibm.com/developerworks/rational/library/nov05/cernosek/
  abgerufen am 22.05.2008
- *About the Eclipse Foundation*
  http://www.eclipse.org/org
  abgerufen am 22.05.2008
- *ActionFactory (Eclipse Platform API Specification)*
  http://help.eclipse.org/help32/index.jsp?topic=/org.eclipse.platform.doc.isv/reference/api/org/eclipse/ui/actions/ActionFactory.html
  abgerufen am 22.05.2008
- *Common Navigator Framework for Platform/UI in 3.2*
  http://www.eclipsecon.org/2006/Sub.do?id=260
  abgerufen am 22.05.2008
- *Common Navigator Tutorial 1*
  http://rcpquickstart.wordpress.com/2007/04/25/common-navigator-tutorial-1-hello-world
  abgerufen am 22.05.2008
- *Eclipse Download Overview*
  http://www.eclipse.org/downloads
  abgerufen am 22.05.2008
- *Eclipse Naming Conventions*
  http://wiki.eclipse.org/index.php/Naming_Conventions
  abgerufen am 22.05.2008
- *Eclipse Platform Technical Overview*
  http://www.eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.html
  abgerufen am 22.05.2008

- *Eclipse User Interface Guidelines*
  http://wiki.eclipse.org/User_Interface_Guidelines
  abgerufen am 22.05.2008
- *Eclipse.org*
  http://www.eclipse.org
  abgerufen am 22.05.2008
- *GUI Design - Linktipps zu Usability*
  http://www.gui-design.de/link-tipps.htm
  abgerufen am 22.05.2008
- *IAction (Eclipse Platform API Specification)*
  http://help.eclipse.org/help32/index.jsp?topic=/org.eclipse.platform.doc.isv/reference/api/org/eclipse/jface/action/IAction.html
  abgerufen am 22.05.2008
- *IBM Design principles*
  https://www-306.ibm.com/software/ucd/designconcepts/designbasics.html
  abgerufen am 22.05.2008
- *IEditorInput (Eclipse Platform API Specification)*
  http://help.eclipse.org/help32/index.jsp?topic=/org.eclipse.platform.doc.isv/reference/api/org/eclipse/ui/IEditorInput.html
  abgerufen am 22.05.2008
- *ImageDescriptor (Eclipse Platform API Specification)*
  http://help.eclipse.org/help33/index.jsp?topic=/org.eclipse.platform.doc.isv/reference/api/org/eclipse/jface/resource/ImageDescriptor.html
  abgerufen am 22.05.2008
- *IStatus (Eclipse Platform API Specification)*
  http://help.eclipse.org/help32/index.jsp?topic=/org.eclipse.platform.doc.isv/reference/api/org/eclipse/core/runtime/IStatus.html
  abgerufen am 22.05.2008

- *IWizard (Eclipse Platform API Specification)*
  http://help.eclipse.org/help31/index.jsp?topic=/org.eclipse.platform.doc.isv/reference/api/org/eclipse/jface/wizard/package-summary.html
  abgerufen am 22.05.2008
- *Java 6 Standard Edition*
  http://java.sun.com/javase/6
  abgerufen am 22.05.2008
- *Java Enterprise Edition – Technologies*
  http://java.sun.com/javaee/technologies
  abgerufen am 22.05.2008
- *Java Interfaces*
  http://java.sun.com/docs/books/tutorial/java/concepts/interface.html
  abgerufen am 22.05.2008
- *Java Native Interface*
  http://java.sun.com/j2se/1.4.2/docs/guide/jni/spec/intro.html
  abgerufen am 22.05.2008
- *Java Persistence API*
  http://java.sun.com/javaee/technologies/persistence.jsp
  abgerufen am 22.05.2008
- *Java Standard Edition – JDBC*
  http://java.sun.com/javase/6/docs/technotes/guides/jdbc/index.html
  abgerufen am 22.05.2008
- *Java Standard Edition – Logging*
  http://java.sun.com/javase/6/docs/technotes/guides/logging/index.html
  abgerufen am 22.05.2008
- *Java Standard Edition – Math Functionality*
  http://java.sun.com/javase/6/docs/technotes/guides/math/index.html
  abgerufen am 22.05.2008

- *Java Standard Edition – Package Summary*
  http://java.sun.com/javase/6/docs/api/java/util/zip/package-summary.html
  abgerufen am 22.05.2008
- *Java Standard Edition – Technologies*
  http://java.sun.com/javase/technologies/index.jsp
  abgerufen am 22.05.2008
- *JavaDoc Tool*
  http://java.sun.com/j2se/javadoc
  abgerufen am 22.05.2008
- *Metrics*
  http://metrics.sourceforge.net
  abgerufen am 22.05.2008
- *Office (2007) Open XML-Dateiformate*
  http://msdn2.microsoft.com/de-de/library/aa338205.aspx
  abgerufen am 22.05.2008
- *Rapid developing a Rich Client/Server Application based RCP (JALCEDO)*
  http://www.eclipsecon.org/2007/index.php?page=sub/&id=4043
  abgerufen am 22.05.2008
- *RegargetAction (Eclipse Platform API Specification)*
  http://help.eclipse.org/help32/index.jsp?topic=/org.eclipse.platform.doc.isv/reference/api/org/eclipse/ui/actions/RetargetAction.html
  abgerufen am 22.05.2008
- *SLF4J FAQ*
  http://www.slf4j.org/faq.html
  abgerufen am 22.05.2008
- *Sun Products*
  http://developers.sun.com/products
  abgerufen am 22.05.2008

- *SWT, Swing or AWT: Which is right for you?*
  http://www.ibm.com/developerworks/library/os-swingswt/index.html
  abgerufen am 22.05.2008
- *SWT: Managing Operating System Resources*
  http://www.eclipse.org/articles/swt-design-2/swt-design-2.html
  abgerufen am 22.05.2008
- *The Eclipse Jobs API*
  http://www.eclipse.org/articles/Article-Concurrency/jobs-api.html
  abgerufen am 22.05.2008
- *W3C – Extensible Markup Language (XML)*
  http://www.w3.org/XML
  abgerufen am 22.05.2008
- *W3C Soap*
  http://www.w3.org/TR/soap
  abgerufen am 22.05.2008
- *XML in the Java Platform Standard Edition*
  http://java.sun.com/javase/6/docs/technotes/guides/xml/index.html
  abgerufen am 22.05.2008

10 Abbildungsverzeichnis

11 Tabellenverzeichnis