Thomas Richter

Konzeption und prototypische Implementierung eines Ansatzes zur Anbindung eines graphischen Benutzerinterfaces an eine bestehende Geschäftsanwendung

eingereicht als

DIPLOMARBEIT

an der

HOCHSCHULE MITTWEIDA

UNIVERSITY OF APPLIED SCIENCES

Fachbereich Mathematik Physik Informatik

Mittweida, 2010

Erstprüfer: Prof. Dr. Wilfried Schubert

Zweitprüfer: Dipl. Inf. Andreas Mehner

Vorgelegte Arbeit wurde verteidigt am:

Bibliographische Beschreibung:

Richter, Thomas:
Konzeption und prototypische Implementierung eines Ansatzes zur Anbindung eines graphischen Benutzerinterfaces an eine bestehende Geschäftsanwendung. 2010. – 110 S. Mittweida, Hochschule Mittweida, Fachbereich Mathematik, Physik, Informatik Diplomarbeit, 2010

Referat:

Diese Diplomarbeit gibt Ausblicke und Lösungen zur Anbindung eines graphischen Benutzerinterfaces an eine in Sybase PowerBuilder entwickelte bestehende Geschäftsanwendung, für eine schrittweise Überführung eines Clientprogrammes in eine zukunftsorientierte Anwendung. Dabei werden zunächst theoretische Grundlagen ermittelt sowie die im bestehenden Programm eingesetzten Technologien analysiert. Anschließend werden mögliche Lösungen für das Problem erarbeitet und abgewogen. Weiterhin wird eine konkrete Variante im Konzept ausgearbeitet und als Prototyp umgesetzt, gefolgt von einer Überlegung zur weiteren schrittweisen Vorgehensweise bei der sukzessiven Überführung der einzelnen Komponenten. Abschließend erfolgen eine Auswertung der Arbeit und der Ausblick auf zukünftige Erweiterungen sowie Möglichkeiten.

Inhaltsverzeichnis

	Inhaltsverzeichnis	**I**
	Abbildungsverzeichnis	**IV**
	Tabellenverzeichnis	**V**
	Abkürzungsverzeichnis	**V**
	Vorwort	1
Kapitel 1	**Einleitung**	**2**
1.1.	Problemstellung	3
1.2.	Zielstellung	5
1.3.	Abgrenzung	7
1.4.	Vorgehensweise	8
Kapitel 2	**Theoretische Grundlagen**	**9**
2.1.	Softwarealterung	10
2.2.	Softwareergonomie	11
2.3.	Usability	12
2.4.	Software Redesign	13
2.5.	Software Sanierung	14
2.6.	4GL - Fourth Generation Language	14
2.7.	Prototyping	15
Kapitel 3	**Ist-Analyse und angewandte Technologien**	**17**
3.1.	Die Leasingsoftware Leasman	18
3.1.1.	Sybase PowerBuilder	18
3.1.2.	Anzahl und Verteilung der Klassen im Leasman	20
3.1.3.	Vererbungshierarchie der Fenster	21
3.1.4.	Benutzerabhängige Oberfläche	22
3.1.5.	Modularer Aufbau	23
3.2.	Business Logic Server (BLS)	25
3.2.1.	Webservice	27
3.2.2.	Spring Framework	27
3.2.3.	Hibernate	28

Kapitel 4	**Geeignete Technologien für den Prototyp**	**30**
4.1.	Webbasierende oder eigenständige Anwendung	31
4.2.	Wahl der Programmiersprache	32
4.2.1.	Die .NET Sprache C#	33
4.2.2.	JavaFX	34
4.2.3.	Java Swing	34
4.2.4.	Standard Widget Toolkit (SWT)	35
4.2.5.	Die Programmiersprachen im Vergleich	36
4.3.	Rich Client Anwendung	38
4.3.1.	NetBeans RCP	39
4.3.2.	Eclipse RCP	40
4.3.3.	Die Frameworks im Vergleich	40
Kapitel 5	**Konzeption des Prototyps**	**42**
5.1.	Eigenschaften von Eclipse RCP	43
5.1.1.	Eclipse Equinox OSGi	43
5.1.2.	Plugins	43
5.1.3.	Classloading	44
5.1.4.	Extension Points	45
5.1.5.	Actions	45
5.1.6.	Die Workbench	46
5.2.	Verwendung der Workbenchkomponenten	48
5.3.	Der Actionserver	49
5.3.1.	Timer gesteuertes Polling	50
5.3.2.	Shared Objects	51
5.3.3.	Aufbau auf PowerBuilder-Seite	51
5.3.4.	Aufbau auf Java-Seite	54
5.3.5.	Funktionsweise des Actionservers	56
5.4.	Der GUIManager	63
5.4.1.	Aufbau des GUIManagers	64
5.4.2.	Funktionsweise des GUIManagers	66
5.5.	Zentralisierung über Lifecycle	68

Kapitel 6 Prototypische Implementierung 69
6.1. Implementierung des Actionservers 70
6.1.1. Die Thread Klassen 70
6.1.2. Die Runnable Klassen 72
6.1.3. Die Actionserver Klasse 74
6.1.4. Probleme bei der Implementierung 76
6.2. Implementierung des GUIManagers 77
6.2.1. Das WindowUtil 78
6.2.2. Die WorkbenchFactory 79
6.2.3. Der BasicEditor 80
6.2.4. Der BasicControlListener 83
6.3. Die Starter Klasse 84
6.4. Java Klassen im Leasman laden 85
6.5. Probleme bei der Implementierung 86
6.6. Testen des Prototyps 87

Kapitel 7 Vorgehen bei der weiteren Überführung 88
7.1. Allgemeines Vorgehen 89
7.2. Migrationsreihenfolge 90
7.3. Ergonomie und Redesign der Benutzeroberfläche 91
7.4. Test der überführten Komponenten 92

Kapitel 8 Schlussbetrachtung 93
8.1. Fazit 94
8.2. Ausblick 96

Anlagen 97
Literaturverzeichnis 98
Quellenverzeichnis 100
Selbständigkeitserklärung 103

Abbildungsverzeichnis

Abbildung 1: Aktueller Aufbau des Leasman mit Anbindung an BLS 4
Abbildung 2: Gewünschter Zustand nach Einbindung des Prototyps 6
Abbildung 3: Lebenszyklus eines Softwareproduktes [ebc] 11
Abbildung 4: Kommunikation zwischen PowerBuilder und Java 19
Abbildung 5: Verteilung der verschiedenen Klassen des Leasman 20
Abbildung 6: Vererbungshierarchie der Fenster 22
Abbildung 7: Menüstruktur Leasman 24
Abbildung 8: 3-Schicht Architektur Leasman mit Einbindung BLS [Mehner08] 26
Abbildung 9: Classloading der Plugins [OSGi09, S. 37] 44
Abbildung 10: Workbench der Entwicklungsumgebung Eclipse 46
Abbildung 11: Komponenten der bisherigen GUI des Leasman 48
Abbildung 12: Kommunikationsmodell Actionserver 50
Abbildung 13: Klassendiagramm Actionserver PowerBuilder 52
Abbildung 14: Klassendiagramm PBActionManager Java-Seite 54
Abbildung 15: Sequenzdiagramm Initialisierung Actionserverthread 57
Abbildung 16: Sequenzdiagramm Ausführung einer Action im Actionserver 59
Abbildung 17: Sequenzdiagramm Beenden des Actionservers 62
Abbildung 18: Klassendiagramm Aufbau GUIManager 64
Abbildung 19: Öffnen eines Childfensters im PB und Integrierung in RCP Editor 67
Abbildung 20: Programmausschnitt *of_set_runnable() ucc_cu_thread* 71
Abbildung 21: Programmausschnitt *of_set_runnble()* in *ucc_application_thread* 72
Abbildung 22: Programmausschnitt Actionserverloop PowerBuilder 73
Abbildung 23: Ausführbare Actions im Actionserver 74
Abbildung 24: Methode *of_execute_action()* der Klasse *ucc_ac_action_server* 76
Abbildung 25: Funktion *wf_opensheet()* in *w_mdi_applikation* 77
Abbildung 26: Java Klasse *WindowUtil* 78
Abbildung 27: Klasse *WorkbenchFactory openNewEditor()* 79
Abbildung 28: Methode *createPartControl()* im *BasicEditor* 81
Abbildung 29: Methode *doSave()* im *BasicEditor* 82
Abbildung 30: Funktion *resizeWindow()* im *BasicControlListener* 83
Abbildung 31: Starter Klasse 84
Abbildung 32: Zukünftige Umsetzungen der Geschäftslogik im BLS [Mehner08] 90
Abbildung 33: LeasmanWorkbench v0.7 95

Tabellenverzeichnis

Tabelle 1: Entscheidungstabelle möglicher Programmiersprachen 37
Tabelle 2: Entscheidungstabelle NetBeans RCP, Eclipse RCP 41

Abkürzungsverzeichnis

BLS	Business Logic Server der DELTA proveris AG
EDV	Elektronische Datenverarbeitung
JDBC	Java Datenbank Schnittstelle
SQL	Structured Query Language
JVM	Virtuelle Maschine von Java
XML	Extensible Markup Language
EJB	Enterprise Java Beans
GUI	Graphical User Interface
API	Application Programming Interface
4GL	Fourth Generation Language
AG	Aktiengesellschaft
C#	C-Sharp
URI	Uniform Resource Identifier
POJO	Plain Old Java Object
HQL	Hibernate Query Language
AWT	Abstract Window Toolkit
JNI	Java Native Interface
PBNI	PowerBuilder Native Interface
SWT	Standard Widget Toolkit
RMI	Remote Method Invocation
IDE	Integrated Development Environment
OSGi	Open Services Gateway initiative
JAR	Java Archive

Vorwort

Die Entwicklung neuer Technologien im Bereich Software schreitet stetig voran. Während vor einigen Jahren die Wahl einer geeigneten Programmiersprache relativ überschaubar war, bieten sich dem versierten Programmierer heute eine Fülle von Möglichkeiten, um seine Vorstellungen und Wünsche entsprechend umzusetzen.

Vor allem im Bereich der Oberflächenentwicklung ist dies deutlich zu erkennen. Musste eine Benutzeroberfläche früher noch wenigstens alle funktionalen Anforderungen erfüllen, spielen heute zunehmend auch Individualität und Bedienerfreundlichkeit eine entscheidende Rolle.

Dies haben viele Firmen schon längst erkannt und bieten in ihren Anwendungen zahlreiche Features an, die Benutzer von der hohen Güte und dem Zeitgeist ihres Produktes überzeugen sollen.

Ein besonderes Problem stellt dieser Trend für Softwareprodukte dar, welche erstmals vor mehreren Jahren erstellt wurden und sich bis heute kontinuierlich weiter entwickelt haben. Eine Portierung[1] ist meist nicht ohne weiteres möglich, insbesondere dann nicht, wenn der Umfang der Veränderungen erhebliche Zeit in Anspruch nimmt oder sich alte und neue Technologien nur schwer verbinden lassen.

Um dabei nicht zu riskieren, dass ein Produkt keinen Absatz mehr findet, weil es veraltet ist, müssen hier einschneidende Veränderungen vorgenommen werden, um auf dem Softwaremarkt konkurrenzfähig zu bleiben.

Mit diesem Thema und konkret am Beispiel des Leasman der Firma DELTA proveris AG, beschäftigt sich diese Diplomarbeit. Hier wird eine Möglichkeit aufgezeigt, mit der sich die mit Sybase PowerBuilder erstellte Benutzeroberfläche des Leasman, kontinuierlich in einem Entwicklungsprozess hin, zu einem eigenständigen graphischen Benutzerinterface überführen und wie sich das Problem der vorübergehenden Koexistenz beider Technologien bewältigen lässt.

[1] Wechsel eines wesentlichen Teils einer eingesetzten EDV-Lösung

Kapitel 1
Einleitung

Dieser Teil dient zur Heranführung an den Inhalt der Diplomarbeit und der näheren Erläuterung des Problems sowie dem gestellten Ziel. Außerdem wird die Vorgehensweise besprochen, die im Verlauf dieser Arbeit zum angestrebten Ziel führen soll.

1.1. Problemstellung

Der Leasman ist das Kernprodukt des mittelständigen Unternehmens DELTA proveris AG. Mit seiner Hilfe lassen sich allgemeine und spezielle Organisationsabläufe im Leasing- und Finanzbereich abbilden.

Vor mehr als zehn Jahren wurde mit der Implementierung begonnen und die Wahl der Entwicklungsumgebung fiel auf den damals sehr fortschrittlichen PowerBuilder von Sybase, welcher dem Vorbild einer 4GL Programmiersprache entspricht. Mit diesem war es möglich, in kürzester Zeit eine Geschäftsanwendung zu entwickeln, welche sowohl funktionellen als auch graphischen Ansprüchen genügte. Ein großer Vorteil des PowerBuilders ist dabei der Umgang mit Datenbanken. Dafür werden sogenannte Datawindows verwendet, die einfachen Zugriff auf Daten bieten und diese graphisch darstellen.

Die damals verwendete Version des PowerBuilders war die Version 6.0. Bis heute wurde die Entwicklung von Sybase kontinuierlich weiter voran getrieben und somit befindet sich der PowerBuilder heute in der Version 12.0. Mit den neusten Versionen haben sich auch die Lizenzverträge verändert. Früher wurden noch Sammellizenzen vergeben, was heute nicht mehr möglich ist. Jeder Entwickler benötigt für den Einsatz an seinem Rechner nun eine eigene Lizenz. Dies erhöht die Wartungs- und Entwicklungskosten. Desweiteren wird in Communitys spekuliert, dass Sybase die Weiterentwicklung und damit den Support des PowerBuilders bald einstellen wird. Zudem kommt, dass der PowerBuilder nicht so erweiterungsfähig ist, wie dies für die Umsetzung von zukünftigen Konzepten bei DELTA proveris nötig wäre.

Bereits seit längerer Zeit hat man sich bei DELTA proveris das ehrgeizige Ziel gesetzt, die Geschäftslogik des Leasman in eine eigenständige Schicht, den BLS (Business Logic Server), auszulagern. Diese neue Schicht basiert auf Java Technologien und ein Großteil der Logik ist bereits überführt. Durch diese Umstrukturierung ist bereits ein wichtiger Schritt getan, um die Abhängigkeit vom PowerBuilder zu lösen. Es befindet sich jedoch noch sehr viel Geschäftslogik im Leasman selbst, wo diese nahezu untrennbar mit der graphischen Benutzeroberfläche verbunden ist.

Jene graphische Benutzeroberfläche soll nun ebenfalls ersetzt werden, was eine große Herausforderung darstellt.

Da der Leasman einen sehr hohen Funktionsumfang und daher auch entsprechend viel graphische Elemente besitzt, ist es nicht möglich, die graphische Benutzeroberfläche in einem Zug zu überführen und anzubinden. Dies muss sukzessive geschehen. Alte graphische Komponenten müssen weiterhin bestehen, bis sie vollständig überführt sind. Für diesen hybriden Systemaufbau von PowerBuilder und anderen graphischen Komponenten gibt es bisher noch keine Lösung.

Ein weiteres Problem ergibt sich bei der lokalen Kommunikation auf Programmebene zwischen PowerBuilder und anderen Programmiersprachen. Insbesondere in Richtung des PowerBuilders, denn diese Kommunikation ist im PowerBuilder nicht vorgesehen.

Das nachfolgende Komponentendiagramm zeigt den derzeitigen Aufbau des Leasman mit Anbindung an den BLS:

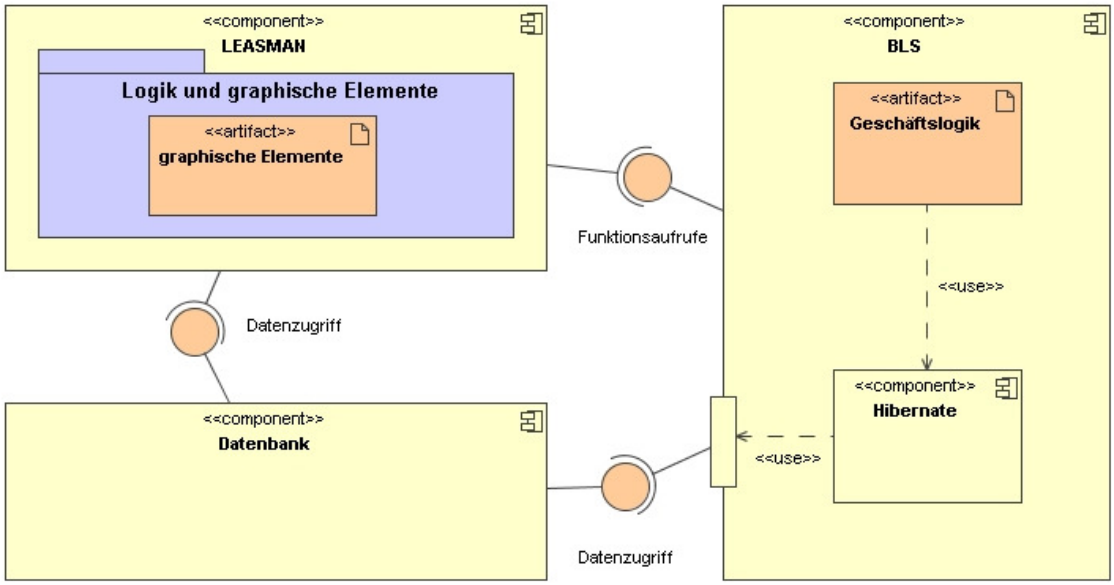


Abbildung 1: Aktueller Aufbau des Leasman mit Anbindung an BLS

1.2. Zielstellung

Die Ziele dieser Diplomarbeit liegen darin, eine Möglichkeit zu finden, wie sich eine auf aktuellen Technologien basierende graphische Benutzeroberfläche gleichzeitig an den im PowerBuilder und den im Business Logic Server befindlichen Teil der Geschäftslogik anbinden lässt. Dazu ist es notwendig, die für die DELTA proveris AG geeignetste Technologie zu ermitteln und ein Konzept zu erstellen, woraus ein Prototyp entwickelt werden kann, der einen Ansatz für die Lösung der Problemstellung bietet.

Teile der alten Benutzeroberfläche müssen vorerst bestehen bleiben, bis sie durch neue ersetzt sind. Hier ist das Ziel eine Möglichkeit zu finden, wie diese Fenster weiterhin verwendet werden können. Zukünftig und mit fortlaufendem Entwicklungsprozess soll der PowerBuilder immer weiter an Einfluss verlieren und letztlich vollständig abgelöst werden.

Es muss also in der Übergangszeit ein System geben, das aus unterschiedlichen Teilsystemen zusammen gesetzt ist. Jedoch müssen alle Komponenten miteinander kommunizieren und auch gegenseitig Ereignisse auslösen können. Insbesondere der PowerBuilder ist für derartige Anwendungen nicht ausgelegt. Hier sollen Möglichkeiten gefunden werden, die dieses Problem lösen.

Da dieses Hybridsystem auf Grund der längerfristigen Entwicklungsdauer auch an den Kunden ausgeliefert werden muss, ist es notwendig, die Benutzeroberfläche so zu gestalten, dass dem Anwender dadurch keine Einschränkungen hinsichtlich der Bedienung und Funktionalität entstehen. Dies setzt voraus, dass bei der Untersuchung auch darauf geachtet wird, wie sich Bedienerfreundlichkeit und Usability zukünftig positiv erhalten und verbessern lassen.

Das Ergebnis der Untersuchung und Konzeptionierung soll ein Prototyp sein, welcher aufzeigt, wie alte und neue Technologie koexistent und funktionell miteinander agieren.

Dieses Komponentendiagramm zeigt den Aufbau des Leasman, wie er nach der Einbindung des neuen Prototyps aussehen sollte:

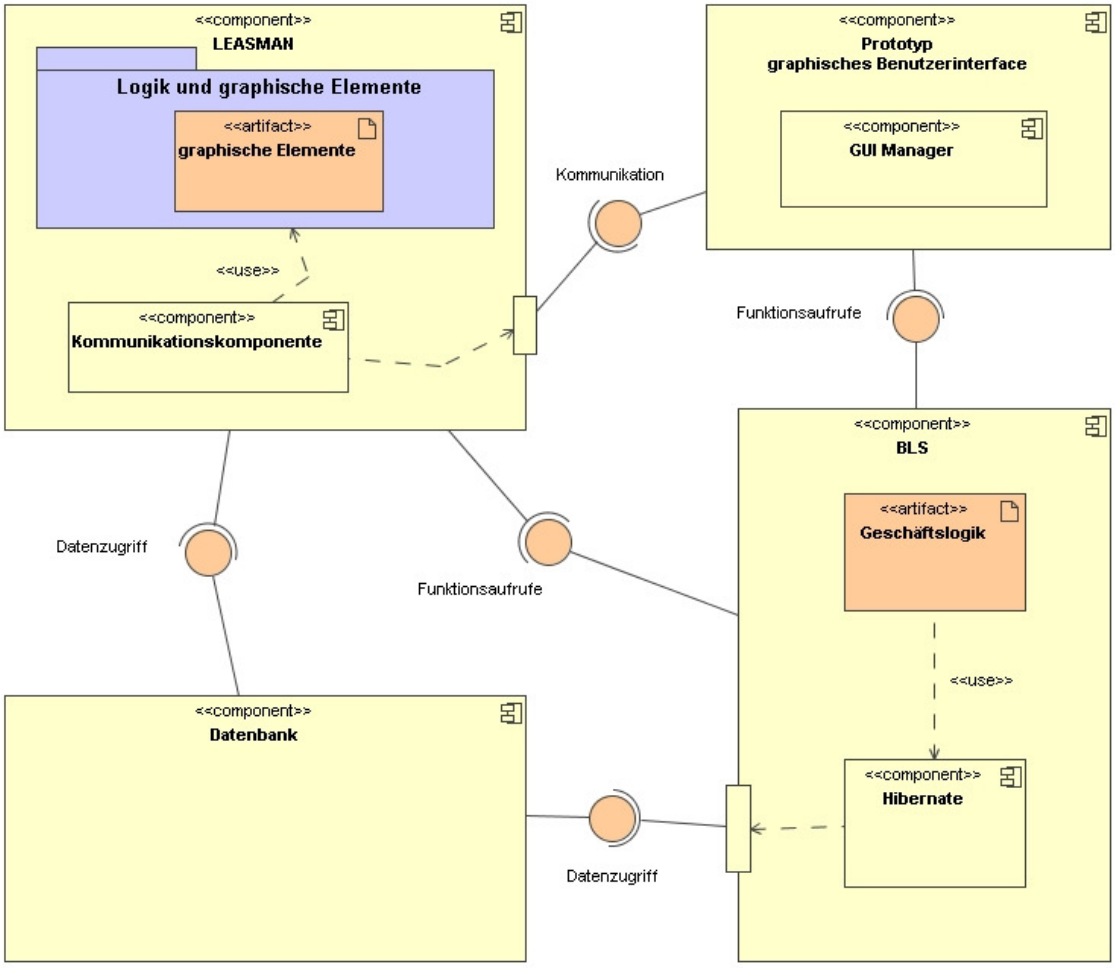


Abbildung 2: Gewünschter Zustand nach Einbindung des Prototyps

Für die Kommunikation zwischen Leasman und dem Prototyp der graphischen Benutzeroberfläche, muss im Leasman eine neue Komponente angelegt werden, die Aktionen aus der neuen GUI im Leasman auslöst. Ein Manager im neuen Prototyp übernimmt die Manipulation der graphischen Elemente des PowerBuilders. Ziel ist es, diese Kommunikation zu realisieren und diese Komponenten sowie einen einfachen Prototyp zu erstellen.

1.3. Abgrenzung

Da das Thema der Diplomarbeit sehr umfangreich ist, muss eine Abgrenzung hinsichtlich des Inhaltes und den Zielen geschehen.

Hauptziel soll die Erarbeitung einer Möglichkeit sein, wodurch die Kommunikation zwischen PowerBuilder und Java in beiden Richtungen ermöglicht wird und wie die bestehenden Komponenten der bisherigen Benutzeroberfläche vorerst weiter verwendet werden können.

Es soll nicht Inhalt dieser Diplomarbeit sein, die Überführung der bestehenden graphischen Benutzeroberfläche des Leasman zu planen und detailliert zu beschreiben. Dies wär aufgrund der Menge der Funktionen und Komponenten des Leasman im Rahmen dieser Diplomarbeit nicht zu bewältigen. Es sollen nur ausblicke vermittelt werden, die eine Überführung möglich machen.

Dabei soll es auch nicht Gegenstand dieser Arbeit sein, die prototypische Implementierung soweit zu perfektionieren, dass ein reibungsloser und fehlerfreier Einsatz des Prototyps möglich ist. Dies muss durch einen langfristigen Entwicklungsprozess nach Erarbeitung der Möglichkeiten geschehen.

Somit befasst sich diese Diplomarbeit nur mit der Erbringung eines Nachweises, dass die Kommunikation zwischen PowerBuilder und Java in beiden Richtungen möglich ist, sowie dass die bestehenden Fenster und Komponenten des Leasman weiter verwendet werden können.

Dazu gehört ebenfalls die geeignetste Technologie zu finden, mit der dies möglich ist und die beim späteren Überführungsprozess, sowie dem zukünftigen Einsatz, die meisten Vorteile verspricht.

Diese Arbeit liefert nur die Grundlagen, die eine Überführung der graphischen benutzeroberfläche möglich machen.

1.4. Vorgehensweise

Die Vorgehensweise bei der Umsetzung der Ziele dieser Diplomarbeit besteht darin, dass als erstes die theoretischen Grundlagen, welche für das Problem relevant sind, angesprochen werden.

Anschließend folgt eine Analyse des Ist-Zustandes der bestehenden Komponenten, insbesondere des Leasman und des BLS. Dabei sollen die verwendeten Technologien angesprochen und der Aufbau untersucht werden.

Im folgenden Teil findet eine Untersuchung der für die Konzeptionierung infrage kommenden Technologien statt. Dabei soll durch einen Vergleich ermittelt werden, welche für die DELTA proveris AG am geeignetsten sind.

Im nächsten Teil soll auf der Grundlage der ermittelten Technologie ein Prototyp konzeptioniert werden, welcher es ermöglicht, die bestehenden Fenster des PowerBuilders vorerst weiterhin zu verwenden, so dass ein hybrides System entsteht, dessen Verhalten dem einer nicht hybriden Anwendung gleicht. Dazu soll ein Actionserver konzeptioniert werden, welcher es ermöglicht, Anfragen für Aktionen von Java-Seite aus zu empfangen und diese im PowerBuilder auszuführen. Weiterhin soll die Verwaltung der graphischen Komponenten des PowerBuilders durch die Konzeption eines Managers erfolgen.

Der nächste Schritt soll die parktische Umsetzung des Prototyps sein. Dabei werden wichtige Bestandteile und Probleme bei der Implementierung genannt.

Der anschließende Teil gibt kurze Ausblicke und Empfehlungen für die weitere Vorgehensweise bei der Überführung der einzelnen Komponenten, nachdem die Implementierung und Stabilisierung des Prototyps abgeschlossen ist.

Schlussendlich wird im letzten Teil ein Fazit der vorliegenden Arbeit erstellt und es erfolgt ein Ausblick auf die möglichen zukünftigen Entwicklungen, welche durch den Einsatz der neuen graphischen Benutzeroberfläche denkbar sind.

Kapitel 2

Theoretische Grundlagen

Dieser Teil vermittelt einen kurzen Überblick über die theoretischen Grundlagen, die zum besseren Verständnis des Inhaltes und der Anforderungen an das gestellte Problem benötigt werden. Somit wird eine zielgerechte Sicht auf den wesentlichen Sachverhalt erlangt, welche im weiteren Verlauf Zusammenhänge leichter erkennen lässt.

2.1. Softwarealterung

„ Das ist historisch gewachsen … “

DELTA proveris AG

Unter Softwarealterung wird die allmähliche Verschlechterung der Qualität eines Softwareproduktes verstanden. Nachdem eine Software die klassischen Entwicklungsstadien durchlaufen hat und sozusagen erwachsen geworden ist, beginnt der längste Zyklus einer Software: die Wartung. Mit diesem Stadium beginnt praktisch schon der allmähliche Verfall.

In dieser Zeit expandiert die Codebasis und durch ständiges Überarbeiten beziehungsweise Erweitern von Methoden, wird der anfangs solide Quellcode immer mehr in Mitleidenschaft gezogen.

Parallel dazu diversifiziert[2] das System, was dadurch entsteht, dass neue Anforderungen gestellt werden, die bei der Entwicklung nicht vollständig berücksichtigt wurden oder noch nicht absehbar waren. So könnte zum Beispiel der Wunsch aufkommen, dass die Software auch andere Betriebssysteme unterstützt. Dies, soweit es überhaupt möglich ist, erfordert wiederum weitere Versionen, welche ihrerseits ebenfalls gewartet und weiter entwickelt werden müssen. Somit wird die Erweiterbarkeit in Mitleidenschaft gezogen, denn Änderungen, die alle Versionen betreffen, müssen dann unter verschiedenen Rahmenbedingungen umgesetzt werden. Hier werden dann auch Notlösungen gefunden, die von den üblichen Standards abweichen.

Hinzu kommt, dass ein System, welches einen jahrelangen Entwicklungsprozess durchläuft, ständig neuen Generationen von Entwicklern ausgesetzt ist. Diese haben meist nicht den vollen Überblick über die funktionellen Inhalte oder Zusammenhänge. Außerdem bringt jede Generation neue Programmierstile und Standards mit, die den Code weiter verzerren können.

Dieser Prozess kann im Extremfall zum Ableben eines Softwareproduktes führen, weil eine Weiterentwicklung nur noch unter erschwerten Bedingungen oder erheblichem finanziellen Aufwand möglich ist. [Vergl. Hoffmann08, S.371 – 407]

[2] Veränderung, Erweiterung, Abweichung – Spezialisierung auf mehrere Bereiche

Die nachfolgende Darstellung zeigt den Lebenszyklus eines Softwareproduktes:

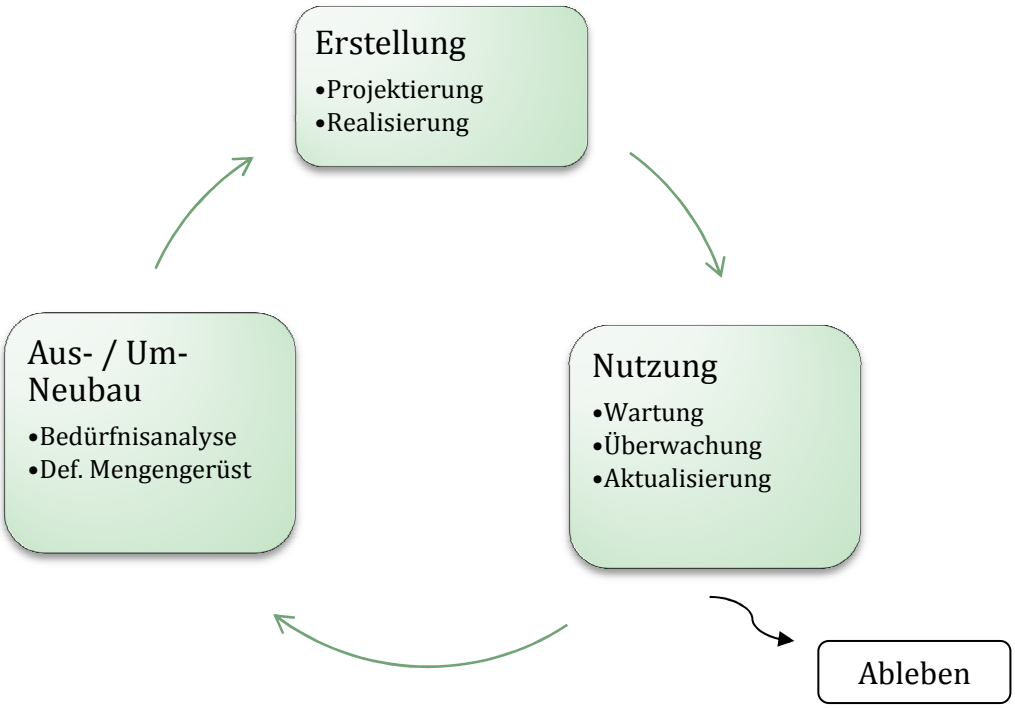


Abbildung 3: Lebenszyklus eines Softwareproduktes [ebc]

2.2. Softwareergonomie

Unter Softwareergonomie wird ein Teil der Softwareentwicklung zusammen gefasst, der sich mit der Bedienerfreundlichkeit eines Softwaresystems beschäftigt. Im Vordergrund stehen dabei stets Effektivität, Effizienz und Zufriedenheit der Bediener, die ein Programm benutzen.

Vor allem Benutzeroberflächen spiegeln diese wichtigen Eigenschaften von Software wider. Durch die richtige Konzeptionierung und Planung von Benutzeroberflächen kann die Bedienerfreundlichkeit eines Systems auf ein hohes Niveau gebracht werden.

Wenn bei der Entwicklung der Aspekt der Softwareergonomie vernachlässigt wird, hat dies direkt Auswirkung auf die Effektivität und Effizienz der Benutzer. Resultierende Probleme sind hohe Einarbeitungszeiten oder unentbehrliche Experten, die gelernt haben, mit dem System umzugehen. Solche Fachkräfte dürfen nicht ausfallen, beziehungsweise können nicht einfach ersetzt werden.

Weiterhin ist es möglich, dass ein Softwaresystem gegen die gewohnte Arbeitsweise funktioniert, wodurch die Benutzer unnötig angestrengt und ihre Konzentrationsfähigkeit herab gesetzt wird.

Softwareergonomie liefert die Richtlinien zur Analyse sowie Modellierung und Gestaltung von aufgabengerechten Benutzerschnittstellen.
[Vergl. Herczeg05, S.1 – 9, S.115 – 144]

2.3. Usability

Obwohl Usability im Wesentlichen Bestandteil der Softwareergonomie ist, kann sie genauer definiert werden.

Usability beschreibt nicht die Benutzerfreundlichkeit eines Softwareproduktes, sondern die Gebrauchstauglichkeit für eine bestimmte Gruppe von Benutzern, die dieses für ihre Zwecke einsetzen. So kann ein Programm zwar benutzerfreundlich jedoch für die Zielgruppe, die es einsetzt, gänzlich ungeeignet sein. Es gilt also nicht allein darauf zu achten, dass eine Benutzeroberfläche den modernen Standards entspricht, sondern auch, dass auf die individuellen Bedürfnisse des Endbenutzers eingegangen wird.

Gelingt es die Usability zu verbessern, wirkt sich dies direkt auf die Effektivität eines Softwareproduktes aus. Wenn es dem Anwender leichter fällt, seine Aufgaben mit dem verwendeten Werkzeug zu erledigen und er dadurch seine Konzentrationsfähigkeit länger aufrecht erhält, kann er mehr Arbeit in derselben Zeit verrichten, was wiederum Kosten senkt.

Durch Optimierung der Gebrauchstauglichkeit kann ein wichtiger Wettbewerbsvorteil verschafft werden, der in den nächsten Jahren von entscheidendem Vorteil sein wird. Es gilt diesen Vorteil nachhaltig und langfristig zu sichern. [Vergl. usab]

„ Usability is more than just making something easy-to-use and efficient. It's about pleasing users. Visual design can be just as important as good navigation, search and information hierarchy when it comes to pleasing users. If the visual design is not up to par then users will likely complain. "

Lani Hathaway

Usability Engineer at Electronic Arts [comx]

2.4. Software Redesign

Das Wort Redesign hört man oft in Verbindung mit Webseiten. Dort wird ein Redesign dann notwendig, wenn das bisherige Modell der Webseite nicht mehr den Standards entspricht und die Besucherzahlen schwinden.

Für Anwendungssoftware trifft dies auch zu. Hier wird ein Redesign nötig, wenn die Benutzeroberfläche einer Anwendung, im Gegensatz zur Umgebung in der sie ausgeführt wird, veraltet erscheint.

Mit wachsender Prozessorleistung und immer besseren Technologien, die verwendet werden, haben sich auch die Gestaltungsmöglichkeiten stark verändert. Eine etablierte Anwendung kann dieses Defizit anfangs durch seine Funktionalität kompensieren. Wenn jedoch neue Konkurrenzanwendungen im Bezug auf Funktionalität gleichziehen, wird es immer schwerer, die Software zu veräußern. An diesem Punkt ist ein Redesign nötig, sonst kann dies zum Ableben des Softwareproduktes führen.

Wird ein Redesign durchgeführt, sollte darauf geachtet werden, dass moderne Technologien und Standards eingesetzt werden. Das alte Konzept darf dabei nicht vollständig verworfen werden, damit der Benutzer nicht alles neu lernen muss. Es ist besser, von Version zu Version kleine Neuerungen einzuführen, um den Anwender langsam an die neue Form zu gewöhnen.

Weiterhin sollte man genau abwägen, welche Teile des alten Modells dem Redesign zum Opfer fallen, oder welche Teile in das neue Design übernommen werden sollten, da sie einen Vorteil darstellen. Dabei kann auf die Erfahrungen zurück gegriffen werden, die mit der alten Anwendung gesammelt wurden.

Das Redesign kann sich aber auch auf die Architektur beziehen, wobei jedoch nicht gemeint ist, die Architektur durch eine neue zu ersetzen, wie beim Architektur-Reengineering, sondern lediglich die Überarbeitung beziehungsweise Korrektur der Architektur.

Letztendlich sollte das neue Design eine deutliche Verbesserung zum alten Konzept darstellen. [Vergl. HK03]

2.5. Software Sanierung

Der Begriff Sanierung kommt aus dem Lateinischen und bedeutet Heilung oder Erneuerung.

Im übertragenen Sinne kann man diese Beschreibung auch auf Software abbilden, da Anwendungen im Rahmen ihrer Weiterentwicklung zahlreichen Ausbauten und Änderungen unterworfen sind und sich ihr struktureller Zustand dadurch stetig verschlechtert. Dieser Prozess ist nicht zu verhindern.

Ein wesentlicher Punkt ist, dass oft aus verschiedenen Gründen Notlösungen, sogenannte Workarounds[3], eingearbeitet werden, die dann vielleicht vergessen, oder nie wieder aus dem Code entfernt werden. Mit der Zeit führt dies dazu, dass eine Anwendung regelrecht „degeneriert".

Das wichtigste Werkzeug, neben den bereits in den vorherigen Punkten genannten, ist hierbei das Refactoring, welches direkt dort ansetzt. Gezielt wird auf Stellen im Programm geachtet, die überarbeitet werden sollten, weil sie entweder nicht mehr nachvollziehbar oder nur noch unter erschwertem Aufwand erweiterbar sind. [Vergl. Kübeck09, S. 11 – 26]

2.6. 4GL - Fourth Generation Language

Der Begriff 4GL wurde erstmals von James Martin in seinem Buch „Applications Development Without Programmers" geprägt.

Während bei Programmiersprachen der ersten und zweiten Generation direkt mit der Hardware kommuniziert wird, benötigt man für Programmiersprachen der dritten Generation bereits einen Compiler[4]. Dieser übersetzt den programmierten Code für das jeweilige System, in dem es ausgeführt wird, in Maschinensprache. Solche Sprachen sind zwar schon in einer leicht verständlichen Weise aufgebaut. Jedoch ist vor allem die Fehlerbehandlung noch sehr umständlich und alle einzelnen Elemente müssen zum Großteil mit Hand erstellt werden.

[3] Unter Kompromissen eingegangene Notlösung für ein Problem.

[4] Übersetzungsprogramm, welches Programmcode in Maschinensprache überführt.

4GL war der nächste Schritt in der Evolution der Programmiersprachen. Meist gibt es für deren Entwicklung eine spezielle Entwicklungsumgebung, womit man einfach und schnell graphische Anwendungen über Drag und Drop[5] zusammen stellen und die einzelnen Elemente dann mit Funktionalität belegen kann. Dadurch verringert sich die Entwicklungszeit erheblich und durch den automatisch generierten Code wird das Programm weniger fehleranfällig.

Diese Programmiersprachen waren in den 90er Jahren sehr beliebt, da sie im Gegensatz zu den bis dahin verwendeten Sprachen der dritten Generation, die Entwicklungszeit erheblich verkürzten. Außerdem galten nach den damaligen Standards Programme, die mit 4GLs entwickelt wurden, als sehr gut erweiterbar. Da heute jedoch der Trend immer mehr zu vielschichtigen Anwendungen geht und auch immer mehr verschiedene Technologien eingesetzt werden, geht man heute wieder zunehmend von 4GL Anwendungen weg, weil diese oft nur über eine oder zwei Schichten verfügen und sich nur sehr schlecht in ein Mehrschichtsystem überführen lassen. Die einst viel zitierte Kosteneinsparung durch 4GLs relativiert sich bei alten Anwendungen nun durch erhöhten Aufwand für notwendig gewordene Refactorings. [Vergl. 4GL]

2.7. Prototyping

Die gängigste Methode der Softwareentwicklung ist das Wasserfallmodell. Dabei wird ein neues Softwareprodukt Schritt für Schritt geplant und entwickelt. Jede Stufe erfordert eine genaue Vorstellung vom Endprodukt. In manchen Fällen ist eine strikte Vorgehensweise nach diesem Modell jedoch nicht möglich, da Aussehen und Funktionsweise des Produktes möglicherweise noch nicht genau feststehen.

Für diesen Fall werden in der Softwareentwicklung Prototypen eingesetzt. Durch iterative Weiterentwicklung eines groben Entwurfes können damit spezielle Eigenschaften einer Software heraus gearbeitet werden. Beispielsweise weil der Kunde keine genaue Vorstellung von der graphischen Benutzeroberfläche hat oder weil ein konkretes Konzept zur Lösung eines Problems noch nicht vorliegt.

[5] Objekte können mittels Zeigegerät wie Maus über ein Ziel gezogen und abgelegt werden.

Es werden beim Prototyping mehrere verschiedene Vorgehensweisen unterschieden. Zum einen das *explorative Prototyping*, bei dem der Benutzer bei der Entwicklung mit einbezogen wird. Dieser stellt Anforderungen, welche Entwickler dann zur Anschauung an einem Prototyp umsetzten. Anschließend entscheidet er, ob dies seinen Wünschen entspricht. Ist er noch nicht zufrieden, kann er seine Anforderungen ändern und das neue Ergebnis direkt sehen. Dieser Vorgang wiederholt sich, bis Entwickler und Benutzer zu einer Übereinstimmung gekommen sind.

Eine weitere Art des Prototypings ist das *experimentelle Prototyping*. Bei diesem werden verschiedene Möglichkeiten zur Lösung eines Problems durchgespielt, um die beste daraus zu ermitteln.

Exploratives- und experimentelles Prototyping dienen lediglich dem Zweck Erfahrungen für das Endprodukt zu sammeln. Sind die Arbeiten am Prototyp beendet, wird er verworfen und die gesammelten Erkenntnisse fließen dann in den Entwicklungs- und Planungsprozess der Software ein. Deshalb bezeichnet man diese Arten auch als Wegwerfprototypen.

Anders verhält es sich mit dem *evolutiven Prototyping*. Bei dieser Art des Prototypings wird an einem sehr frühen Punkt bei der Entwicklung einer Software mit der Programmierung begonnen. Dieser Vorläufer wird dann sukzessive weiterentwickelt. Somit geht der Prototyp irgendwann in das zu entwickelnde Endprodukt über. Solche Vorgehensweisen sind durchaus häufig in der Softwareentwicklung.

In der Praxis werden diese Prototypingarten oft vermischt, um den Anforderungen des Entwicklungsprozesses gerecht zu werden. [Vergl. Prot]

Kapitel 3
Ist-Analyse und angewandte Technologien

In diesem Teil wird der Aufbau des Leasman sowie des Business Logic Servers kurz erklärt. Außerdem werden die dafür verwendeten Technologien angesprochen. Dies soll dazu dienen, einen Überblick über die Rahmenbedingungen zu vermitteln, um die Problematik und weitere Vorgehensweise besser zu verstehen.

3.1. Die Leasingsoftware Leasman

Der Leasman ist das Kernprodukt des mittelständigen Unternehmens DELTA proveris AG. Mit seiner Hilfe lassen sich allgemeine und spezielle Organisationsabläufe im Leasing- und Finanzbereich abbilden.

Die Entwicklung des Leasman begann bereits vor mehr als zehn Jahren und wurde bis heute kontinuierlich fortgesetzt. Als Entwicklungsumgebung wurde der PowerBuilder von Sybase eingesetzt. Dabei handelt es sich um eine Sprache der vierten Generation. Leasman orientiert sich am Client-/Servermodell und besitzt eine grafische Benutzeroberfläche als Frontend, die ebenfalls mit PowerBuilder erstellt wurde.

3.1.1. Sybase PowerBuilder

PowerBuilder ist eine Programmiersprache der vierten Generation und bietet eine integrierte Entwicklungsumgebung[6], die von der Firma Sybase angeboten wird. PowerBuilder setzt auf das Client-/Serverprinzip, woraus sich ergibt, dass damit hauptsächlich zweischichtige Anwendungen entworfen werden können. Zur Anwendung kommt dabei die Programmiersprache Powerscript. Diese ist objektorientiert[7] und verfügt über ein Exceptionhandling[8], ähnlich dem von Java oder C#.

Ursprünglich wurde der PowerBuilder von Powersoft entwickelt und war eine der ersten integrierten Entwicklungsumgebungen. Später ging Powersoft eine Kooperation mit Sybase ein und der Ausbau des PowerBuilders wurde unter Sybase weiter voran getrieben. Die aktuellste Version ist die Version 12.0 und wurde für das erste Quartal 2010 angekündigt.

Ein besonderes Merkmal ist die enge Zusammenarbeit mit Datenbanken, wobei über sogenannte Datawindows SQL Anfragen und deren Ergebnisse graphisch verarbeitet und auf der Benutzeroberfläche angezeigt werden können. Änderungen an diesen graphischen Komponenten haben direkt Auswirkungen auf die Daten der Datenbank.

[6] Anwendung zur Entwicklung von Software.

[7] Zusammengehörige Funktionalitäten werden in Objekten vereint.

[8] Fehlerbehandlung

PowerBuilder war in den 90er Jahren sehr beliebt, da sich mit seiner Hilfe sehr schnell und einfach große grafische Anwendungen erstellen lassen, die als Datenhaltung verschiedene Arten von Datenbanken unterstützen können.

Mehrere Erweiterungen ermöglichen es dem PowerBuilder seine Funktionen auszubauen und bieten damit ein gewisses Maß an Flexibilität. Zum einen ist dabei das PowerBuilder Application Server Plugin zu nennen, mit dessen Hilfe sich PowerBuilder Objekte in Application Servern wie z.B. JBoss ausführen lassen.

Weiterhin kann man das .NET Framework ab der Version 12 direkt einsetzen und somit viele Funktionalitäten und Vorteile von diesem verwenden. Es gibt ab der Version 12 des PowerBuilders eine weitere Entwicklungsumgebung, mit der sich .NET Anwendungen erstellen lassen.

Eine sehr interessante Funktion ist die Möglichkeit, Java Klassen einzusetzen. Zu diesem Zweck werden Java Proxies erzeugt, welche von PowerBuilder aus instanziiert werden können. Diese enthalten ein Mapping[9] auf die Methoden einer Java Klasse. Durch den Aufruf einer Funktion in diesem Proxy, wird die Funktion in der Java Maschine ausgeführt. Es können somit in Java Prozesse angestoßen oder Daten verarbeitet sowie deren Rückgabewerte weiter verwendet werden. Dies wird durch das PowerBuilder Native Interface (PBNI) realisiert. PBNI ermöglicht es sowohl PowerBuilder Klassen von C++ aus zu instanziieren, als auch C++ Funktionalität in PowerBuilder zu benutzen. Über diesen Umweg wird dann wiederum mittels JNI, welches ebenfalls C++ verwendet, eine Verbindung zu Java hergestellt. [Armstrong04, S. 1 – 18, S.605 – 658]

Diese vereinfachte Abbildung verdeutlicht die Kommunikation zwischen PowerBuilder und Java mittels JNI und PBNI:

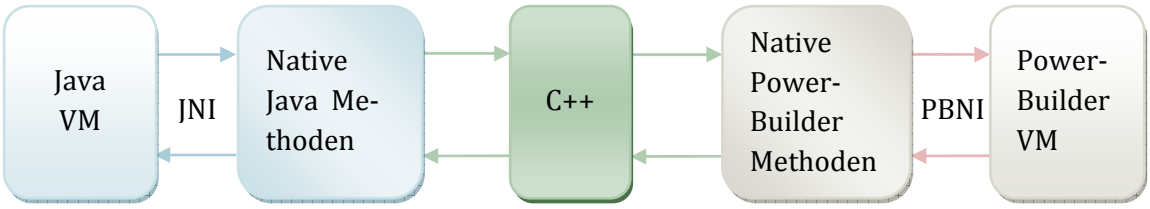


Abbildung 4: Kommunikation zwischen PowerBuilder und Java

[9] Abbildung von Daten oder Funktionalität zwischen verschiedenen Interpreten.

3.1.2. Anzahl und Verteilung der Klassen im Leasman

Über die Jahre hat sich der Leasman sehr stark vergrößert. Dies liegt unter anderem an der Diversifizierung durch mehrere parallele Projekte und sehr viele Anforderungen von Kunden, welche in das System eingearbeitet wurden. Daraus resultiert natürlich eine hohe Anzahl an Klassen und Fenstern.

Im Kreisdiagramm sind die verschiedenen Klassenarten und deren Anteil an der gesamten Objektanzahl veranschaulicht:

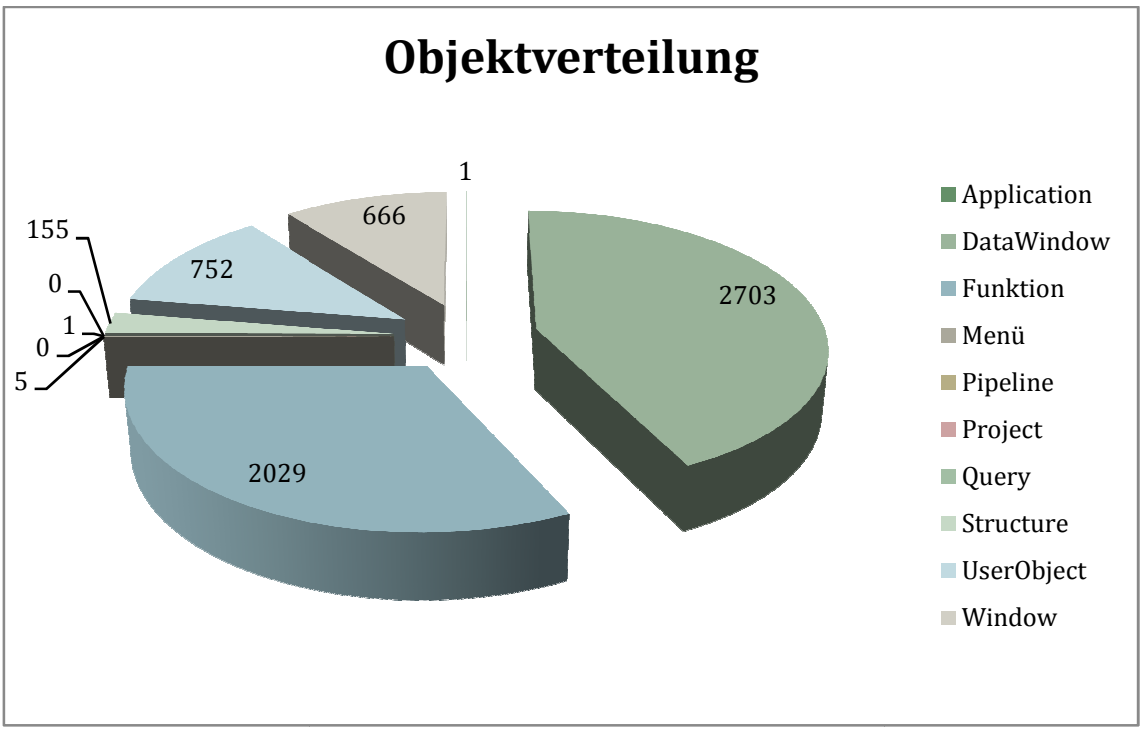


Abbildung 5: Verteilung der verschiedenen Klassen des Leasman

Aus dem Kreisdiagramm ist sehr gut zu erkennen, dass graphische Elemente mehr als die Hälfte aller Klassen ausmachen. Die Anzahl der Fenster beläuft sich auf 666 und die der Datawindows auf 2703. Insgesamt sind das 3369 graphische Elemente, die überführt werden müssen.

3.1.3. Vererbungshierarchie der Fenster

Um Fenstern im Leasman mit einem einheitlichen Aussehen zu versehen und gemeinsame Funktionalität zu abstrahieren, sind die Fenster in einer Vererbungshierarchie angeordnet.

Allerdings haben sich über die Zeit, durch die Entwicklungsumgebung PowerBuilder, Smells[10] in diesen Hierarchien gebildet. Diese entstehen dadurch, dass beim Ableiten eines Fensters einige Eigenschaften der Elternklassen nicht auf die Kinder vererbt, sondern kopiert werden. So wurde beispielsweise in den Elternklassen die Farbe der Fenster auf Silber (ein Grauton) gestellt. Diese Eigenschaft wird beim ableiten kopiert. Wenn man nun die Farbe des Elternfensters ändert, bleibt die Farbe der Kinder erhalten.

Ähnlich verhält es sich mit der Vererbung von Events und Funktionen. Wurde bei der Erstellung von Kindern nicht darauf geachtet, die Funktionalität der Elternklassen explizit aufzurufen, ob gewollt oder ungewollt, ist die Abstraktionskette an diesen Stellen unterbrochen und funktionelle Änderungen in den Elternklassen können nicht an Kinder weiter vererbt werden. Einige wenige Fenster durchbrechen diese Hierarchie vollständig, indem sie gar nicht von den Basisklassen erben.

Abbildung 6 zeigt die Vererbungshierarchie der Fenster. Als Basisklasse für alle Fenster dient die Klasse *w_basis*. Bei dieser werden grundlegende Eigenschaften, wie zum Beispiel die Größe, festgelegt. Von dieser abgeleitet ist *w_btn*, welches einige Schaltflächen für Speichern oder Abbrechen enthält. Davon erbt *w_btn_1dw* und von diesem wiederum *w_btn_2dw*. Diese Fenster sind für die Verwendung von Datawindows vorgesehen. Ein weiteres Fenster, das von *w_basis* erbt ist *w_mdi_rahmen* und von diesem *w_mdi_applikation*. Diese Hierarchie ist für das Hauptfenster vorgesehen, welches direkt nach dem Start des Leasman angezeigt wird.

[10] „Übel riechender Code“ – schlechter Programmierstiel

Dieses Klassendiagramm veranschaulicht die Vererbungshierarchie der Fenster:

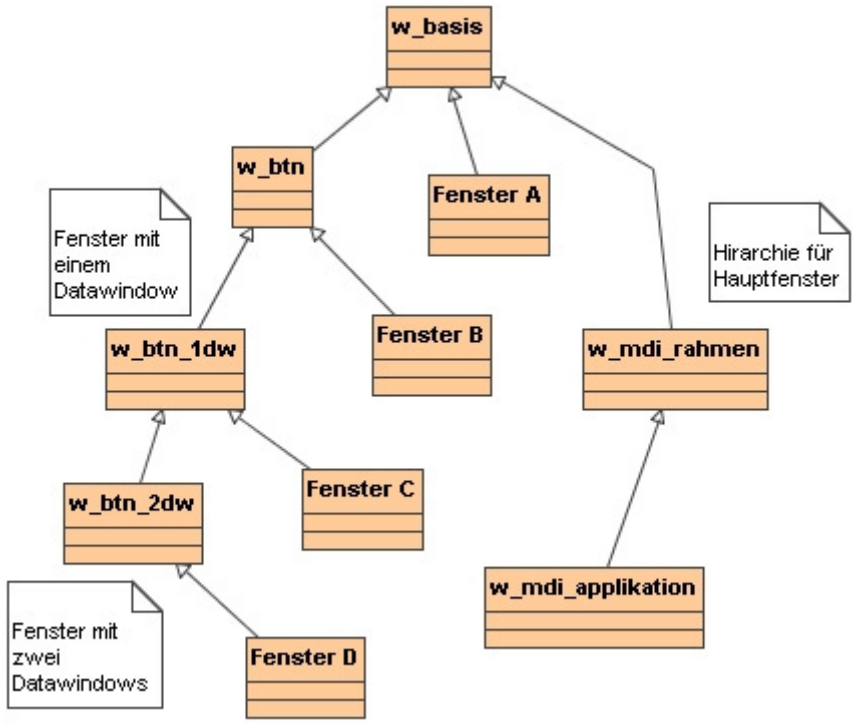


Abbildung 6: Vererbungshierarchie der Fenster

3.1.4. Benutzerabhängige Oberfläche

Zusätzlich kann die Benutzeroberfläche noch über verschiedene andere Arten beeinflusst werden. Eine Möglichkeit besteht darin, Menüeinträge ein oder auszublenden, um Anwendern Funktionalitäten frei zuschalten, beziehungsweise zu sperren, wenn diese für den Kunden nicht bereitgestellt werden sollen. Für diese Funktion gibt es Datenbankeinträge, welche die Struktur der Menüs für den Kunden festlegen. Beim Start des Programmes wird diese ausgelesen und dem Anwender stehen nur Funktionalitäten zur Verfügung, die in seiner Lizenz enthalten sind. Eine Manipulation dieser ist nicht möglich, da die Richtigkeit der Daten durch Hashcodes verifiziert wird.

Durch unterschiedliche Rechte von Benutzern werden einige Funktionalitäten auch in Fenstern und Auswahldialogen eingeschränkt. Diese Einschränkungen werden ebenfalls durch Einträge in der Datenbank gesteuert. Dadurch kann das Aussehen der Fenster noch einmal variieren.

3.1.5. Modularer Aufbau

Die Analyse der Architektur hat ergeben, dass der Leasman aufgrund der Entwicklungsumgebung PowerBuilder nicht über in sich abgeschlossene Komponenten verfügt. Der PowerBuilder stellt keine Möglichkeiten, wie Packages oder die Bildung von Teilsystemen durch Auslagerung in Projekten, zur Verfügung, wie sie aus anderen Programmiersprachen bekannt sind. Vielmehr ist die gesamte Anwendung als Ganzes zu verstehen. Es gibt zwar Bibliotheken, welche gemeinsame Funktionalitäten vereinen sollen. Diese haben aber eher den Charakter von Ordnern und dienen hauptsächlich zur Strukturierung. So sind Klassen gemeinsamer Funktionalitäten zwar zusammengefasst geordnet, untereinander haben aber alle Objekte Beziehungen über diese Bibliotheken hinaus. Es gibt keine klar zu erkennenden Grenzen, daher lässt sich auf dieser Ebene kein modularer Aufbau ableiten.

Die modulare Aufteilung orientiert sich beim Leasman an seiner Fensterstruktur. Über die Menüstruktur des Hauptfensters lassen sich zusammengehörige Komponenten erkennen.

Jeder Hauptmenüpunkt untergliedert sich in weitere Untermenüs, von denen aus Programmeinstiege erreicht werden können. Diese Programmeinstiege werden durch Auswahldialoge dargestellt, welche Auswahlmöglichkeiten für Vorgänge zu diesem Programmeinstieg bieten, die zu einer eigenen Programmmaske führen. Von diesen Masken aus können wiederum Fenster geöffnet werden, die sich über mehrere Ebenen erstrecken und auch zu anderen Einstiegen gehören können.

Bis zu diesen Programmeinstiegen und deren Auswahlmöglichkeiten kann eine klare Trennung vorgenommen werden. Jeder Einstieg beinhaltet explizit Auswahlpunkte, welche zu einem bestimmten Bereich des Verarbeitungsprozesses eines Leasingantrages führen. Somit können diese Programmeinstiege als zusammengehörige Teile verstanden werden, welche wiederum in kleinere Teilbereiche untergliedert und für sich abgegrenzt sind.

Eine genaue Analyse, welche Fenster betroffen sind, wenn ein Einstieg überführt werden soll und welche Beziehungen zwischen ihnen bestehen, ist hier allerdings nicht möglich, da dies den Rahmen dieser Diplomarbeit, aufgrund der hohen Anzahl der Fenster, übersteigt. Dies muss im späteren Überführungsprozess geschehen.

Die folgende Darstellung zeigt die Menüstruktur des Leasman. Jeder Untermenüpunkt stellt einen Einstieg dar:

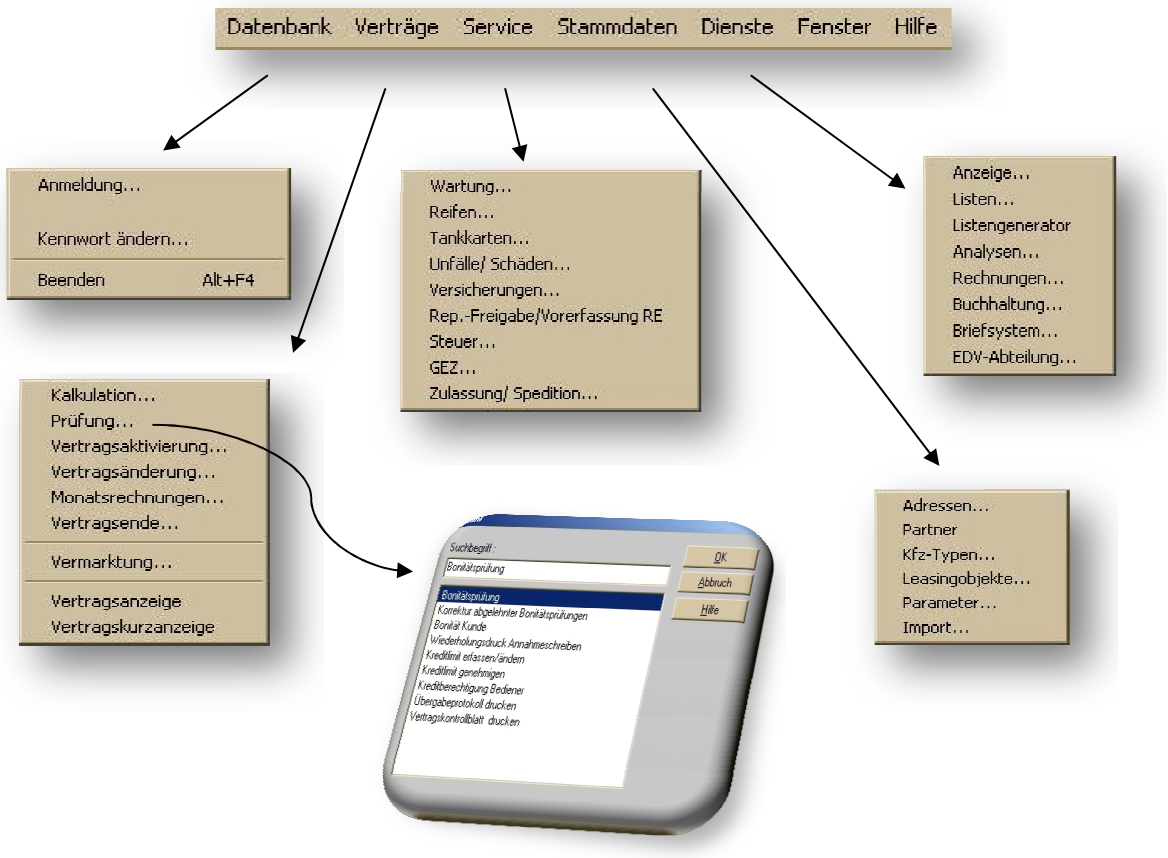


Abbildung 7: Menüstruktur Leasman

In Abbildung 7 ist das Hauptmenü des Leasman dargestellt. Von Interesse für die Analyse abgrenzbarer Teile, sind die Menüs *Datenbank*, *Verträge*, *Service, Stammdaten* und *Dienste*. Diese stellen große Programmbereiche dar. Jedes dieser Hauptmenüs verfügt über viele einzelne Untermenüs, welche zu Programmeinstiegen führen, die wiederum diese Programmbereiche in einzelne zusammengehörige Gruppen einteilen. Innerhalb dieser Einstiege gibt es eine weitere Unterteilung durch eine Auswahlliste, die diese Gruppen in einzelne fachliche Teile eines Leasingprozesses untergliedert. Jeder Auswahlpunkt führt zu einem Modul welches durch ein Fenster dargestellt wird.

Der oben dargestellte Programmeinstieg zeigt die Module für die Prüfung von Verträgen.

3.2. Business Logic Server (BLS)

Der BLS repräsentiert die neue Mittelschicht des Leasman und befindet sich derzeit in der Entwicklung. Fertiggestellte Teile des BLS sind bereits im Leasman eingebunden und werden auch von Kunden verwendet. Mit Hilfe des BLS soll die Geschäftslogik ausgelagert und der bisherige Client nach und nach in einen Thin Client überführt werden.

Als Programmiersprache kommt hauptsächlich Java zum Einsatz, welches eine hervorragende Erweiterbarkeit bietet und zudem systemunabhängig ist. Die vielen Technologien, welche mit Java zusammen arbeiten, erhöhen seine Flexibilität weiterhin.

Die Geschäftslogik ist in Geschäftsobjekten[11] gekapselt, welche eingehende Daten sowohl validieren, als auch wenn nötig vervollständigen. Mittels Hibernate werden diese Daten anschließend auf eine Datenbank abgebildet und gespeichert. Weiterhin können Objekte über das Spring Framework[12] an mehreren Stellen wiederverwendet und geladen werden. Dies vereinfacht die Weitergabe und Bearbeitung von Objektdaten an verschiedenen Stellen.

Der Business Logic Server bietet seine Dienste remote mittels Webservice an und kann somit von entfernten Rechnern aus angesprochen werden, um Funktionalität ausführen sowie Daten speichern und abrufen zu können.

Aufgrund des sehr flexiblen Aufbaus des Business Logic Servers, ist es sehr einfach, ein graphisches Benutzerinterface daran anzubinden, da die über Webservice angebotenen Funktionalitäten komfortabel aufgerufen werden können.

Im BLS ist eine modulare Trennung der Geschäftslogik bereits vorgenommen. Diese richtet sich ebenfalls nach der in Abbildung 7 gezeigten fensterorientierten Einteilung des Geschäftsprozesses. Hier wurde darauf geachtet, eine klare Separation der einzelnen Komponenten einzuhalten und Zyklen sowie zu starke komponentenübergreifende Beziehungen zu vermeiden.

[11] engl. Business Object: Enthalten nicht nur Daten für einen Geschäftsvorgang sondern auch für diese nötige Verarbeitungslogik.

[12] Rahmen zur Erstellung einer Softwareanwendung.

Diese Abbildung veranschaulicht die durch Einbindung des BLS entstandene 3-Schicht Architektur:

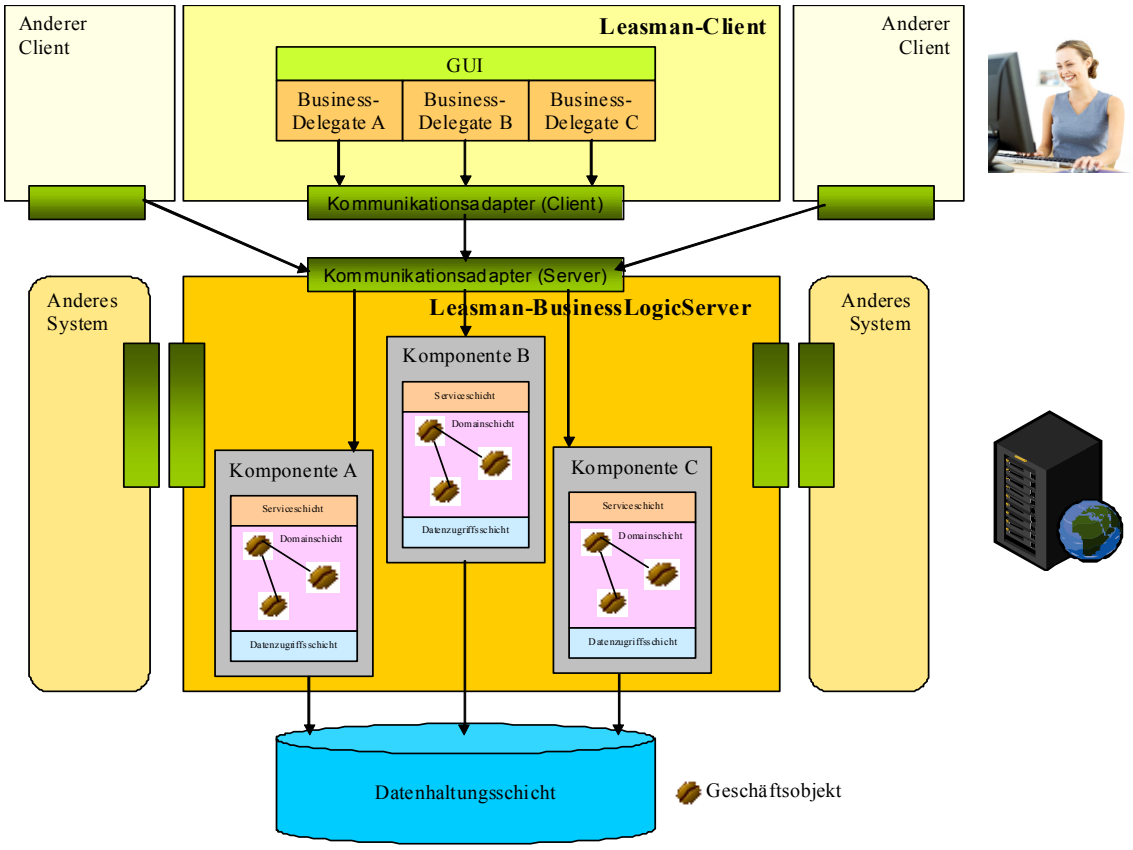


Abbildung 8: 3-Schicht Architektur Leasman mit Einbindung BLS [Mehner08]

Abbildung 8 veranschaulicht die Architektur des Leasman mit Einbindung des BLS. Es sind drei Schichten zu erkennen. Die oberste Schicht ist die Präsentationsschicht, die im Moment noch aus der im PowerBuilder erstellten Benutzeroberfläche besteht. Die Businessdelegates verbinden den Client jeweils mit einer Komponente der Mittelschicht im BLS.

Die Mittelschicht ist modular aufgebaut. Jede Komponente unterteilt sich in eine Serviceschicht, welche die darunterliegende Geschäftslogik verdeckt und eine einheitliche Schnittstelle bereit stellt. Weiter unten liegt eine Domainschicht, welche die Geschäftslogik enthält. Die Businessobjekte sind als Beans dargestellt. Eine weitere Datenzugriffsschicht dient zur Kommunikation mit der Datenbank.

Der dargestellte Kommunikationsadapter kann über verschiedene Arten, wie zum Beispiel EJB oder RMI realisiert werden.

3.2.1. Webservice

Webservices sind heutzutage weit verbreitet und werden immer öfter eingesetzt. Mit Webservices kann man Funktionalität anbieten, die von entfernten Rechnern aus aufgerufen und benutzt wird. Viele Firmen setzen Webservice ein, um Teile ihres Dienstes für Anwender zur Verfügung zu stellen, die diese gern in ihre Programme einbeziehen möchten. So zum Beispiel die Webservices von Google, Amazon oder auch eBay.

Ein Client baut eine Verbindung zu einem Server über eine eindeutige URI (Uniform Resource Identifier) auf. Nachdem ein Handshake ausgeführt wurde, können Daten ausgetauscht werden.

Dazu wird eine standardisierte Nachricht im XML Format vom Client an den Server geschickt, die Informationen über den gewünschten Webservice trägt. Der Server antwortet darauf ebenfalls mit einer Nachricht im XML Format, welche die angeforderten Informationen enthält. Diese Informationen können dann dank ihrer standardisierten Darstellungsform mit Hilfe eines XML Parsers in verschiedene Programmiersprachen übersetzt und wie gewohnt weiter verarbeitet werden. [Vergl. Fitch08]

3.2.2. Spring Framework

Das Spring Framework für Java wurde erstmals im Oktober 2002 von Rod Johnson als Begleitmaterial zu seinem Buch „Export One on One J2EE Design und Development“ veröffentlicht. Zu diesem Zeitpunkt befand sich das Framework noch in der Entwicklungsphase und die weitere Verwendung war noch nicht absehbar. Im Laufe der nächsten Jahre entwickelte sich Spring zu einem sehr beliebten und wichtigen Framework, welches im Jahre 2006 mit dem JAX Award[13] ausgezeichnet wurde.

[13] Wettbewerb für innovative Anwendungen

Spring bietet die Möglichkeit, verschiedenste Technologien wie Hibernate oder EJB, einheitlich zu konfigurieren und einzusetzen. Verschiedene APIs müssen nicht jeweils für sich selbst überwacht und gewartet, sondern können zentriert über das Spring Framework verwaltet werden. Dies hilft, den eigentlichen Programmcode von dem zur Konfiguration nötigen Quellcode frei zu halten.

Weiterhin lassen sich Abhängigkeiten zwischen Objekten komfortabel verwalten. Dies ist vor allem in der serviceorientierten Entwicklung wichtig, da diese Services von einander abhängig sind. Diese Abhängigkeiten müssen im Quellcode irgendwie realisiert werden. Durch Spring suchen sich Objekte nicht Abhängigkeiten zu anderen Objekten, sondern bekommen diese vom Springframework über sogenannte Dependency Injection zugewiesen.

Die Konfiguration erfolgt über die standardisierte XML Sprache. Dabei bleiben die Java Objekte weitestgehend unberührt. In XML Files werden Klasse und deren Abhängigkeiten beschrieben. Diese sogenannten Spring Beans lassen sich dann laden oder in Klassen, die diese verwenden „injizieren“, und müssen nicht umständlich instanziiert werden. [Vergl. Wolff06, S.V - VI, S.1 - 4]

3.2.3. Hibernate

Ein weiteres Open-Source Framework ist Hibernate. Es bietet die Möglichkeit, Java Objekte in relationalen Datenbanken[14] abzuspeichern. Diese Vorgehensweise bezeichnet man als Objekt-Relationales Mapping. Hibernate verwendet dazu die standardisierte JDBC-Schnittstelle.

Da die Java Objekte als POJOs (Plain Old Java Objects) behandelt werden, befindet sich die Beschreibung dieser in Mapping Dateien. Diese sind in XML verfasst. Mit ihrer Hilfe wird Hibernate mitgeteilt, wie ein Java Objekt aufgebaut ist und wie es abgespeichert wird. Alternativ kann die Beschreibung auch an Hand von Annotationen, die an die entsprechenden Elemente einer Klasse angehängt werden, erfolgen.

[14] Elektronisches Datenverwaltungssystem nach Edgar Frank Codd

Hibernate unterstützt eine eigene Sprache, das sogenannte HQL (Hibernate Query Language). In HQL geschriebenen Anfragen werden von Hibernate abhängig vom verwendeten Dialekt in SQL Statements umgewandelt. Das hat den Vorteil, dass Datenbanksysteme ausgetauscht werden können, ohne Änderungen an der Syntax der Anfragen vornehmen zu müssen. Es genügt Hibernate mitzuteilen, dass sich der SQL Dialekt geändert hat. Das Framework ist weit verbreitet und wird ständig weiter entwickelt.

Durch seine Unterstützung in Spring bietet es eine hohe Flexibilität. [Hib10]

Kapitel 4
Geeignete Technologien für den Prototyp

Nachdem die theoretischen Grundlagen für das bessere Verständnis der Thematik und des bestehenden Systems erläutert und analysiert wurden, wird in diesem Teil auf die Frage der für die Umsetzung zu verwendenden Technologie eingegangen. Dabei soll schrittweise an die voraussichtlich beste Möglichkeit herangeführt werden.

Für diese Analyse müssen einige wichtige Anforderungen an den Prototyp beachtet werden, welche Auswirkung auf die Wahl der Technologie haben. Der wichtigste Aspekt dabei ist, dass die alten PowerBuilder-Fenster vorerst weiter verwendet werden müssen, bis sie vollständig in die neue GUI überführt sind. Weitere wichtige Punkte sind Zukunftssicherheit und Flexibilität. In Anbetracht der bisher verwendeten Technologien sollte sich die neue Komponente außerdem leicht in die bisherige Architektur integrieren lassen.

4.1. Webbasierende oder eigenständige Anwendung

Zu Beginn wird untersucht, ob es sich bei dem zukünftigen Frontend[15] des Leasman um eine Webapplikation oder eine eigenständige Clientanwendung handeln soll.

Eine Webapplication hat im Gegensatz zu einer nicht webbasierenden Anwendung einige Vorteile, aber auch verschiedene Nachteile. Ein entscheidender Vorteil ist, dass eine Webapplikation keinerlei weitere Software benötigt, sondern lediglich ein Webbrowser zur Verfügung stehen muss, in dem die Applikation ausgeführt wird. Die Anwendung wird dabei vollständig vom Webserver übertragen und muss nicht auf dem Clientsystem installiert sein. Dadurch müssen Wartungsarbeiten lediglich an einer Stelle vorgenommen werden und dem Anwender steht immer die aktuellste Version zur Verfügung. Des Weiteren sind diese Webanwendungen Multiuser[16] fähig.

Nachteile ergeben sich aus der Tatsache, dass alle Informationen mit dem Webserver ausgetauscht werden müssen. Dies erhöht zum einen den Overhead[17], was eine möglichst schnelle Datenverbindung voraussetzt, als auch das Risiko für die Sicherheit gegenüber unerwünschten Zugriffen.

Zu den Nachteilen kommen noch weitere Kriterien, die den Einsatz einer webbasierenden Oberfläche ausschließen, denn die hohe Benutzerfreundlichkeit und Usability einer eigenständigen Anwendung kann mit einer Webapplikationen nicht erreicht werden. Zwar bieten verschiedene neuere Technologien wie Ajax Ansätze, um dem Endanwender das Gefühl zu vermitteln, dass es sich um eine eigenständige Anwendung handelt, die hohe Güte vergleichbarer Frameworks im nicht webbasierenden Bereich, kann damit jedoch nicht erreicht werden.

[15] Dem Endanwender am nächsten gelegener Teil der Anwendung bzw. Benutzeroberfläche

[16] Mehrere Benutzer können gleichzeitig mit einer Anwendung arbeiten.

[17] In der Datenverarbeitung Begriff für Informationen, die zusätzlich übertragen werden müssen.

Das stärkste Argument gegen den Einsatz einer Webapplikation ist jenes, dass die bestehenden Komponenten des Leasman nicht weiter verwendbar wären. Dem steht zwar entgegen, dass eine Umsetzung als Webanwendung wesentlich weniger Zeit in Anspruch nimmt als eine eigenständige Softwarelösung, doch die Überführung würde sich dennoch über einen längeren Zeitraum erstrecken. Auf die Verwendung der bestehenden Komponenten kann daher nicht verzichtet werden.

Diese Kriterien schließen den Einsatz einer webbasierenden Anwendung somit aus. Webanwendungen besitzen zwar viele Eigenschaften, die sie als Frontend attraktiv machen, den Möglichkeiten einer eigenständigen nicht webbasierenden Anwendung jedoch, sind sie deutlich unterlegen.

4.2. Wahl der Programmiersprache

Die Wahl der Programmiersprache knüpft sich an verschiedene Anforderungen und Eigenschaften. Sie sollte sich in möglichst vielen Bereichen damit decken.

- ✓ Die Hauptanforderung ist die Unterstützung der bisherigen PowerBuilder Komponenten und eine Möglichkeit zwischen den Programmiersprachen Informationen auszutauschen.
- ✓ Außerdem sollte die gewählte Sprache gut erweiterbar sein sowie möglichst viele Technologien unterstützen.
- ✓ Dabei ist auch zu berücksichtigen, wie diese Programmiersprache von den firmeninternen Ressourcen profitieren kann.
- o Als weiteres Kriterium muss untersucht werden, wie populär beziehungsweise zukunftssicher die gewählte Programmiersprache ist und wie hoch der Innovationsgrad eingeschätzt werden kann.
- o In Anbetracht der späteren Weiterentwicklung spielen Systemunabhängigkeit und Portierbarkeit eine Rolle.
- o Zudem ist zu beachten, ob der Einsatz eines Frameworks möglich ist.
- o Und letztendlich, wie das Look-and-Feel der Benutzeroberfläche ist.

✓	*Sollkriterium*
o	*optionales Kriterium*

4.2.1. Die .NET Sprache C#

Die Programmiersprache C# (C-Sharp) ist eine Entwicklung von Microsoft, Hewlett-Packard und Intel. Sie baut auf das .NET Framework von Microsoft auf und ist somit nur bedingt systemunabhängig, da eine Portierung hier hauptsächlich auf Windows basierenden Betriebssystemen möglich ist. Es gibt zwar Ansätze, das .NET Framework auf andere Systeme zu übertragen, diese sind aber noch nicht ausgereift.

C# bietet viele Features, welche durchaus attraktiv zur Umsetzung der Anforderungen sind. Sehr interessant ist, dass der Code anderer Programmiersprachen verwendet werden kann. So ist es in einem C# Programm sehr einfach, aus dem Code heraus COM Anwendungen oder auch Visual Basic, C++ und andere .NET Sprachen nicht nur aufzurufen, sondern auch von deren Klassen abzuleiten.

Da über das .NET Framework auf die Windows API zugegriffen werden kann, wäre es möglich, die bisherigen Fenster des Leasman so zu beeinflussen, dass sie in die neue Benutzeroberfläche integriert werden können. Durch die immer bessere Unterstützung des .NET Frameworks in PowerBuilder ist eine Kommunikation zwischen C# und PowerBuilder möglich.

Für die Entwicklung von C# Programmen gibt es verschiedene Entwicklungsumgebungen. Die bekannteste ist Visual Studio von Microsoft. Damit lassen sich sehr einfach und schnell graphische Benutzeroberflächen entwickeln. Mittels Drag and Drop können verschiedene Komponenten zu einer Anwendung zusammen gefügt werden, aus welcher dann C# Code generiert wird. Diese Komponenten lassen sich anschließend ähnlich wie im PowerBuilder mit Programmcode hinterlegen. Allerdings sind für Visual Studio Lizenzgebühren notwendig.
[Vergl. Gunnerson00, Teil 1 - 3]

4.2.2. JavaFX

Eine sehr neue Entwicklung von Sun Microsystems ist JavaFX. Es befindet sich in direkter Konkurrenz zu Adobe Flash und Microsofts Silverlight. Obwohl es ursprünglich für Webanwendungen und kleinere Geräte wie Handys und DVD Player entwickelt wurde, kann es als Nachfolger von Swing angesehen werden. Dieses hatte sich für Desktopanwendungen nie richtig durchsetzen können.

Momentan befindet es sich noch mitten in der Entwicklungsphase. Sun Microsystems hat entschieden, die verschiedenen Versionen vorerst nicht abwärtskompatibel zu gestalten, um auf die Wünsche der Community besser eingehen zu können. Dies bedeutet, dass mit jeder neuen Version Anpassungen im bestehenden Quellcode vorgenommen werden müssen.

JavaFX bietet einen sehr hohen Innovationsgrad und der Trend lässt erkennen, dass es sich zum neuen Standard für die Entwicklung von grafischen Benutzeroberflächen unter Java entwickeln wird.
[Java09.2007, S.16 – 19] [Java04.2010, S.63 – 67]

4.2.3. Java Swing

Swing ist eine Entwicklung von Sun Microsystems und seit Java 1.2 fester Bestandteil der Runtime. Swing baut auf das ältere Abstract Window Toolkit (AWT) auf und ist heute die Standardschnittstelle für die Entwicklung von grafischen Benutzeroberflächen unter Java. Dabei bietet es im Vergleich zu AWT weitere Möglichkeiten der Gestaltung.

Das Aussehen und Erscheinungsbild der Komponenten kann auf verschiedene Arten beeinflusst werden. Es orientiert sich jedoch nicht am Aussehen der Komponenten des Betriebssystems, in dem es ausgeführt wird. Dadurch unterscheidet sich seine Benutzeroberfläche, was sein Look and Feel[18] angeht, immer von der Umgebung in der sie gerade läuft.

[18] Erscheinungsbild und Verhalten einer graphischen Benutzeroberfläche

Durch die Verwendung von eigenen graphischen Komponenten ist Swing relativ langsam. Auch für Swing gibt es Drag und Drop basierende Editoren, mit denen sich Oberflächen komfortabel und schnell erstellen lassen.

Für die Kommunikation und die Verwendung der bestehenden PowerBuilder-Fenster ist der Einsatz von JNI (Java Native Interface) denkbar. Mit diesem lassen sich systemeigene Funktionen aufrufen, welche zur Manipulation der Komponenten benutzt werden können. Die Verwendung von JNI ist jedoch umständlich und eine Portierung auf andere Systeme wird dadurch stark eingeschränkt. [Swing]

Swing Application Framework:

Das Swing Application Framework kann eingesetzt werden, um kleinere Anwendungen zu entwickeln. Dabei hilft es, den Quellcode kompakt zu halten und Swing „richtig“ zu verwenden. Denn der schlechte Ruf von Swing, bezüglich der Geschwindigkeit, ist meist auf falsche Anwendung zurück zu führen. Es wird Funktionalität hinter den Elementen zu oft im Mainthread ausgeführt, anstatt diese in separaten Threads ablaufen zu lassen.

Das Swing Application Framework befindet sich aber noch in der Version 1.0, was keine hohe Funktionsvielfalt vermuten lässt. Für den Einsatz des Frameworks spricht eine sehr leichte Erlernbarkeit im Gegensatz zu beispielsweise Rich Client Anwendungen. [Java02.2008, S.70 – 76]

4.2.4. Standard Widget Toolkit (SWT)

Das Standard Widget Toolkit von Eclipse.org wurde 2001 für die graphische Benutzeroberfläche der Entwicklungsumgebung Eclipse entworfen. Da diese auf Java Technologien basiert, wollte man nicht auf Swing von Sun Microsystems setzen und entwickelte ein eigenes Toolkit, welches native Methoden des jeweiligen Betriebssystems einsetzt.

Dabei benutzt SWT die systemeigenen Komponenten, um seine Benutzeroberfläche zu gestalten. Durch diese nativen Komponenten ergibt sich ein Geschwindigkeitsvorteil gegenüber Swing, welches seine graphischen Komponenten selbst zeichnet. SWT nutzt dazu ebenfalls JNI.

Sind für einige Elemente keine nativen Widgets vorhanden, greift SWT auf Java Implementierungen zurück. Durch diese systemeigenen Widgets, hat eine in SWT geschriebene Anwendung, immer das Look and Feel des Betriebssystems, in dem es ausgeführt wird, was die Benutzerfreundlichkeit und Usability deutlich verbessert.

SWT ist unter einer open Source Lizenz erhältlich und kann zur Erstellung von Anwendungen kostenfrei genutzt werden. Dabei ist die derzeitige Ausbaustufe sehr hoch und wird ständig weiter entwickelt. [Sippel08, S.32, S.69 – 80] [Daum05, S.129 – 130]

JFace:

Optional kann eine Abstraktionsschicht mit dem Namen JFace verwendet werden, die auf SWT aufbaut. Durch diese wird der Umgang mit der API von SWT erleichtert und die Erstellung von einigen Komponenten einfacher. JFace ist zwar ebenfalls open Source, kann aber nicht eigenständig benutzt werden, da es Bestandteil von Eclipse ist. Soll JFace auch in anderen Entwicklungsumgebungen eingesetzt werden, müssen dessen Klassen und abhängige Pakete per Hand aus Eclipse extrahiert werden. [Sippel08, S.81 – 84]

4.2.5. Die Programmiersprachen im Vergleich

Ein Vergleich der genannten Programmiersprachen soll helfen, die für DELTA proveris am besten geeignete Variante zu ermitteln. Dazu wird die Unterstützung der einzelnen Kriterien bewertet und gegenübergestellt.

Die Bewertung der Unterstützung für die einzelnen Kriterien erfolgt der Einfachheit halber durch *Wenig*, *Mittel* und *Stark*.

Zusätzlich sind die Kriterien gewichtet. Dies erfolgt durch Färbung in drei Farbtönen, wobei dunklere Färbungen für ein höheres Gewicht stehen.

Diese Tabelle stellt die Bewertung der Kriterien gegenüber:

Eigenschaft	Webanwendung	C#	JavaFX	Swing	SWT
Verw. der PB-Fenster	Wenig	Mittel	Wenig	Mittel	Stark
Erweiterbarkeit	Mittel	Mittel	Mittel	Mittel	Mittel
Interne Ressourcen	Mittel	Wenig	Wenig	Mittel	Mittel
Innovationsgrad	Mittel	Mittel	Stark	Wenig	Mittel
Portierbarkeit	Stark	Wenig	Stark	Stark	Mittel
Framework vorh.	Mittel	Mittel	Wenig	Stark	Stark
Look and Feel	Wenig	Stark	Mittel	Wenig	Stark

Tabelle 1: Entscheidungstabelle einsetzbarer Programmiersprachen

Aus der Tabelle lässt sich erkennen, dass sowohl eine Webanwendung als auch eine auf JavaFX basierende graphische Benutzeroberfläche, durch die schlechte Unterstützung der bestehenden PowerBuilder Komponenten ausgeschlossen werden können.

Auch wenn C# und PowerBuilder zusammen eine gute Grundlage zur Lösung der Aufgabenstellung bieten, ist der Einsatz nicht empfehlenswert. Dies ergibt sich vor allem daraus, dass bei DELTA proveris hauptsächlich Java basierende Technologien eingesetzt werden und die inneren Ressourcen daher nur mit Wenig einzuschätzen sind. Die Einführung einer weiteren Programmiersprache würde sich außerdem negativ auf die bisherige Architektur auswirken. Dazu kommt, dass sich die Portierbarkeit von C# Anwendungen hauptsächlich auf Windows basierende Systeme begrenzt.

Swing und SWT bieten beide eine gute Unterstützung der gewünschten Kriterien. Dabei zeichnet sich Swing durch eine sehr gute Portierbarkeit aus und SWT hat dagegen ein sehr gutes Look-And-Feel. Wobei unter Swing der Einsatz von JNI notwendig ist, welches SWT bereits in sein Framework integriert. Daher ist die Unterstützung der PowerBuilder-Fenster in SWT mit Stark einzuschätzen und für Swing lediglich mit Mittel.

Für beide Technologien gibt es sehr gute Frameworks. Von Interesse sind dabei die Umsetzungen als Rich Client Plattform (RCP). Diese sollen im Weiteren noch einmal gegenübergestellt werden, da sich deren Verwendung anbietet.

4.3. Rich Client Anwendung

Vor einigen Jahren entwickelte sich ein regelrechter Internetboom. Dadurch wurde das Verlangen der Menschen nach uneingeschränkten Informationen gestillt. Mit diesem Aufschwung vergrößerte sich auch das Angebot an webbasierenden Lösungen für Clientanwendungen. Einige Vertreter dieses Trends glaubten sogar, dass schon bald alle Anwendungen nur noch über den Webbrowser ausgeführt werden.

Doch es gab immer noch viele Entwickler, die auf Desktopanwendungen als Client setzten, da die technologischen Möglichkeiten damit meist viel umfangreicher waren. Jedoch hatten Webanwendungen immer noch viele Vorteile. Einige Entscheidende waren die einfache Entwicklung und Wartung durch wiederverwendbare Komponenten. Aus diesem Bestreben heraus entwickelte sich der Wunsch, diese Vorzüge auch in Desktopanwendungen zu übernehmen.

Wie der Name Rich Client schon sagt, handelt es sich um Clients, die reich an Funktionalität sind. Der Entwickler kann dabei auf eine Fülle von vorgefertigten Komponenten zurückgreifen, um seine Anwendung möglichst schnell, aber mit hohen funktionalen Möglichkeiten zu erstellen.

Berthold Daum charakterisiert in seinem Buch „Rich-Client-Entwicklung mit Eclipse 3.1 – Anwendungen entwickeln mit der Rich-Client-Platform“ eine RCP wie folgt:

1. Sie besitzen eine State-*of-the-Art-Benutzeroberfläche*[19], die sich an dem graphischen Design des nativen Systems orientiert.
2. Sie haben eine *Notlaufeigenschaft*, die dem Programm erlaubt auch dann weiter zu arbeiten, wenn die Verbindung zum Server unterbrochen wird.
3. Die Konfiguration und Installation wird nicht vom Endanwender vorgenommen, sondern durch eine Art Update Funktion realisiert. (Server-managed clients) [Vergl. Daum05, S. 3 – 4]

[19] Höchst verfügbarer Entwicklungsstand einer Technologie

Viele der Vorzüge und Merkmale machen den Einsatz einer Rich Client Plattform (RCP) für die Umsetzung interessant. Zwar ist die Lernkurve anfangs sehr steil, um die vielen neuen Eigenschaften und Vorgehensweisen zur Entwicklung einer RCP Anwendung zu verstehen. Dies zahlt sich jedoch später durch die Einsparung von Entwicklungszeit, aufgrund der vielen Möglichkeiten der API, wieder aus.

Aus der vorherigen Untersuchung der anwendbaren Technologien lies sich schon erkennen, dass für die Erstellung vorrangig ein Java basierendes System in Frage kommt, um von den bestehenden Ressourcen zu profitieren. Swing und SWT fielen dabei in die engere Wahl. Nun sollen sowohl das auf Swing aufbauende NetBeans RCP und das auf SWT basierende Eclipse RCP auf seine Tauglichkeit für die prototypische Implementierung untersucht werden. [Vergl. Daum05, S. 1 – 10]

4.3.1. NetBeans RCP

Die NetBeans Plattform von Sun Microsystems basiert vollständig auf der Java SE und verwendet Swing zur Darstellung seiner graphischen Komponenten. Es bietet weitreichende Möglichkeiten zur Erstellung von graphischen Benutzeroberflächen nach dem Prinzip einer Rich Client Plattform.

NetBeans ermöglicht den modularen Aufbau einer Anwendung durch Entwicklung einzelner Plugins. Diese können in die Anwendung integriert werden, um seine Funktionalität zu erweitern.

Eine andere Erweiterungsmöglichkeit bieten Packs. Dabei handelt es sich um größere Pakete, die mehrere Module enthalten. Diese können verwendet werden, um den Einsatzbereich einer Anwendung komplett zu verändern oder zu erweitern.

Mit NetBeans steht eine sehr ausgereifte Plattform zur Verfügung, welche für nahezu jedes Problem eine professionelle Lösung bietet. Da NetBeans auf Swing basiert, muss keine neue Syntax erlernt werden. Die Entwickler müssen sich lediglich in die API einarbeiten. Zudem gibt es zahlreiche Beispiele und Dokumentationen für NetBeans. Durch die Verwendung von Java sind in NetBeans entwickelte Rich Clients vollständig systemunabhängig.

Wie bereits festgestellt wurde, ist negativ zu bewerten, dass Swing nicht das Look-and-Feel der nativen Umgebung vermittelt. Außerdem ergeben sich Probleme bei der Manipulation der bestehenden PowerBuilder Komponenten. Denn dies muss hier über JNI realisiert werden. [Vergl. Petri08,S.1 – 24]

4.3.2. Eclipse RCP

Eclipse ist eine weit verbreitete Entwicklungsumgebung für verschiedene Programmiersprachen. Ursprünglich war es für die Entwicklung von Java Anwendungen gedacht, wird aber heute für verschiedene Anwendungsbereiche eingesetzt.

Eclipse ist aus der früheren Version von IBMs Visual Age of Java 4.0 hervorgegangen und wird auch heute noch zu einem Großteil durch IBM finanziert. Es ist quelloffen und wird von der rechtlich eigenständigen Eclipse Foundation weiter entwickelt. Das Framework, welches zur Entwicklung von Eclipse eingesetzt wurde, wird für die Erstellung von RCP Anwendungen angeboten.

Die IDE bietet ein sehr gutes, natives Look-and-Feel, bedingt durch die Verwendung von SWT, welches bereits auf JNI aufbaut und dessen Funktionalität abstrahiert. Die Manipulation und Einbindung der bestehenden PowerBuilder Komponenten ist damit auf einfachem Wege möglich.

Auch hier ergibt sich eine hohe Einarbeitungsphase, da nicht nur die API erlernt werden muss, sondern auch die Syntax von SWT und JFace. Dies lohnt sich jedoch durchaus, da mit Eclipse RCP sehr hochwertige Clients entwickelt werden können, die viele vorgefertigte Möglichkeiten bieten. [Vergl. Daum05, S. 9 – 14]

4.3.3. Die Frameworks im Vergleich

Ein abschließender Vergleich der beiden Rich Client Platformen soll zur Entscheidung führen, welches Framework für die Konzeptionierung des Prototyps eingesetzt werden soll.

In Tabelle 2 erfolgt noch einmal eine Gegenüberstellung der beiden Frameworks miteinander, dabei geschieht die Bewertung der Unterstützung für die einzelnen Kriterien wie beim Vergleich der Programmiersprachen nach *Wenig*, *Mittel* und *Stark*. Auch hier ist durch verschiedene Farbtöne angedeutet, wie stark die einzelnen Kriterien gewichtet sind.

Nachfolgend erfolgt die Bewertung der Beiden RCP Frameworks:

Eigenschaft	NetBeans RCP	Eclipse RCP
Verw. der PB-Fenster	Mittel	Stark
Erweiterbarkeit	Stark	Stark
Interne Ressourcen	Mittel	Mittel
Innovationsgrad	Mittel	Mittel
Portierbarkeit	Stark	Mittel
Look and Feel	Mittel	Stark

Tabelle 2: Entscheidungstabelle NetBeans RCP, Eclipse RCP

Aus der Tabelle ist zu erkennen, dass sich beide Frameworks bei der Unterstützung der Kriterien sehr ähneln. Es gibt lediglich Unterschiede bei Portierbarkeit, Look and Feel und der Verwendung der PowerBuilder-Fenster.

NetBeans bietet hier die bessere Portierbarkeit. Dagegen steht das bessere Look-and-Feel bei Eclipse RCP. Das wichtigste Kriterium allerding, ist die Unterstützung der PowerBuilder-Fenster, welches bei Eclipse RCP mit Stark und bei NetBeans RCP nur mit Mittel eingeschätzt wird. Diese Tatsache lässt die Entscheidung letztendlich zu Gunsten von Eclipse RCP ausfallen.

Kapitel 5

Konzeption des Prototyps

In diesem Teil erfolgt die Konzeptionierung des Prototyps, im Bezug auf die gewählte Programmiersprache SWT und dem Framework Eclipse RCP.

Dabei sollen die wichtigsten Eigenschaften der Eclipse Rich Client Plattform genannt und auf den Aufbau einer RCP Anwendung unter Berücksichtigung der graphischen Oberflächenteile zur Verwendung für den Prototyp eingegangen werden.

Weiterhin erfolgt die Konzeptionierung eines Actionservers, mit dessen Hilfe sich von Java aus Aktionen in PowerBuilder ausführen lassen. Zudem wird erläutert, wie in entgegengesetzter Richtung vom PowerBuilder aus Aktionen in Java ausgelöst werden können.

Außerdem erfolgt die Konzeption eines GUI Managers, der für die Manipulation der PowerBuilder-Fenster zuständig sein soll.

5.1. Eigenschaften von Eclipse RCP

Da die Eclipse Rich Client Platform im vorherigen Kapitel als geeignetste Technologie für die Konzeptionierung eines Prototyps ermittelt wurde, werden hier noch einmal wichtige Eigenschaften dieser erläutert. Dies soll helfen, das Konzept von Eclipse RCP besser zu verstehen.

5.1.1. Eclipse Equinox OSGi

Eclipse RCP ist nach einem Komponentenkonzept aufgebaut und verwendet hierfür ein OSGi Framework als Laufzeitumgebung. Speziell eine von der Eclipse Fundation eigens zur Verwendung für die Eclipse IDE implementierte Version, das Equinox OSGi Framework.

OSGi ermöglicht es, Softwarekomponenten (Bundles) und Services, während der Laufzeit einer Anwendung dynamisch zu integrieren und zu entfernen, ohne dass diese Anwendung angehalten werden muss.

Bevor OSGi als Laufzeitumgebung für Eclipse eingesetzt wurde, besaß Eclipse eine eigene Laufzeitumgebung, mit der die Anwendung modularisiert wurde. Mit der Version 3.0 der Eclipse IDE wurde dies dann auf OSGi umgestellt.

[Vergl. Daum05, S. 26 – 27] [Sippel08, S.359 – 360] [Wütherich08, S.11 – 37]

5.1.2. Plugins

Plugins sind eigenständige Komponenten einer RCP Anwendung, die zur Laufzeit mittels Hot Plugging[20] in die Anwendung eingefügt und wieder entfernt werden können.

Da Eclipse RCP bis zur Version 3.0 nicht auf OSGi basierte, gibt es Unstimmigkeiten in der Namenskonvention. Bei RCP heißen die Komponenten daher nicht Bundles, sondern Plugins. Diese können aber als gleich angesehen werden. In neueren Versionen von Eclipse wird außerdem immer häufiger von Bundles als von Plugins gesprochen.

[20] Installieren, deinstallieren ohne Neustart

Die in einem Plugin enthaltenen Klassen und Ressourcen sind für andere Plugins nicht sichtbar. Sollen Klassen für andere Plugins sichtbar gemacht werden, muss ein Plugin die Packages, die diese Klassen enthalten exportieren. Plugins, welche diese Klassen nutzen möchten, müssen diese wiederum importieren. Dadurch kommt es unter den Plugins nicht zu Ressourcenproblemen.
[Sippel08, S.29 – 31]

5.1.3. Classloading

Jedes Plugin hat seinen eigenen Classloader. Dies ermöglicht es, Plugins unterschiedlicher Herkunft in einer Anwendung einzubinden, ohne deren Aufbau zu kennen, da jedes zum Laden seiner Klassen einen eigenen Namensraum verwendet und es damit nicht zu Konflikten kommen kann. Denn jedes Plugin kennt nur seine eigenen geladenen Klassen.

Durch das Exportieren und Importieren von Packages können Klassen eines Plugins für andere Plugins sichtbar gemacht und deren Klassen verwendet werden. Außerdem ist es möglich, Packages anzugeben, welche im Application- oder Systemclassloader geladen werden und somit allen Plugins zur Verfügung stehen. Zum Beispiel die Packages java.lang.* oder java.util.*, welche automatisch im Applicationclassloader geladen werden, damit sie in allen Packages genutzt werden können und diese nicht jedes Plugin in seinen eigenen Classloader laden muss. [Vergl. Wütherich08, S. 87 – 91]

Diese Abbildung verdeutlicht das Classloading der Plugins:

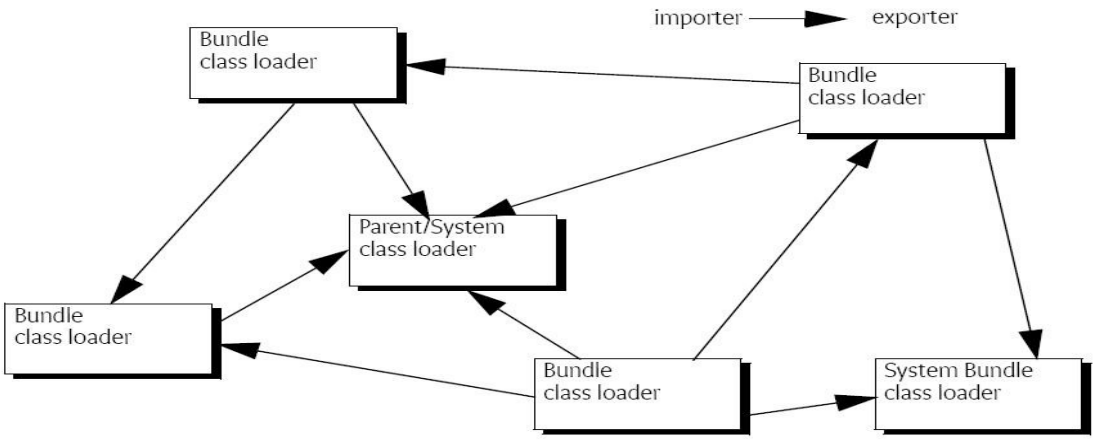


Abbildung 9: Classloading der Plugins [OSGi09, S. 37]

5.1.4. Extension Points

Erweiterungspunkte sind ein wichtiger Bestandteil des Plugin-Konzeptes von Eclipse RCP. Durch sie lassen sich Funktionalitäten von vorhandenen Plugins erweitern. In der Literatur werden diese Extension-Points oft mit Steckdosen verglichen, an denen sich Plugins anstecken können.

Bestehende Plugins bieten diese Extension Points als eine Schnittstelle an. An diesen Schnittstellen können sich andere Plugins anschließen, um dessen Funktionalität zu erweitern. Dazu müssen Plugins diese Erweiterungspunkte implementieren. Beim Start des Plugins werden diese Erweiterungspunkte initialisiert. Dies geschieht auch beim Hot-Plugging von Plugins.

So kann zum Beispiel erreicht werden, dass ein Plugin seine Menüeinträge in die vorhandenen Menüeinträge eines anderen Plugins mit integriert, ohne diesen Menüpunkt selbst noch einmal zu erstellen. Extension Points können für viele verschiedene Arten von Funktionalitäten eingesetzt werden.
[Sippel08, S.30 – 31, S.179 – 184]

5.1.5. Actions

Eclipse RCP bietet ein Konzept zur Wiederverwendung von Aktionen, die über Menüs oder Buttons ausgeführt werden sollen. Dazu können Actions registriert und bei Bedarf geladen werden.

Diese Actions enthalten den Quellcode, der nach aktivieren dieser Action zur Ausführung gebracht werden soll, in einer *run()* Methode. Weiterhin können Eigenschaften der Action hinterlegt werden, wie zum Beispiel ein Icon, welches bei Einbindung dieser Action in eine Toolbar angezeigt werden soll.

Somit können diese Actions einmal instanziiert und an mehreren Stellen verwendet werden. Zum Beispiel einmal im Menü und in der Toolbar. Oder an anderen Stellen wo Links, Buttons oder andere Elemente untergebracht sind, die vom Benutzer angesprochen werden können.

Eclipse RCP verwaltet selbständig die Darstellung dieser Actions in Abhängigkeit von der Stelle, an der sie eingebunden werden und ob es Buttons, Menüeinträge oder ähnliches sind. [Vergl. Daum05, S. 161]

5.1.6. Die Workbench

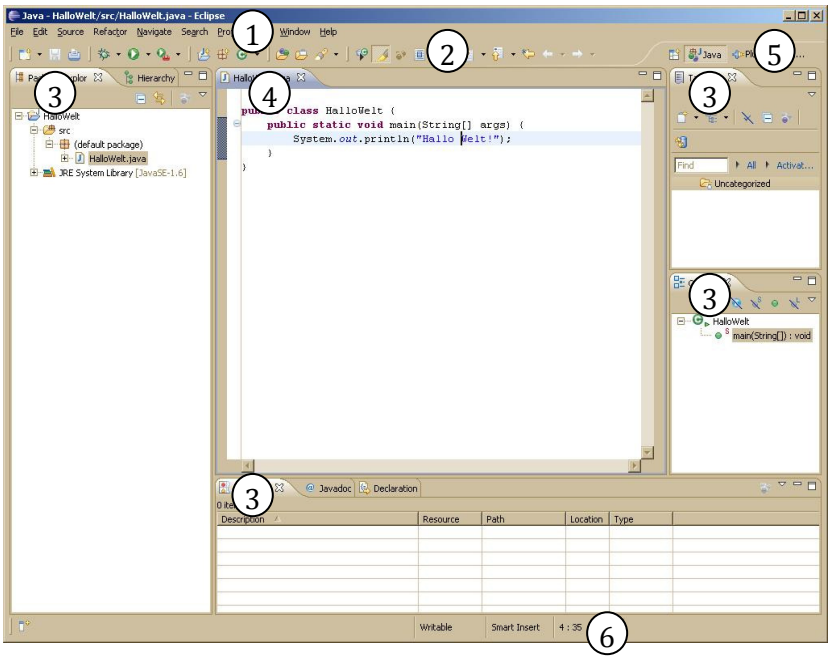
Die Workbench ist der Rahmen, in dem eine graphische RCP Anwendung strukturiert ist. In ihr werden alle Komponenten und graphischen Elemente angezeigt, welche zur RCP Anwendung gehören.

Es kann immer nur maximal eine Workbench geben. Sobald das letzte Fenster der Workbench geschlossen wird, schließt sich automatisch auch die Workbench.

Eine Workbench kann mehrere Fenster besitzen. Diese sind wiederum in Pages aufgeteilt. In diesen Pages befinden sich dann die Editoren, Views und andere graphischen Komponenten.

Das Aussehen einer Eclipse RCP Anwendung wird somit durch die Workbench festgelegt. Es können nur Anwendungen entwickelt werden, die diesem Oberflächenaufbau entsprechen.

Diese Abbildung zeigt die Workbench der Entwicklungsumgebung Eclipse und deren graphische Komponenten:



1 Menüzeile
2 Coolbar
3 Views
4 Editor
5 Perspektiven
6 Statuszeile

Abbildung 10: Workbench der Entwicklungsumgebung Eclipse

- *Menüzeile:*

  Die Menüzeile dient wie in jedem anderen Programm der Navigation.

- *Coolbar:*

  Die Coolbar dient ebenfalls der Navigation und kann Aktionen als Icons beinhalten.

- *Views:*

  Views dienen zur Darstellung von unterstützenden Informationen wie Navigationsbäumen oder Textausgaben. Es kann immer nur eine Ausprägung einer bestimmten View angezeigt werden. Views können innerhalb der Workbench beliebig verschoben und auch von der Workbench gelöst werden.

- *Editor:*

  Editoren zeigen typischerweise den Hauptinhalt der Anwendung. Sie können nur in der Editorarea angezeigt werden. Hier können mehrere Editoren desselben Typs geöffnet sein.

- *Perspektive:*

  Perspektiven dienen dazu, den Inhalt der Workbench auf verschiedene Weisen darzustellen. Es kann nicht nur die Anordnung der Views unterschiedlich, sondern auch andere Views geöffnet sein. Zwischen diesen Perspektiven kann man hin und her schalten.

- *Statuszeile:*

  Die Statuszeile kann weitere Informationen enthalten. Hier können auch Buttons, Icons und andere Schaltflächen und Elemente angelegt werden.
  [Vergl. Daum05, S. 43 – 57,] [Sippel08, S.93 – 95]

5.2. Verwendung der Workbenchkomponenten

Der Aufbau des Prototyps der LeasmanWorkbench soll sich vorerst am Design des alten Hauptfensters des Leasman orientieren. Das Hauptfenster wird dabei vollständig durch eine Neuimplementierung in Eclipse RCP ersetzt.

Die Menüstruktur wird genau so übernommen, wie sie im Leasman ist. Die Toolbar wird in Eclipse RCP durch die Coolbar ersetzt.

In der Statusleiste des Leasman werden einige Informationen über das aktuelle Geschehen sowie die Uhrzeit im Leasman Hauptfenster angezeigt. Diese Funktionen werden später ebenfalls mit der Statusleiste der Workbench realisiert.

Modale Dialoge bleiben im Prototyp unberührt. Es werden weiterhin die vom bisherigen Leasman verwendet.

Alle Childwindows, die nicht frei sind, sondern nur innerhalb des bisherigen Hauptfensters erscheinen, werden später in einem Editorfenster der Workbench eingebettet.

Diese Abbildung zeigt die graphischen Elemente des Leasman, welche beim Einbinden des Prototyps betroffen sind:

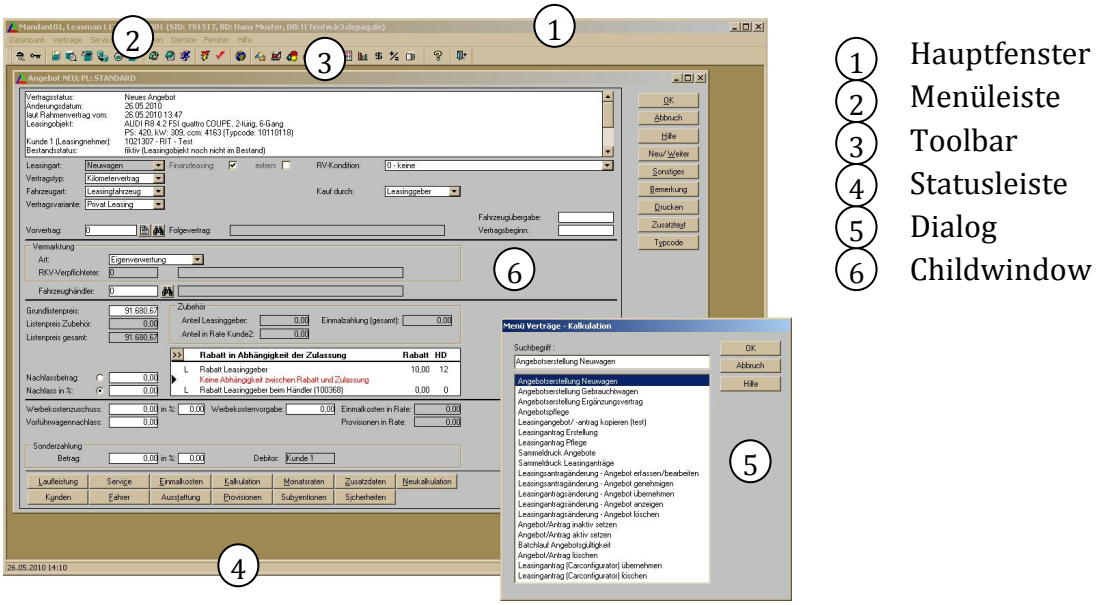


Abbildung 11: Komponenten der bisherigen GUI des Leasman

5.3. Der Actionserver

Der Actionserver ist die wichtigste Komponente des Adapters zwischen PowerBuilder und der Workbench. Mit seiner Hilfe können von Java aus Aktionen im PowerBuilder ausgeführt werden.

Um dies zu realisieren, muss der PowerBuilder auf eingehende Nachrichten (Aktionen) von der Workbench warten. Dies ist jedoch ein Problem, da der PowerBuilder nur als Singlethreadmaschine entwickelt wurde. Würde man den Actionserver im Mainthread starten und auf Nachrichten warten lassen, würde er den einzigen Thread des PowerBuilders so lange blockieren, bis eine Nachricht eingegangen ist und es könnte während dessen nicht im Leasman gearbeitet werden. Dies bedeutet: Es muss ein weiterer Thread erzeugt werden, welcher auf Aktionen der Workbench wartet.

In Abbildung 12 ist die Kommunikation zwischen Workbench und Leasman verdeutlicht. Der Actionserver arbeitet in einem eigenen Thread, welchen er so lange blockiert, bis eine Nachricht von der Workbench eintrifft. Für den Datenaustausch verwendet der Actionserver einen ObjectOutputStream von Java, den er mittels PBNI ansprechen kann. Der Mainthread kann während dessen weiter arbeiten. Nachdem der Actionserver eine Nachricht empfangen hat, löst er die zugehörige Aktion im Leasman aus und wartet danach auf weitere Nachrichten.

Die Action, welche der Server empfängt, beinhaltet lediglich den Namen der PowerBuilder-Action und die zur Ausführung benötigten Parameter. Dieser Name wird verwendet, um auf PowerBuilder-Seite eine gleichnamige Klasse zu erzeugen, in der eine abstrakte Methode ausgeführt wird, welche die Funktionalität für diese Aktion enthält. Sollte es nötig sein, können in der empfangenen Action, die auf Java-Seite ebenfalls als dieselbe Instanz verwendet wird, Rückgabewerte abgelegt werden, die nach Beendigung der Aktion wiederum auf Java-Seite ausgewertet werden können.

Sowohl der Actionserver als auch die Workbench müssen die zum Datenaustausch benötigten Klassen im selben Classloader laden, um dieselbe Referenz auf Objekte zu erhalten.

Die nachfolgende Darstellung verdeutlicht die Kommunikation zwischen Workbench und PowerBuilder:

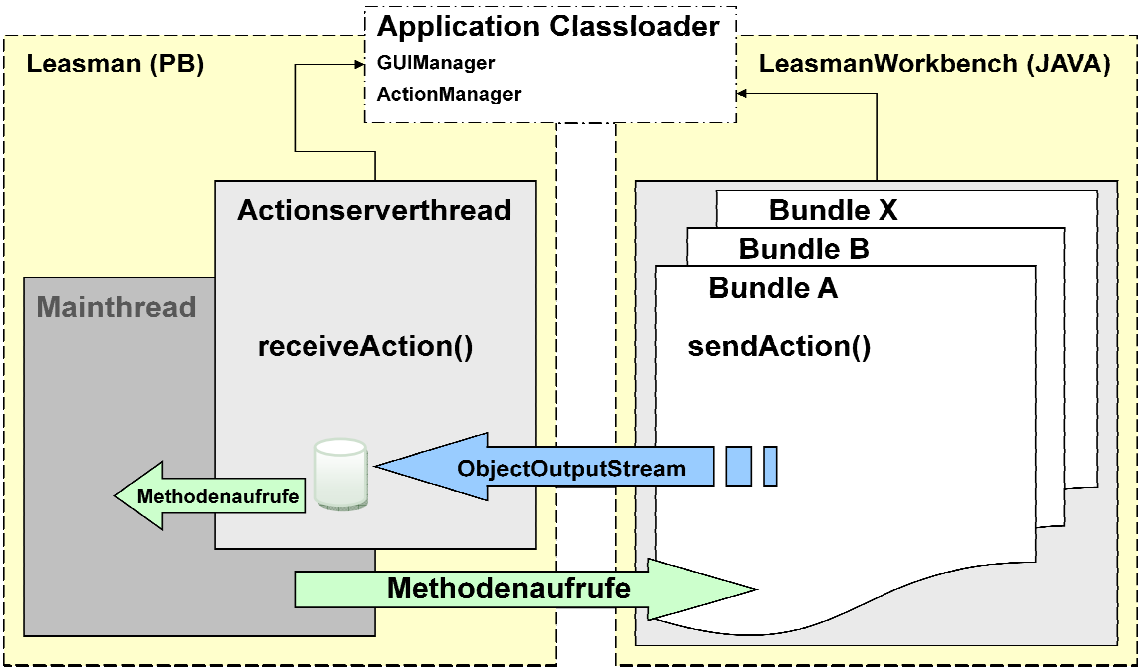


Abbildung 12: Kommunikationsmodell Actionserver

Bei der Recherche der Funktionalitäten im PowerBuilder, welche sich zu diesem Zweck einsetzen lassen, wurden zwei Möglichkeiten gefunden.

5.3.1. Timer gesteuertes Polling[21]

Eine der beiden Möglichkeiten ist ein Polling mittels Timerfunktion, bei dem innerhalb eines bestimmten Zeitintervalls über einen Funktionsaufruf in Java geprüft werden kann, ob eine Nachricht zur Abholung bereit steht. Bei dieser Methode kann es zu Zeitverzögerungen zwischen der Anforderung einer Aktion und dem tatsächlichen Ausführen dieser im PowerBuilder kommen, da das kleinste Intervall eine Sekunde beträgt. Außerdem belastet das Polling die Systemressourcen. Deshalb wird diese Möglichkeit ausgeschlossen und nicht weiter verfolgt.

[21] Zyklische Wertabfrage

5.3.2. Shared Objects

Die zweite Möglichkeit besteht in sogenannten Shared Objects. Damit lassen sich im PowerBuilder Objekte in separaten runtime Sessions laden. Diese Sessions sind im Grunde genommen eine weitere Instanz des PowerBuilders, in denen Objekte zur Ausführung gebracht werden und somit in einem eigenen Prozess ablaufen. Der Vorteil bei dieser Methode ist, dass diese, während sie blockieren, keine Systemressourcen verbrauchen und außerdem synchron ablaufen.

Für Shared Objects stehen im PowerBuilder drei Funktionen zur Verfügung:

- SharedObjectRegister():
  Öffnet weitere runtime Session und meldet Powerobjekte zur Ausführung in separaten Threads an.

- SharedObjectGet():
  Holt die Referenz eines angemeldeten Objektes, welches ausgeführt werden soll.

- SharedObjectUnregister():
  Entfernt ein Objekt aus dem Speicher und schließt die runtime Session bzw. den Thread.

5.3.3. Aufbau auf PowerBuilder-Seite

Um die Funktionalität von nebenläufigen Prozessen durch Shared Objects im PowerBuilder komfortabel nutzen zu können, bietet es sich an, ein Framework zu entwickeln, welches deren Verwendung vereinfacht.

Dieses Framework orientiert sich am Aufbau von Threads in Java. Hier wird ein Runnable übergeben, welches die Funktionalität, die nebenläufig ausgeführt werden soll, enthält. Diese Runnables werden im PowerBuilder als Shared Objects verwendet.

Das nachfolgende Klassendiagramm zeigt den Aufbau des Actionservers im PowerBuilder:

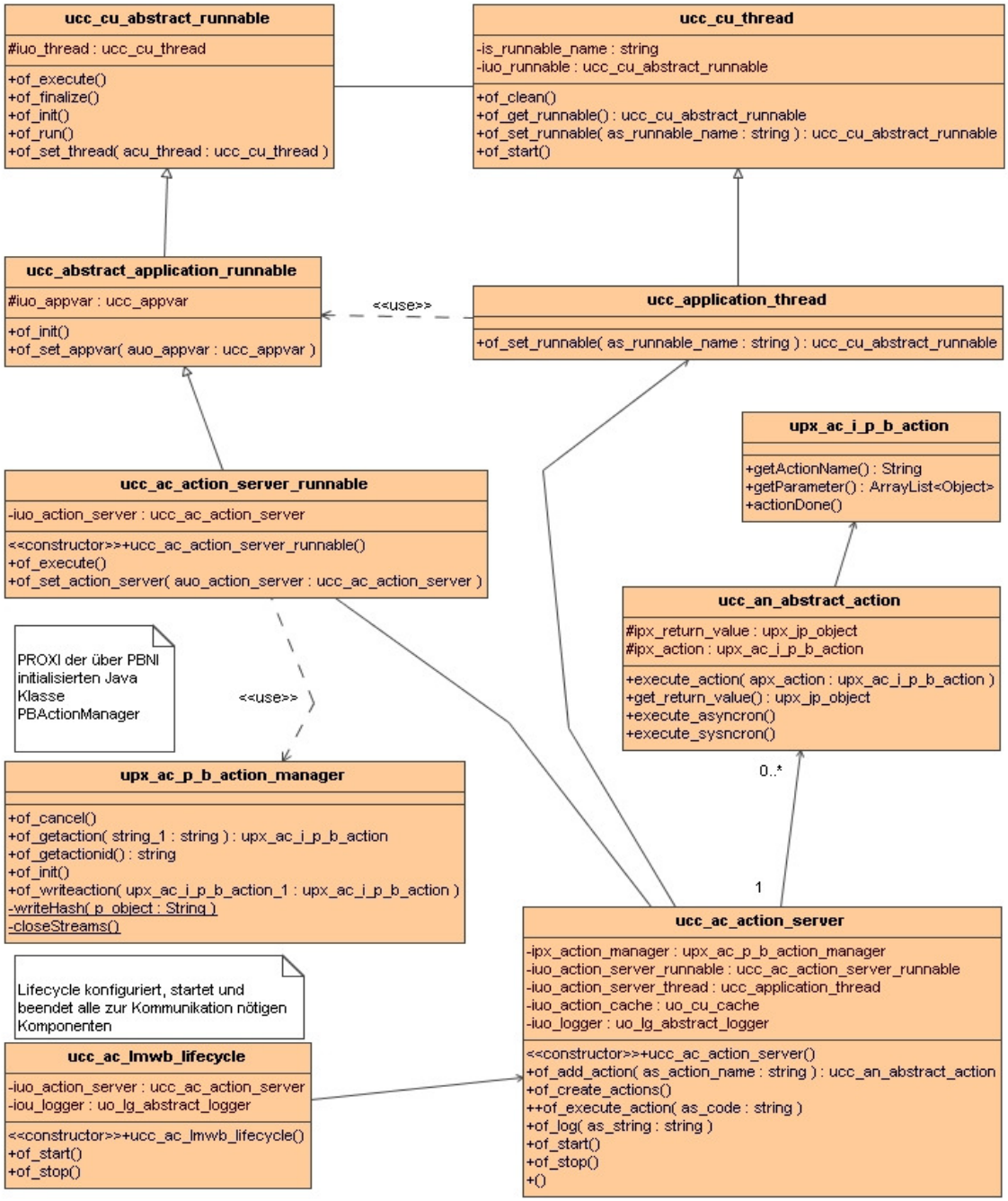


Abbildung 13: Klassendiagramm Actionserver PowerBuilder

Die Klasse *upx_ac_p_b_action_manager* ist ein Proxy, hinter dem sich ein Mapping auf eine Java Klasse verbirgt. Diese Klasse erzeugt die Streams, über welche der PowerBuilder mitgeteilt bekommt, dass eine neue Aktion ausgelöst wurde. Initialisiert wird diese Java Klasse vom PowerBuilder aus.

Der Actionserver wird im PowerBuilder ausgeführt und verwendet für die Übertragung der Actions Java Klassen. Für die Verwendung von Shared Objects wird ein Framework implementiert, welches es ermöglicht, Threads im PowerBuilder zu verwenden.

Die Obersten Klassen *ucc_cu_thread* und *ucc_cu_abstract_runnable* bilden die Basis für die Verwendung. Sie dienen dazu, Threads in allen mit PowerBuilder erstellten Programmen bei DELTA proveris einzusetzen. Die Threadklasse beinhaltet die Funktionalität um ein Runnable für parallele Verarbeitung anzumelden und den Thread zu starten. Die Runnableklasse liefert den abstrakten Rahmen für den Aufbau eines Runnables. Die Methoden *of_init()*, of_*execute()* und *of_finalize()* müssen überschrieben werden, um die Funktionalität zu definieren, welche bei *of_run()* ausgeführt wird. Die Methode *of_init()* dient zur Initialisierung der zu bearbeitenden Aufgabe. Hier können Vorkehrungen getroffen werden, die zur Ausführung benötigt werden. Mittels *of_execute()* wird die abzuarbeitende Funktionalität definiert. Über die Funktion *of_finalize()* ist es möglich, Änderungen wieder rückgängig zu machen, die für die Bearbeitung nötig waren oder Speicher frei zu geben, beziehungsweise Objekte zu zerstören und Ähnliches.

Eine Ableitung von diesen Klassen spezialisiert die Verwendung der Threads für den Einsatz in Leasman spezifischen Projekten. Hier wird eine globale Variable gesetzt, welche für die Verwendung im Leasman benötigt wird. Diese ist in der neuen PowerBuilder Runtime, in der der neue Thread läuft, noch nicht bekannt, da diese nicht von der Runtime automatisch übernommen wird.

Die Klasse *ucc_ac_action_server_runnable* ist die konkrete Implementierung des Shared Objects, welches die Funktionalität enthält, die im Actionserverthread abgearbeitet wird. Diese verwendet eine Instanz der Java Klasse PBActionManager und wartet an dessen ObjectInputStream auf eingehende Nachrichten.

Die Klasse ucc_*ac_action_server* ist die Implementierung des Action Servers selbst. Es werden die zugehörigen Actions, welche dann im PowerBuilder ausgeführt werden, erzeugt und in einem Cache abgelegt. Außerdem wird der Thread initialisiert und gestartet. Beim Beenden werden alle notwendigen Aktionen ausgelöst, damit der Thread beendet und die Streams geschlossen werden.

5.3.4. Aufbau auf Java-Seite

Auf Java-Seite befindet sich die Klasse *PBActionManager*. Diese ist mit seinen ObjectInput- und ObjectOutputStreams die direkte Verbindung zwischen Java und PowerBuilder. Auf PowerBuilder-Seite wird sie über den Proxy *upx_ac_p_b_action_manager* angesprochen. Auch hier wird ein kleines Framework erstellt, um die Verwendung von Actions in der LeasmanWorkbench einfacher zu gestalten.

Das nachfolgende Klassendiagramm zeigt den Aufbau des PBActionManagers auf Java-Seite:

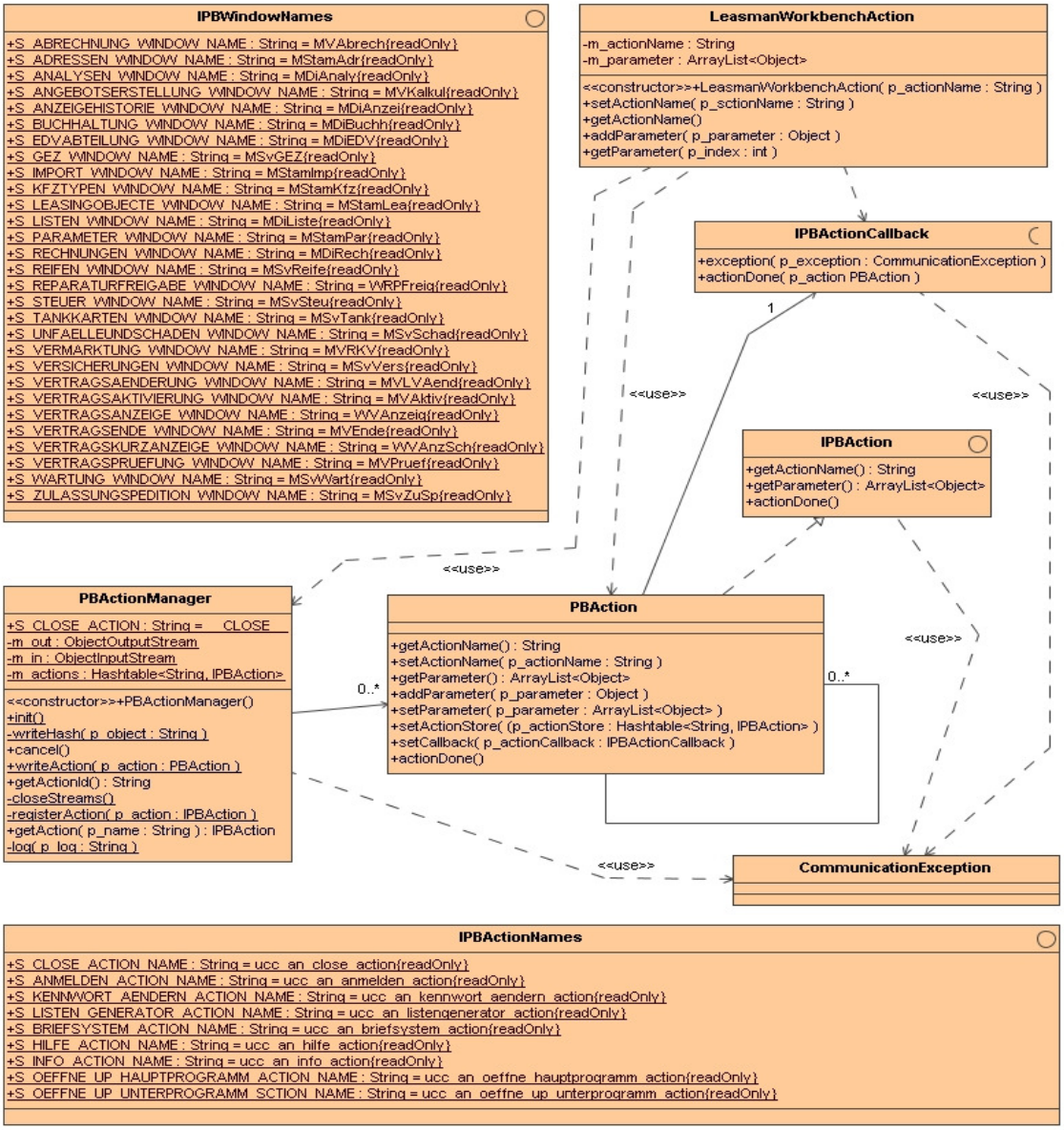


Abbildung 14: Klassendiagramm PBActionManager Java-Seite

Die Klasse *PBActionManager* zentralisiert die Kommunikation zwischen Java und PowerBuilder. Durch seine statischen Streams kann die Klasse von beiden Seiten gleichermaßen verwendet werden, da diese nach der Erzeugung dieser Klasse überall dieselben Instanzen sind.

Über den Aufruf der *init()* Methode vom PowerBuilder aus, werden die Streams erzeugt. Auf Java-Seite wird diese Klasse dann nur noch auf statischem Wege verwendet, so dass hier keine weitere Instanz angelegt werden muss. Über *writeAction()* wird eine Action an den PowerBuilder verschickt.

Zu diesem Zweck gibt es eine weitere Klasse *PBAction*, welche den Namen der Action enthält, die auf PowerBuilder-Seite ausgeführt werden soll und eine Liste der dafür nötigen Parameter.

Auf PowerBuilder-Seite müssen dabei nicht alle Methoden der Klasse sichtbar gemacht werden. Hierfür implementiert diese ein weiteres Interface *IPBAction*, welches auf PowerBuilder-Seite verwendet wird und nur die Methoden enthält, die dort benötigt werden. Die Klasse *PBAction* wird auf PowerBuilder-Seite nie initialisiert und es reicht daher aus, dort das Interface bekannt zu machen.

Da an den PowerBuilder lediglich der Hashcode der Action übertragen wird, nicht die Action selbst, enthält der Actionmanager eine Liste aller Actions, die noch zur Ausführung bereit stehen. Nachdem der PowerBuilder den Hashcode einer Action erhalten hat, wird von dort aus die zugehörige Action aus seinem Cache des Actionmanagers geholt und die darin enthaltenen Daten auf PowerBuilder-Seite ausgelesen. Eine Referenz auf diesen Cache enthält auch jede Action selbst, damit diese sich nach beenden der Übertragung selbständig wieder aus dem Cache entfernen kann.

Der Action ist ein *IPBActionCallback* zu übergeben, welches mit Funktionalität versehen werden kann, die ausgeführt wird, wenn die Action beendet wird und wie im Falle einer Exception zu verfahren ist.

Für die Ausführung einer Action im PowerBuilder, die über einen Button oder Menüpunkt gestartet wird, muss eine Klasse *LeasmanWorkbenchAction* erstellt werden, welche von Action des RCP Frameworks ableitet.

Dieser Action wird im Konstruktor der Name der auszuführenden Action im PowerBuilder übergeben. Beim Aktivieren löst diese *WorkbenchAction* dann die zugehörige Aktion im Leasman aus. Die Logik für den Aufruf der Action und wie nach beenden der Action verfahren wird, ist in dieser Klasse implementiert und für alle Aufrufe gleich.

Das Interface *IPBActionNames* enthält alle Namen der Actions, die im PowerBuilder implementiert sind. Das Interface *IPBWindowNames* enthält die Namen aller Einstiege, die vom Menü aus erreicht werden können. Diese werden in Verbindung mit den Actions *ucc_an_oeffne_hauptprogramm_action* und *ucc_an_oeffne_unterprogramm_action* auf PowerBuilder-Seite verwendet. Beide Actions können mehrfach verwendet werden, da zum Ausführen lediglich der Name des zu öffnenden Fensters übergeben wird.

5.3.5. Funktionsweise des Actionservers

Die Funktionsweise des Actionservers kann mit Sequenzdiagrammen sehr gut erklärt werden. Hier wird deutlich, wie die Aufrufkette und der Informationsfluss beim Starten des Actionservers und Ausführen einer Action ist.

Zur besseren Übersicht erfolgt eine Aufteilung des Sequenzdiagrammes in drei Teile. Das erste Diagramm verdeutlicht die Initialisierung des Serverthreads. Das zweite Diagramm beschreibt den Ablauf bei der Verarbeitung einer Action und das dritte Diagramm beschreibt das Beenden des Actionservers.

➢ Initialisierung des Actionserverthreads:

Die Initialisierung des Actionserverthreads erfolgt direkt nach dem Start des Leasman. Dabei werden die Actions erzeugt, welche im PowerBuilder ausgelöst werden können und in einem Cache abgelegt, aus dem diese bei Bedarf geladen werden. Danach erfolgt die Anmeldung des Actionserverrunnables als Shared Object und dessen Start. Sind diese Vorgänge abgearbeitet, wartet der Server auf Nachrichten am ObjectInputStream.

Das nachfolgende Sequenzdiagramm veranschaulicht die Initialisierung des Actionserverthreads:

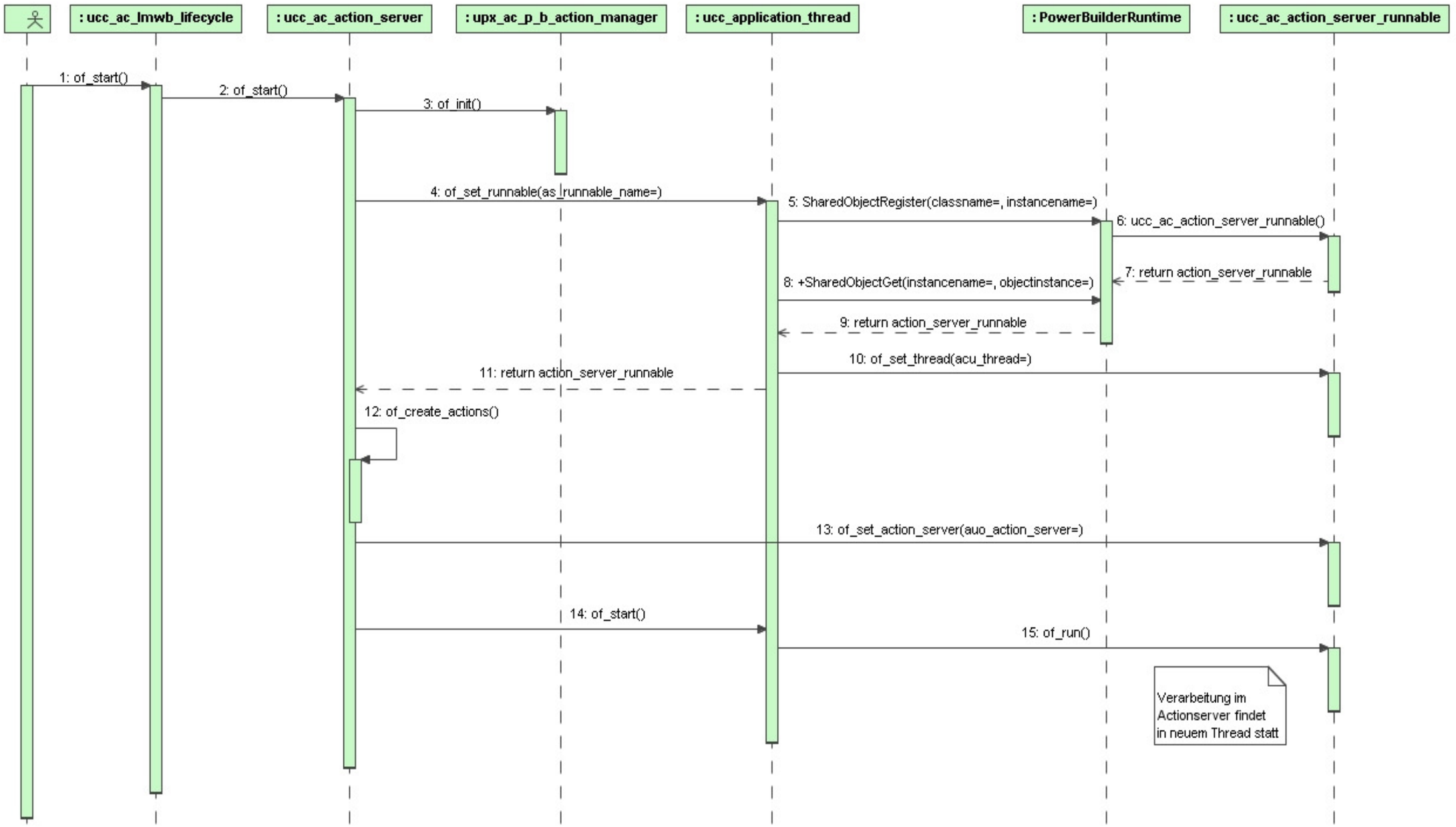


Abbildung 15: Sequenzdiagramm Initialisierung Actionserverthread

Das Diagramm in Abbildung 15 zeigt die Initialisierung des Serverthreads. Durch den Start des Actionservers wird die Java Klasse *PBActionManager* über den Java Proxy *upx_ac_p_b_action_manager* initialisiert und als erstes daran die *init()* Methode aufgerufen, welche die statischen Streams erzeugt.

Danach wird ein *ucc_application_thread* Objekt erzeugt, das den Namen des Runnables übergeben bekommt, welches als Shared Object geladen werden soll. Dies ist das Actionserverrunnable. Hier wird eine neue PowerBuilder Runtime geöffnet und das Runnable als Shared Object angemeldet. Abschließend wird eine Referenz auf das Shared Object bei der Runtime abgeholt und zurück gegeben. Vorher wird das Threadobjekt selbst hinein gesetzt, um nachdem das Runnable abgearbeitet ist am Threadobjekt *clean()* aufrufen zu können, was das Shared Object wieder abmeldet.

Nun erstellt der Actionserver die ausführbaren Actions und legt diese in einem Cache ab, um sie später laden zu können.

Danach wird das Actionserverobjekt selbst in das Runnable gesetzt um es dem Runnable zu ermöglichen an diesem Objekt Actions auszuführen.
Zu guter Letzt wird der Actionserverthread gestartet.

➢ Ausführung einer Action im Actionserver:

Nachdem der Actionserverthread gestartet ist, können Nachrichten an ihn versendet werden. Dies geschieht hauptsächlich von Java-Seite aus, da der Server für die Kommunikation von Java zu PowerBuilder entwickelt wurde. Es wäre auch möglich, Aktionen von PowerBuilder-Seite aus auszulösen, dies ist jedoch nicht sinnvoll, da dort Funktionsaufrufe direkt ausgeführt werden können.

Der Actionserver wartet in einer Schleife ständig auf eingehende Nachrichten am ObjectInputStream. Wenn eine Action ausgeführt wurde, kehrt er zum Ausgangspunkt zurück und wartet auf weitere eingehende Nachrichten.

Übertragen werden dabei nur die Hashcodes der *PBAction* Klassen aus Java, welcher gleichzeitig die Schlüssel sind, unter denen die Actions im Cache des Actionmanagers abgelegt sind.

Dieses Sequenzdiagramm beschreibt den Ablauf bei der Ausführung einer Action im Actionserver:

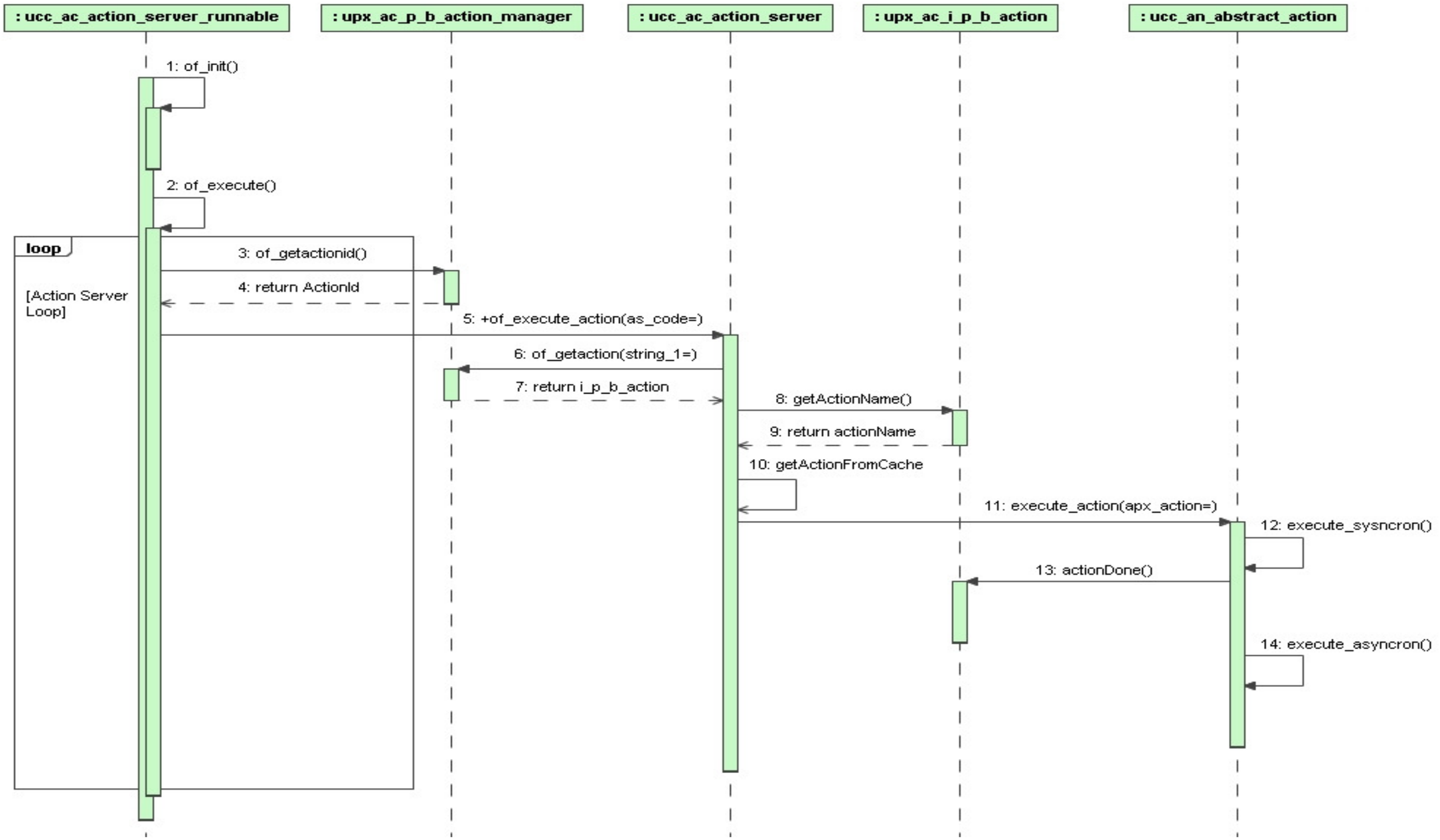


Abbildung 16: Sequenzdiagramm Ausführung einer Action im Actionserver

Abbildung 16 zeigt die Bearbeitung einer Action im Actionserver. Nachdem *of_start()* im Threadobjekt aufgerufen wurden, wird mit Aufruf der Funktion *of_run()* am Runnable die Funktionalität des Actionserverrunnables abgearbeitet.

Als erstes erfolgt der Aufruf der Funktion *of_init()*, hier wird jedoch keine Funktionalität implementiert, da dies für den Actionserver nicht benötigt wird.

Danach wird *of_execute()* aufgerufen. In dieser Methode befindet sich die Funktionalität des Actionservers. Hier wird zu Beginn eine Instanz von *upx_ac_p_b_action_manager* angelegt und an dessen ObjectInputStream in der Funktion *getActionId()* auf eine Nachricht gewartet.

Sobald auf Java-Seite etwas in den Stream geschrieben wurde, blockiert diese Funktion auf PowerBuilder-Seite nicht mehr und der Hashcode einer Action wird abgeholt. Dieser Hashcode wird an das Actionserverobjekt mit dem Aufruf Execute Action übergeben.

Der Actionserver holt sich über die Java Klasse *PBActionManager* die zum Hashcode passende Action aus dem Cache. Diese Action enthält Name und Parameter der Action, die auf PowerBuilder-Seite ausgeführt werden soll.

Anschließend holt sich der Actionserver auf PowerBuilder-Seite die Action mit diesem Namen aus seinem Cache und führt an dieser *execute_action()* aus. Dabei wird die aus Java kommende Action mit übergeben, um an dieser *actionDone()* aufrufen zu können.

Die Methode *execute_action()* arbeitet drei weitere private Methoden ab. Als erstes *execute_syncron(),* welche überschrieben werden muss, wenn auf PowerBuilder-Seite erst *actionDone()* aufgerufen werden soll, nachdem die Action bearbeitet wurde. Dies dient zu dem Zweck, dass auf Java-Seite Rückgabewerte dieser Action abgewartet werden müssen, oder zum Beispiel die Bearbeitung von anderen Aufgaben für diese Zeit blockiert werden soll. Danach wird an der Action von Java *actionDone()* aufgerufen. Hier wird auf Java-Seite diese Action aus dem Cache entfernt und am Callback ebenfalls *actionDone()* ausgeführt. Abschließend wird *execute_asyncron()* aufgerufen. Diese Methode würde überschrieben, wenn die Workbench nicht auf Rückgabewerte vom Leasman warten muss und dort weiter gearbeitet werden kann. Danach wartet der Actionserver wieder auf eingehende Nachrichten.

➢ Beenden des Actionservers:

Der Actionserver wird beendet, wenn der Leasman geschlossen wird. Dies geschieht durch den Aufruf von *of_stop()* im ucc_*ac_actionserver*. Diese Methode ruft wiederum im *PBActionManager* auf Java-Seite die Funktion *cancel()* auf.

Dies führt dazu, dass in den ObjectOutputStream der String „__CLOSE__“ geschrieben wird.

Auf PowerBuilder-Seite wartet der Actionserver noch über den Aufruf von *of_getactionid()* an dem Proxy *upx_ac_p_b_action_manger* der Java Klasse *PBActionManager* am ObjectInputStream auf Nachrichten.

In dieser Methode wird der String „__CLOSE__“ aus dem ObjectOutputStream gelesen. In diesem Falle wird die private Funktion *closeStreams()* aufgerufen, wodurch die beiden Streams geschlossen und auf *null* gesetzt werden.

Danach wird der String weiter an den PowerBuilder übergeben. Nun wird die Schleife im Actionserverrunnable, ohne den Aufruf *execute_action()*, verlassen und die Funktion *of_finalize()* aufgerufen, welche beim Actionserverrunnable jedoch leer ist, da keine Funktionalität implementiert wird, die ausgeführt werden müsste, wenn dieses Runnable beendet ist.

Zuletzt wird am Threadobjekt die Funktion *of_clean()* aufgerufen. In dieser Funktion wird das Shared Objekt des Actionserverrunnables aus der PowerBuilder Runtime entfernt und der Speicher freigegeben, sowie der Thread damit zerstört.

Die Abbildung 17 auf der nächsten Seite veranschaulicht diesen Vorgang in einem weiteren Sequenzdiagramm.

Dieses Sequenzdiagramm veranschaulicht den Ablauf beim Beenden des Actionservers:

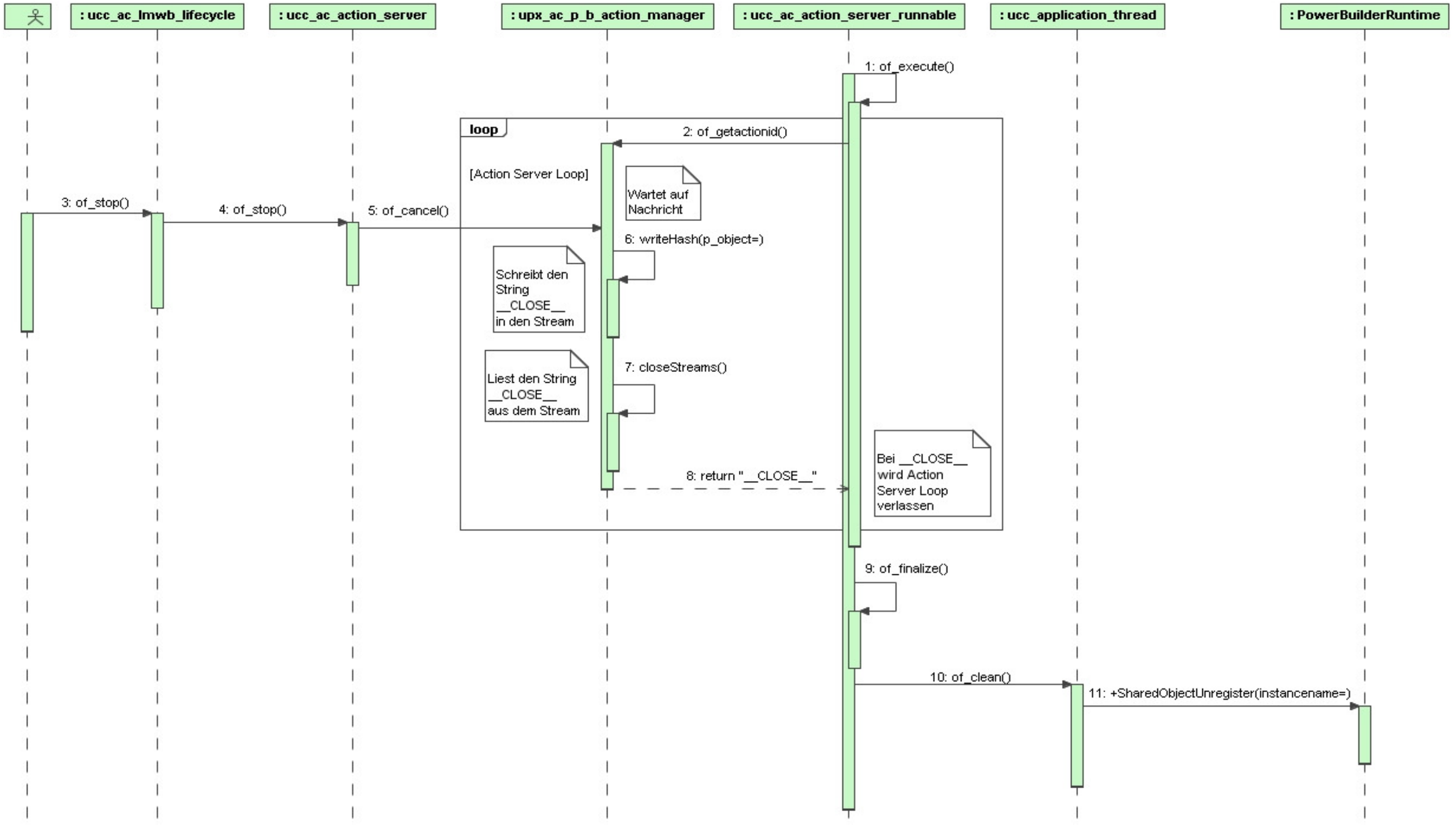


Abbildung 17: Sequenzdiagramm Beenden des Actionservers

5.4. Der GUIManager

Um die bestehenden PowerBuilder Komponenten weiter verwenden zu können, muss eine Möglichkeit geschaffen werden, diese Fenster dahingehen zu manipulieren, dass sie mit der Workbench zusammen arbeiten. Es soll dabei der Eindruck entstehen, dass es sich um eine eigenständige Anwendung handelt und möglichst nicht erkennbar sein, dass zwei Anwendungen nebeneinander ablaufen.

Dies ist über die Verwendung der Windows API möglich. Mit dieser lassen sich Fenster unter Microsoft Windows in vielfältiger Weise manipulieren. Dadurch geht zwar die Portabilität auf andere nicht Microsoft Betriebssysteme verloren, dies ist aber für die Zeit des hybriden Betriebes der Workbench nicht notwendig, denn mit PowerBuilder erstellte Anwendungen sind nur unter Windows Betriebssystemen lauffähig.

Der Zugriff auf die nativen Funktionen der Windows API wird über das Package *OS.** von SWT realisiert. Damit bietet SWT ein Framework an, welches über JNI Zugriff auf Windows API Funktionen verschafft. Die meisten dieser Funktionen benötigen als Parameter ein Handle[22] auf das zu manipulierende Fenster. Dies ist ein Integer wert, der jedem Fenster eindeutig zugeordnet ist.

Im PowerBuilder kann über eine Globale Funktion *Handle(window w)* ein Handle für ein bestimmtes Fenster abgefragt werden. Dieses Handle wird dann an den *GUIManager* übergeben und dieser übernimmt die weitere Manipulation.

Dadurch ist es möglich, Fenster nicht mehr im PowerBuilder anzuzeigen, sondern innerhalb eines SWT-Fensters, indem den Fenstern als Elternteil ein SWT-Fenster zugewiesen wird.

Darüber hinaus sind auch optische Veränderungen der Fenster erreichbar, dass zum Beispiel Menüleiste und Titelleiste der Fenster entfernt werden. Die Größe eines Fensters sowie die Position sind ebenfalls veränderbar.

[22] Einem Fenster eindeutig zugeordneter Integer Wert

5.4.1. Aufbau des GUIManagers

Über den *GUIManager* kann der LeasmanWorkbench mitgeteilt werden, dass graphische Elemente des PowerBuilders vorhanden sind, die verarbeitet werden müssen.

Der Aufruf der Methoden erfolgt direkt über die PBNI Schnittstelle an einem im PowerBuilder vorhandenen Proxy der Klasse.

Dieses Klassendiagramm zeigt den Aufbau des GUIManagers:

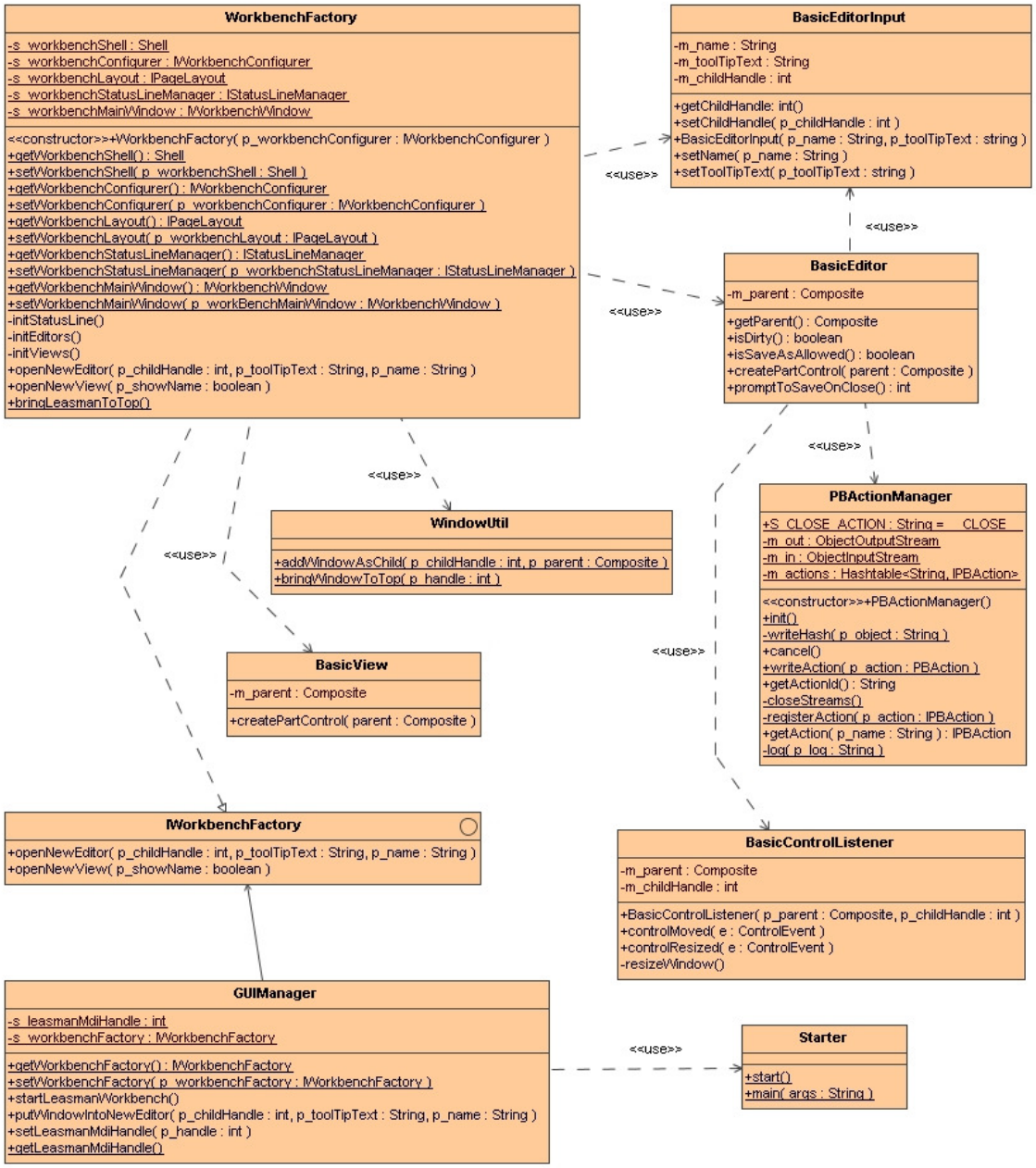


Abbildung 18: Klassendiagramm Aufbau GUIManager

Die Verarbeitung der Anfragen übernimmt nicht der *GUIManager* selbst, sondern die Klasse *WorkbenchFactory*, welche eine Implementierung des Interfaces *IWorkbenchFacory* ist. Dieses Interface wird vom *GUIManager* verwendet.

Die WorkbenchFactory verwendet die statische Klasse *WindowUtil*, in der Methoden vereinbart werden, die spezielle Funktionalitäten mittels Windows API ausführen, wie zum Beispiel ein Fenster einer Anwendung als Kind eines SWT-Fensters zu setzen.

Die Klasse *BaicView* ist eine Ableitung der Klasse *ViewPart* von RCP. Sie ist so aufgebaut, dass sie durch ihr Verhalten später mit PowerBuilder-Fenstern funktioniert. Im Moment werden BasicViews jedoch nicht verwendet, da es dafür noch keine konkrete Einsatzmöglichkeit gibt. Diese Klasse wird ebenfalls von der *WorkbenchFactory* verwendet, um PowerBuilder-Fenster als Kinder einer View anzuzeigen und der Workbench hinzuzufügen.

Die Verwendung des *BasicEditors* ist für Fenster vorgesehen, die im PowerBuilder innerhalb des Hauptfensters angezeigt werden. Sie sind von *EditorPart* des RCP Frameworks abgeleitet und in ihrem Verhalten so angepasst, dass sie mit PowerBuilder-Fenstern als Kinder harmonieren und sich beim Schließen des inneren Fensters entsprechend verhalten. Dieses Verhalten ist zum Beispiel erst sich selbst zuschließen, wenn die inneren Fenster geschlossen sind. Um Actions im PowerBuilder auslösen zu können, verwendet der *BasicEditor* den *PBActionManager*.

Zur Konfiguration und für verschiedene Eingangswerte wie Fenstername und Tooltiptext, initialisiert die *WorkbenchFactory* Ein *BasicEditorInput* Objekt, welches vom *BasicEditor* benötigt wird.

Dem *BasicEditor* wird noch ein *BasicControlListener* hinzugefügt, welcher auf Größenänderungen des Editorfensters reagiert und die Größe des internen Fensters anpasst.

Die *Starter* Klasse wird vom *GUIManager* verwendet um die Workbench auszuführen. Dies geschieht direkt nach der Anmeldung im Leasman. Dazu wird über den *GUIManager* die Funktion *StartLeasmanWorkbench()* aufgerufen, welche wiederum im *Starter* die Funktion *start()* ausführt.

5.4.2. Funktionsweise des GUIManagers

Nachdem auf PowerBuilder-Seite eine Instanz des *upx_ac_g_u_i_managers* angelegt wurde und damit eine Instanz des *GUIManagers* im Java Applicationclassloader vorhanden ist, wird von PowerBuilder-Seite aus das Handle des Leasmanhauptfensters in den *GUIManager* gesetzt. Dies wird später verwendet um mittels Windows API die Benutzeroberfläche des Leasman in den Vordergrund zu rücken und die Anwendung zu schließen. Auf Java-Seite kann diese Variable einfach abgeholt werden, da sie als statisch deklariert ist und somit während der gesamten Laufzeit in jeder Instanz dieselbe ist.

Als nächstes wird die LeasmanWorkbench von PowerBuilder aus gestartet. Das Leasmanhauptfenster selbst wird nach der Anmeldung auf Invisible gestellt, damit als neues Hauptfenster nur noch die Workbench sichtbar ist.

Sobald ein Child Window geöffnet wird, wird am *upx_ac_g_u_i_manager* die Funktion *of_putwindowintoneweditor()* aufgerufen und das Handle sowie der Name in der Titelleiste und der Tooltiptext für den BasicEditor übergeben. Dadurch führt der GUIManager auf Java-Seite die *openNewEditor()* Methode in der *Workbench-Factory* aus.

Nun wird ein *BasicEditor* Input Objekt erzeugt, welches beim Erzeugen eines neuen *BasicEditors* durch Eclipse RCP mit übergeben wird. In diesem werden die Parameter Handle, Name und Tooltiptext mit übergeben.

Das RCP Framework erzeugt nun eigenständig eine neue *BasicEditor* Instanz und setzt das *EditorInput* Objekt hinein. Kurz bevor der Editor vom RCP angezeigt wird, wird die Methode *createPartControl()* ausgeführt. In dieser Methode befindet sich die Logik für die Einbindung des PowerBuilder-Fensters. Über das *BasicEditorInput* Objekt kann auf die Parameter zugegriffen werden. Als erstes wird dann ein neuer Parent über das *WindowUtil* für das PowerBuilder-Fenster gesetzt und anschließen ein *BasicControlListener* hinzu gefügt, welcher beim Resize oder Verschieben des Fensters auch das inneren Fenster anpasst.

Dieses Sequenzdiagramm beschreibt den Ablauf beim Öffnen eines Child Windows im PowerBuilder:

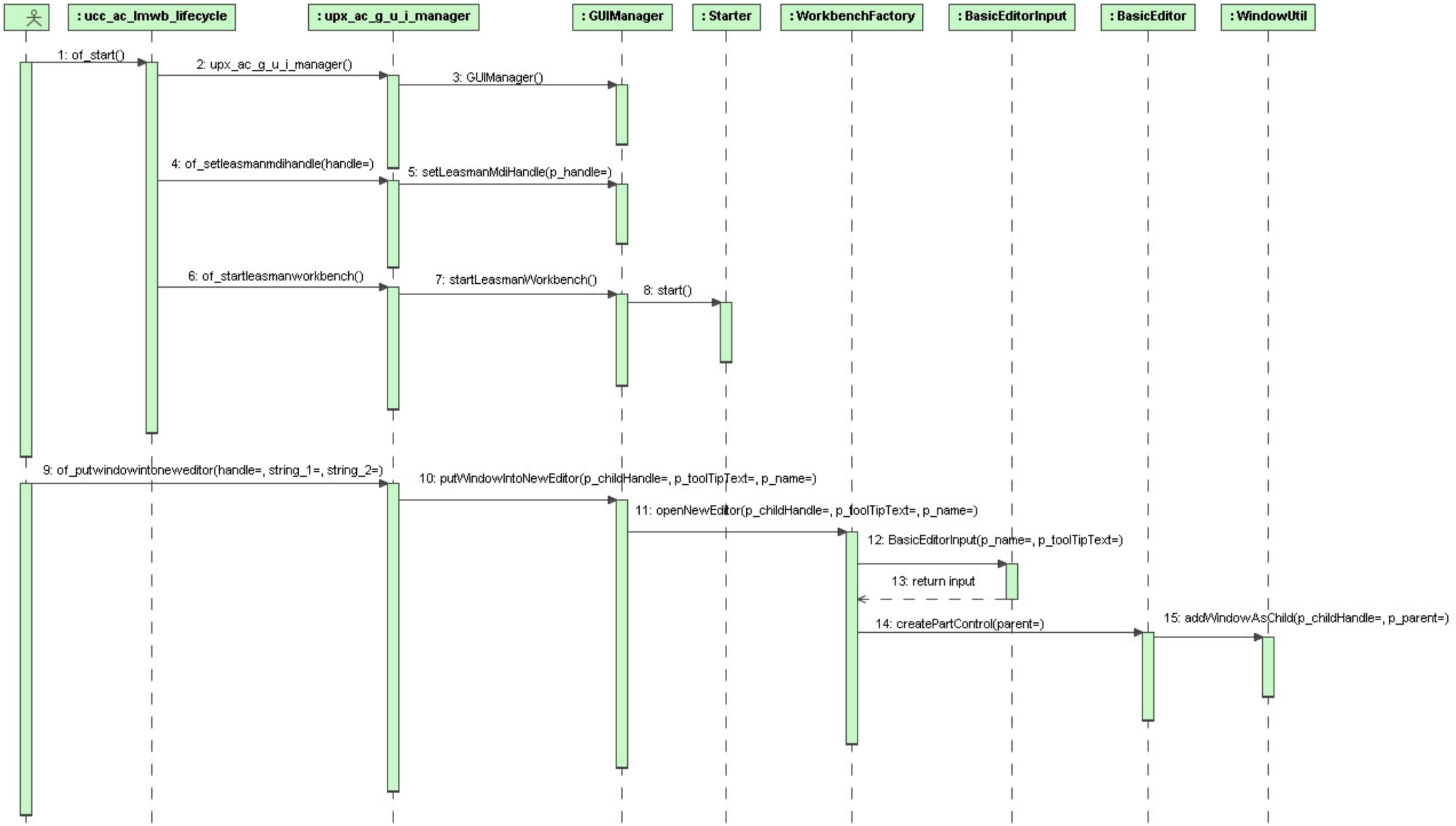


Abbildung 19: Öffnen eines Childfensters im PB und Integrierung in RCP Editor

5.5. Zentralisierung über Lifecycle

Über die Klasse *ucc_ac_lmwb_lifecycle* werden alle Vorbereitungen getroffen, die für die Ausführung der LeasmanWorkbench benötigt werden.

In der Methode *start()* wird als erstes der Actionserver initialisiert. Anschließend wird eine Instanz des *GUIManagers* erstellt und das Handle des Hauptfensters daran übergeben. Danach wird die LeasmanWorkbench gestartet.

Die Methode *stop()* beendet alle Komponenten.

Zukünftig können weiter für den Start notwendige Initialisierungen und Vorbereitungen in diesem Lifecycle eingebracht werden.

Kapitel 6

Prototypische Implementierung

Dieser Abschnitt beschreibt die Umsetzung des Konzeptes als Prototyp. Dabei wird als erstes allgemein besprochen, welche Art des Prototypings eingesetzt werden soll.

Des Weiteren werden wichtige Programmteile des Programmcodes erläutert und auf Probleme bei der Umsetzung eingegangen, die gelöst werden mussten.

Die Umsetzung des Prototyps bei DELTA proveris soll evolutiv geschehen. Nachdem das Konzept ausgearbeitet ist soll ein erster Prototyp erstellt werden, der dann sukzessive weiter entwickelt wird. Dies ermöglicht es die vielen Anpassungen, die für das Verhalten als eigenständige Anwendung nötig sind, nach und nach zu verbessern. Es wäre nicht sinnvoll, diese langwierig erarbeiteten Feinheiten zu verwerfen und von Grund auf neu zu implementieren. Dabei wird ständig auf einen sauberen Programmiersteel geachtet. Wenn die Kommunikation Fehlerfrei funktioniert kann mit der Überführung der bestehenden GUI begonnen werden.

6.1. Implementierung des Actionservers

Die Implementierung des Actionservers findet hauptsächlich im PowerBuilder statt. Hier muss das Framework für den Thread erstellt und die auszuführenden Actions angelegt werden. Außerdem müssen einige Stellen im Programmcode des Leasman angepasst werden, um den Actionserver zu starten, beziehungsweise zu stoppen.

Ein weiterer Teil der Implementierung findet in Java statt. Hier wird die Kernklasse des Actionservers implementiert, welche von Java-Seite aus benutzt wird, der PBActionManager. Außerdem müssen die Java Proxies erstellt werden, über welche der PowerBuilder auf die Java Klassen zugreifen kann, dafür gibt es ein Tool, mit welchem sich im PowerBuilder Proxies aus Java Klassen generieren lassen.

6.1.1. Die Thread Klassen

Das Framework für das Threading im PowerBuilder soll es vereinfachen, Shared Objects zu verwenden und Threads zu definieren, zu starten sowie zu beenden. Der Aufbau dieses Frameworks orientiert sich am Aufbau von Threads in Java.

Hierbei ist die Klasse *ucc_cu_thread* die Klasse über welche sich ein Thread Initialisieren und starten lässt. Über die Methode *of_set_runnable()* wird ein auszuführendes Runnable übergeben und als Shared Object angemeldet. Über die Funktion *of_start()* wird dieses Runnable anschließend ausgeführt.

Der Nachfolgende Programmausschnitt zeigt den Inhalt der Methode of_set_runnable():

```
1  int i
2  string shares[]
3
4  SharedObjectDirectory( shares )
5  i = UpperBound( shares)
6
7  i_runnable_name = as_runnable_name + string( i )
8
9  SharedObjectRegister( as_runnable_name, i_runnable_name )
10 SharedObjectGet( i_runnable_name, i_runnable )
11
12 i_runnable.of_set_thread( this )
13
14 return i_runnable
```

Abbildung 20: Programmausschnitt *of_set_runnable() ucc_cu_thread*

In Abbildung 20 ist die Funktion *of_set_runnable()* im Objekt *ucc_cu_thread* des Leasman zu sehen.

Das Array shares enthält alle als Shared Object registrierten Objekte im PowerBuilder. Es kann über die globale Funktion *SharedObjectDirectory()* abgefragt werden.

Um doppelte Schlüssel beim Registrieren neuer Objekte zu vermeiden, wird die Speicherstelle des höchsten Objektes ermittelt, um eins erhöht und als laufende Variable an den Schlüssel des neuen Shared Objektes angehängt. Doppelte Schlüssel können dadurch nicht entstehen, da sich die Speicherstelle eines Objektes nicht ändert, auch wenn Objekte entfernt werden. Das heißt, die Speicherstelle des nächsten Objektes ist immer höher, als die des zuletzt gespeicherten Objektes.

Danach wird das Shared Object angemeldet, aus dem Speicher geladen und zurück gegeben. Vorher wird noch der Thread selbst in das Runnable gesetzt, damit sich das Runnable beim Beenden selbst aus dem SharedObjectDirectory entfernen kann.

Diese Methode wird in der Klasse *ucc_application_thread* noch einmal überschrieben, um die globale Variable, die zur Ausführung des Leasman benötigt wird, in das Raunnable zu setzen.

Dieser Programmausschnitt zeigt die überschriebene Methode of_set_runnable() in der Klasse ucc_application_thread:

```
1  ucc_abstract_application_runnable luo_application_runnable
2
3  luo_application_runnable = super::of_set_runnable( as_runnable_name )
4
5  luo_application_runnable.of_set_appvar( guo_appvar )
6
7  return luo_application_runnable
```

Abbildung 21: Programmausschnitt *of_set_runnble()* in *ucc_application_thread*

6.1.2. Die Runnable Klassen

Die Klasse *ucc_cu_abstract_runnable* stellt nur das leere Gerüst für ein als Shared Object auszuführendes Runnable dar. Eine weitere Klasse *ucc_application_runnable* leitet von dieser ab. Hier wird die globale Variable guo_appvar gesetzt, die zur Ausführung des Leasman gebraucht wird.

Die Klasse *ucc_ac_action_server_runnable* leitet wiederum von dieser Klasse ab und bildet die konkrete Implementierung des im Actionserverthread ausgeführten Runnables. Hier ist die im Thread ablaufende Funktionalität enthalten. Diese ist in der Funktion *of_execute()* implementiert.

Die Methoden *of_init()* und *of_finalize()* bleiben leer, da hier keine Notwendigen Initialisierungen oder Speicherfreigaben nötig sind.

Der Folgende Programmcodeausschnitt zeigt die of_execute() Funktion des Actionserverrunnables:

```
1  int ret
2  upx_ac_p_b_action_manager lpx_action_manager
3
4  ucc_ad_jvm uo_ad_jvm
5  uo_ad_jvm = create ucc_ad_jvm
6  lpx_action_manager = uo_ad_jvm. of_new_instance( "upx_ac_p_b_action_manager" )
7
8  iuo_action_server.of_log( "Actionloop wird gestartet!" )
9
10 string ls_code
11 do while true
12   try
13     ls_code = lpx_action_manager.of_getActionId( )
14     iuo_action_server.of_log( "Actioncode empfangen: " + ls_code )
15   catch ( upx_ac_communication_exception e )
16     MessageBox( "FEHLER", "Fehler beim empfangen der ActionId!" )
17     continue
18   end try
19
20   if ls_code = "__CLOSE__" then
21     iuo_action_server.of_log( "Actionloop wird beendet!" )
22     exit
23   end if
24
25   iuo_action_server.of_execute_action( ls_code )
26 loop
```

Abbildung 22: Programmausschnitt Actionserverloop PowerBuilder

Die lokale Variable *lpx_action_manager* ist ein Proxy der Java Klasse *PBActionManager*. Es wird in der Java Maschine eine Instanz dieser Klasse erzeugt. Dazu wird die Klasse *uo_ad_jvm* verwendet, welche ein von DELTA proveris entwickeltes Framework ist, womit die im PowerBuilder angebotene Möglichkeit vereinfacht wird, Java Klassen zu instanziieren.

Danach wird eine Endlosschleife gestartet, welche an der Funktion *of_getActionId()* des Actionmanagerproxies blockiert, bis eine Nachricht in den Stream geschrieben wurde. Wenn eine Nachricht empfangen wurde, wird diese am Actionserver Objekt über *of_execute_action()* verarbeitet und die Action ausgeführt. Dann kehrt die Schleife wieder zum Ausgangspunkt zurück und wartet auf weitere Nachrichten. Wird der String „__CLOSE__“ empfangen, wird die Schleife verlassen.

6.1.3. Die Actionserver Klasse

Die Klassen *ucc_ac_action_server* ist die zentrale Klasse des Actionservers im PowerBuilder. Hier kann der Actionserver gestartet und gestoppt werden.

Beim Start wird der Thread initialisiert und alle zugehörigen Komponenten geladen. Beim Stoppen des Actionservers werden alle notwendigen Vorkehrungen getroffen, um den Server anzuhalten.

Zum Start des Actionservers gehört ebenfalls die Instanziierung der auf PowerBuilder-Seite ausführbaren Actions. Diese werden in der privaten Funktion *of_create_actions()* erzeugt und in einem Cache, unter ihrem eigenen Klassennamen als Schlüssel, abgelegt.

Vorerst werden nur sieben Actions angemeldet:

```
1  // New LeasmanWorkbench Actions
2  // Diensteactions
3  of_add_action( "ucc_an_listengenerator_action" )
4  of_add_action( "ucc_an_briefsystem_action" )
5
6  // Hilfeactions
7  of_add_action( "ucc_an_hilfe_action" )
8  of_add_action( "ucc_an_info_action" )
9
10 // OeffneUp Actions
11 of_add_action( "ucc_an_oeffne_up_hauptprogramm_action" )
12 of_add_action( "ucc_an_oeffne_up_unterprogramm_action" )
13
14 // Closeaction
15 of_add_action( "ucc_an_close_action" )
```

Abbildung 23: Ausführbare Actions im Actionserver

- ***ucc_an_listengenerator_action:***

  Öffnet den Listengenerator

- ***ucc_an_briefsystem_action:***

  Öffnet das Briefsystem

- ***ucc_an_hilfe_action:***

  Öffnet die Hilfe

- ***ucc_an_info_action:***

  Öffnet Informationen über Leasman

- ***ucc_an_oeffne_up_hauptprogramm_action:***

  Öffnet ein Programm als Hauptprogramm

- ***ucc_an_oeffne_up_unterprogramm_action:***

  Öffnet ein Programm als Unterprogramm

- ***ucc_an_close_action:***

  Schließt ein Fenster oder den Leasman

Für die Ausführung einer Action implementiert der Actionserver eine Methode *of_execute_action()*. Das Actionserverrunnable ruft diese Methode nach Erhalt einer Nachricht auf und übergibt den Hashcode der auf Java-Seite abgelegten Action. In dieser Methode wird über den *PBActionManager* auf Java-Seite die abgelegte Action abgeholt und der Name der auszuführenden Action auf PowerBuilder-Seite ausgelesen. Nun wird die zugehörige Action auf PowerBuilder-Seite aus dem Cache des Actionservers geladen und an dieser wiederum *of_execute_action()* aufgerufen. Dabei wird die Action von Java-Seite mit übergeben, um daran gegebenenfalls Parameter auslesen zu können oder Rückgabewerte abzulegen. Wenn keine Action auf PowerBuilder-Seite unter dem angeforderten Namen abgelegt ist, wird eine Messagebox mit der Meldung angezeigt, dass für die angeforderte Action keine Implementierung existiert.

Dieser Programmausschnitt zeigt den Inhalt der of_execute_action Methode der Klasse ucc_ac_action_server:

```
1  upx_ac_i_p_b_action lpxi_action
2  ucc_an_abstract_action luo_action
3
4  try
5    lpxi_action = ipx_action_manager.of_getAction( as_code )
6
7    luo_action = iuo_action_cache.get_value( lpxi_action.of_getActionName( ) )
8
9    if not isValid( luo_action ) or isNull( luo_action ) then
10     MessageBox( "Systemfehler", "Die Action "+lpxi_action.of_getActionName( ) +
11       " konnte nicht ausgeführt werden, da keine Inplementierung registriert wurde!" )
12     //hier sollte besser eine Exception übergeben werden
13     lpxi_action.of_actionDone( )
14     return
15   end if
16
17   luo_action.execute_action( lpxi_action )
18
19 catch( Throwable e )
20   MessageBox( "Systemfehler", "Kommunikationsfehler: "+e.getMessage( ) )
21   //hier sollte besser eine Exception übergeben werden
22   lpxi_action.of_actionDone( )
23 end try
```

Abbildung 24: Methode *of_execute_action()* der Klasse *ucc_ac_action_server*

6.1.4. Probleme bei der Implementierung

Bei der Implementierung des Actionservers traten keine größeren Probleme auf. Es wurde zuerst versucht, die Objekte der *PBAction* Klassen direkt zu übertragen. Dies war jedoch nicht möglich. Die Ursache dafür war nicht erkennbar. Vermutlich können Java Proxies keine serialisierten Objekte von Java-Seite zugewiesen werden, da ein Proxy ein PowerBuilder Objekt ist. Es ist nur möglich einfache Datentypen zu übertragen. Dazu zählen auf PowerBuilder-Seite auch Strings.

Daher wurde die Lösung entwickelt, nur den Schlüssel einer in einer Hashlist abgelegten Action zu übertragen und diese auf PowerBuilder-Seite über die PBNI Schnittstelle als Proxy über einen Methodenaufruf abzuholen.

6.2. Implementierung des GUIManagers

Über den *GUIManager* kann von PowerBuilder-Seite aus die Manipulation eines PowerBuilder-Fensters ausgelöst werden. Der Zugriff auf den *GUIManager* erfolgt über einen Java Proxy dieser Klasse.

Wird im Leasman ein Fenster geöffnet, welches durch die Windows API manipuliert werden muss, wird im *GUIManager* eine Methode aufgerufen, der das Handle des Fensters übergeben wird und die dann alles weitere übernimmt.

Dies geschieht in der Methode *wf_opensheet()* in der Klasse *w_mdi_rahmen*. Diese Klasse ist die Basisklasse des Hauptfensters im Leasman. Die Methode wird immer dann gerufen, wenn ein Fenster als child, also inneres Fenster geöffnet werden soll.

In der Klasse *w_mdi_applikation* wird diese Methode nun überschrieben und nach dem Aufruf der Funktionalität der Elternklasse, weitere Funktionalität implementiert, welche das Fenster über den *GUIManager* in einem neuen Editor der Leasman Workbench öffnet. Dafür verwendet die Klasse *GUIManager* die Klasse *WindowUtil* und *WorkbenchFactory*.

Dieser Programmausschnitt zeigt die Methode wf_opensheet():

```
1  int ret
2  ret = super::wf_opensheet( aui_prozent, astr_oeffne_up )
3
4  upx_ac_g_u_i_manager gui_manager
5  gui_manager= guo_appvar.of_getobject( "GUI_MANAGER")
6
7  if isValid(gui_manager) and not isNull(gui_manager) and isValid( iw_child ) and not isNull( iw_child ) then
8    guo_appvar.of_add( string( Handle( iw_child ) ), iw_child )
9    gui_manager.of_putwindowintoneweditor( Handle( iw_child ), iw_child.title , iw_child.title )
10 end if
11
12 return ret
```

Abbildung 25: Funktion *wf_opensheet()* in *w_mdi_applikation*

6.2.1. Das WindowUtil

Das *WindowUtil* ist vorgesehen um Funktionalitäten in einer Klasse zusammen zu fassen, welche über die Windows API realisiert werden. Meist sind mehrere verschiedene Aufrufe der Windows API notwendig, um einen gewünschten Effekt zu erzielen. Damit dies nicht an vielen verschiedenen Stellen implementiert werden muss, sind statische Methoden vorgesehen, die von allen Stellen im Programmcode aus aufgerufen werden können.

Das *WindowUtil* enthält momentan nur zwei Methoden. Die erste Methode fügt einem SWT-Fenster ein Fenster als Kind hinzu. Dazu wird das Handle des Kindes übergeben und das Composite des Elternfensters, zu dem das Kind hinzugefügt werden soll. Hier wird zuerst über die API Funktion *SetWindowLong()* das Aussehen des Fensters verändert. Dabei werden Menüleiste und Titelleiste entfernt. Als nächstes wird über die Funktion *SetParent()* ein neues Elternfenster für das Kind gesetzt.

Die zweite Funktion die das *WindowUtil* enthält, bringt das Fenster eines zugehörigen Handles in den Vordergrund. Dies wird benötigt, um nach Ausführen einer Action den Leasman in den Vordergrund zu rücken, damit seine Fenster vor der LeasmanWorkbench liegen.

Dieser Programmausschnitt zeigt die Java Klasse WindowUtil:

```
1  package framework;
2
3  import logger.Logger;
7
8  @SuppressWarnings("restriction")
9  public class WindowUtil
10 {
11     public static void addWindowAsChild(int p_childHandle, Composite p_parent)
12     {
13         try {
14             OS.SetWindowLong(p_childHandle, OS.GWL_STYLE, OS.GW_CHILD);
15             OS.SetParent(p_childHandle, p_parent.handle);
16         } catch (Throwable e) {
17             Logger.log(e);
18             e.printStackTrace();
19         }
20     }
21
22     public static void bringWindowToTop(int p_handle)
23     {
24         OS.BringWindowToTop(p_handle);
25     }
26 }
```

Abbildung 26: Java Klasse *WindowUtil*

6.2.2. Die WorkbenchFactory

Die *WorkbenchFactory* wird vom *GUIManager* verwendet, um einen neuen BasicEditor zu öffnen und ein Child-Fenster darin anzuzeigen. Die Klasse kann aber auch von Java-Seite aus verwendet werden, um sich wichtige Objekte der Workbench zu holen. Außerdem kann daran die Methode *bringLeasmanToTop()* aufgerufen werden, wodurch die Fenster des bisherigen Leasman in den Vordergrund gerückt werden. Diese Methode wird oft benötigt, damit keine Leasmanfenster von anderen Fenstern verdeckt werden. Weiterhin wird bei der Erzeugung der *WorkbenchFactory* die Satusleiste initialisiert um die Uhrzeit anzuzeigen.

Da alle Felder der WorkbenchFactory statisch sind, ändern sich deren Referenzen über die gesamte Laufzeit der Workbench nicht. Dies vereinfacht den Zugriff auf Instanzen, da dieser über die Klasse *PlatformUI* des RCP Frameworks, womit man Zugriff auf alle wichtigen Objekte der Workbench hat, sehr umständlich ist.

Die *WorkbenchFactory* wird beim Start der Workbench in den *GUIManager* gesetzt.

Die Folgende Abbildung zeigt die Funktion openNewEditor() der WorkbenchFactory:

```
126    /**
127     * Öffnet einen neuen BasicEditor für ein PowerBuilder Fenster.
128     */
129    public void openNewEditor(final int p_childHandle, final String p_toolTipText, final String p_name)
130    {
131        if(!s_workbenchShell.isDisposed())
132        {
133            s_workbenchShell.getDisplay().asyncExec(new Runnable() {
134
135                public void run(){
136
137                    @SuppressWarnings("unused")
138                    BasicEditor basicEditor = null;
139                    BasicEditorInput basicEditorInput = new BasicEditorInput(p_name, p_toolTipText);
140                    basicEditorInput.setChildHandle(p_childHandle);
141
142                    try {
143                        basicEditor = (BasicEditor)s_workbenchMainWindow.getActivePage()
144                            .openEditor(basicEditorInput, BasicEditor.ID, true);
145                    } catch (Throwable e) {
146                        Logger.log(e);
147                        e.printStackTrace();
148                    }
149                }
150            });
151        }
152    }
```

Abbildung 27: Klasse *WorkbenchFactory openNewEditor()*

6.2.3. Der BasicEditor

Im *BasicEditor* werden die im PowerBuilder geöffneten Child Windows angezeigt. Nachdem im *GUIManager* die Funktion *putWindowIntoNewEditor()* aufgerufen und somit über die *WorkbenchFactory* ein neuer *BasicEditor* geöffnet wurde, wird automatisch beim Initialisieren des Editors das PowerBuilder-Fenster hinzugefügt. Dazu wird dem *BasicEditor* mit seinem *BasicEditorInput* Objekt, welches von *EditorInput* ableitet und immer beim Öffnen mit übergeben werden muss, das Handle des PowerBuilder-Fensters übergeben.

Von Eclipse RCP wird beim Initialisieren des Editors automatisch die Methode *createPartControl()* aufgerufen. In dieser Methode wird das Handle des PowerBuilder-Fensters verwendet, um diesem als neuen Parent den Editor zuzuweisen. Dadurch wird das PowerBuilder-Fenster innerhalb des Editors angezeigt.

Um das innere Fenster immer innerhalb der Grenzen des Editorfensters anzuzeigen, wird ein *ControlListener* dem Editor-Fenster hinzu gefügt. Dieser setzt beim Verschieben oder Resizen des Editors die Größe des inneren PowerBuilder-Fensters, sowie seine Position neu. Dazu verwendet er eine Windows API Funktion.

Da das PowerBuilder-Fenster innerhalb jederzeit durch Abbruch oder Speichern geschlossen werden kann und der Editor dann leer offen bleiben würde, muss noch ein Paint Listener hinzu gefügt werden. Sobald das Paint Event am Editor ausgelöst wird, wird versucht das innere PowerBuilder-Fenster über die Windows API in den Vordergrund zu legen. Schlägt die Windows API Funktion fehl, wird davon ausgegangen, dass das innere Fenster geschlossen wurde. Dann kann auch der Editor ohne Speichern geschlossen werden.

Dieser Programmausschnitt zeigt die Methode createPartControl() im BasicEditor:

```
102    @Override
103    public void createPartControl(Composite parent)
104    {
105        try {
106            final BasicEditorInput input = (BasicEditorInput)getEditorInput();
107
108            setPartName(input.getName());
109
110            WindowUtil.addWindowAsChild(input.getChildHandle(), parent);
111
112            parent.addListener(SWT.Paint, new Listener() {
113                public void handleEvent(Event p_event) {
114                    Logger.log("Paint Event im BasicEditor ausgelöst. " +
115                            "Editor Fenster wird geschlossen.");
116                    if(!OS.BringWindowToTop(input.getChildHandle())) {
117                        getSite().getWorkbenchWindow().getActivePage()
118                            .closeEditor(BasicEditor.this, false);
119                    }
120                }
121            });
122        } catch (Throwable e) {
123            Logger.log(e);
124        }
125    }
```

Abbildung 28: Methode *createPartControl()* im *BasicEditor*

Ebenso wichtig ist es, dem inneren PowerBuilder-Fenster mitzuteilen, dass der äußere Editor geschlossen werden soll. Dann muss erst das innere Fenster geschlossen werden und anschließend der Editor. Vorher müssen die Eingaben im Inneren gespeichert werden. Wird der Speicherprozess abgebrochen, muss auch das äußere Editorfenster geöffnet bleiben.

Diese Logik befindet sich in der Methode *doSave()* des BasicEditors. Sie wird vom Eclipse RCP Framework ausgelöst, wenn der Editor geschlossen werden soll.

Dazu wird erst die Workbench disabled, da sich beim Speichern des PowerBuilder-Fensters ein modaler Dialog öffnet, während dessen keine weiteren Eingaben vorgenommen werden dürfen. Zum Auslösen des Vorgangs wird eine Close Action an den PowerBuilder gesendet. Diese enthält das Handle des Fensters, welches geschlossen werden soll. Im PowerBuilder wurden alle offenen Fenster in einem Cache mit dem Handle als Schlüssel abgelegt. Nun wird das Fenster aus diesem Cache gelesen und daran das Close Event ausgelöst.

Der Rückgabewert dieses Close Events wird ausgewertet und in die Parameterliste der gesendeten Action von Java geschrieben. Danach wird *actionDone()* von PowerBuilder-Seite aufgerufen und dieser Rückgabewert auf Java-Seite ausgewertet. Ist dieser Wert true, dann wird das Editor-Fenster geschlossen. Die Workbench wird wieder enabled und erhält den Focus. Wenn der Wert false ist, dann wird der Editor nicht geschlossen, denn das innere Fenster ist noch offen.

Dieser Programmausschnitt zeigt die Methode doSave() des BasicEditors:

```
35    @Override
36    public void doSave(final IProgressMonitor monitor) {
37
38        WorkbenchFactory.getWorkbenchShell().setEnabled(false);
39        WorkbenchFactory.bringLeasmanToTop();
40
41        PBAction action = new PBAction();
42        action.setActionName(IPBActionNames.S_CLOSE_ACTION_NAME);
43        action.addParameter(((BasicEditorInput)getEditorInput()).getChildHandle());
44        action.setCallback(new IPBActionCallback() {
45
46            public void exception(CommunicationException p_exception) {
47
48            }
49
50            public void actionDone(final PBAction p_action) {
51                Runnable r = new Runnable() {
52                    public void run() {
53                        WorkbenchFactory.getWorkbenchShell().setEnabled(true);
54                        WorkbenchFactory.getWorkbenchShell().setFocus();
55                        // Wenn das Fenster im Powerbuilder geschlossen wurde.
56                        boolean isPBWindowClose = (Boolean)p_action.getParameter().get(1);
57                        if(isPBWindowClose) {
58                            // Kann das EditorFenster entgueltig geschlossen werden.
59                            WorkbenchFactory.getWorkbenchMainWindow().getActivePage()
60                                .closeEditor(BasicEditor.this, false);
61                        }
62                    }
63                };
64
65                if(!WorkbenchFactory.getWorkbenchShell().isDisposed())
66                {
67                    WorkbenchFactory.getWorkbenchShell().getDisplay().asyncExec(r);
68                }
69            }
70        });
71
72        try {
73            PBActionManager.writeAction(action);
74        } catch (CommunicationException e) {
75            e.printStackTrace();
76        }
77    }
```

Abbildung 29: Methode *doSave()* im *BasicEditor*

6.2.4. Der BasicControlListener

Der *BasicControlListener* verwaltet den Inhalt des Controls, an welchem er hinzugefügt wird. Dies kann ein Editor, eine View oder ein anderes Composite sein.

Dafür wird das Interface *ControlListener* implementiert, welches die beiden Methoden *controlMoved()* und *controlResized()* vereinbart. In beiden Methoden wird eine private Methode aufgerufen, welche den Quellcode enthält, der das innere PowerBuilder-Fenster über die Windows API an das Elternfenster anpasst. Dafür wird dem *BasicControlListener* das Elterncomposite und das Kindhandle im Konstruktor übergeben. Sobald das Elternfenster resized oder verschoben wird, wird auch das innere Fenster angepasst. Dies wird über die API Funktion *setWindowPlacement()* realisiert. Die Größe des Elternfensters wird aus dem übergebenen Composite ausgelesen.

Dieser Programmausschnitt zeigt die resizeWindow() Funktion, welche im BasicEditorListener implementiert ist:

```
34    private void resizeWindow()
35    {
36        try {
37            WINDOWPLACEMENT lpwndpl = new WINDOWPLACEMENT ();
38            lpwndpl.length = WINDOWPLACEMENT.sizeof;
39            lpwndpl.showCmd = OS.SW_RESTORE;
40            Rectangle r = m_parent.getBounds();
41            lpwndpl.left = r.x;
42            lpwndpl.top = r.y;
43            lpwndpl.right = r.x + r.width;
44            lpwndpl.bottom = r.y + r.height;
45            OS.SetWindowPlacement(m_childHandle, lpwndpl);
46        } catch (Throwable e) {
47            Logger.log(e);
48            e.printStackTrace();
49        }
50    }
```

Abbildung 30: Funktion *resizeWindow()* im *BasicControlListener*

6.3. Die Starter Klasse

Über die *Starter* Klasse kann die LeasmanWorkbench entweder von der Kommandozeile aus oder aus dem Quellcode heraus gestartet werden. Beim Export der RCP Anwendung wird eine Anwendung erzeugt, die über eine EXE gestartet wird. Dadurch wird Eclipse nur mit den Kernplugins geladen und dem Plugin der LeasmanWorkbench.

Da die Workbench allerdings aus dem PowerBuilder heraus gestartet werden muss, ist diese Starter Klasse notwendig.

Diese Abbildung zeigt den Programmcode der Starter Klasse:

```
1  package execute;
2  import org.eclipse.core.runtime.adaptor.EclipseStarter;
4
5  @SuppressWarnings("restriction")
6  public class Starter
7  {
8      private static final String LEASMAN_WORKBENCH_HOME = "W:/root/lm/pb/work/l17/leasman_l17/LMWB/";
9
10     public static void start() {
11
12         Thread t = new Thread(){
13
14             @Override
15             public void run() {
16
17                 EclipseStarter.debug = true;
18                 //Set the right Paths for the RCP Application
19                 System.setProperty(EclipseStarter.PROP_INSTALL_AREA, LEASMAN_WORKBENCH_HOME);
20                 System.setProperty(EclipseStarter.PROP_SYSPATH, LEASMAN_WORKBENCH_HOME + "plugins/");
21
22                 //Hier sollte ev. user.home verwendet werden
23                 System.setProperty(LocationManager.PROP_INSTANCE_AREA, LEASMAN_WORKBENCH_HOME);
24                 System.setProperty(LocationManager.PROP_CONFIG_AREA, LEASMAN_WORKBENCH_HOME + "configuration/");
25
26                 // Der ParentClassLoader soll der ApplicationClassLoader sein.
27                 System.getProperties().put("osgi.parentClassloader", "app");
28                 // Alle Klassen aus diesem Package werden in den ApplicationClassLoader geladen.
29                 System.getProperties().put("org.osgi.framework.bootdelegation", "adapter");
30
31                 try {
32                     EclipseStarter.startup(new String[]{}, null);
33                     EclipseStarter.run(new String[]{});
34                     EclipseStarter.shutdown();
35                 } catch (Exception e) {
36                     // Noch keine Idee wie hier angemessen reagiert werden kann
37                     e.printStackTrace();
38                 }
39             }
40         };
41
42         t.start();
43     }
44
45     public static void main(String[] args) throws Exception
46     {
47         start();
48     }
49 }
```

Abbildung 31: Starter Klasse

In der *Starter* Klasse werden einige Systemproperties gesetzt, damit die Workbench ausgeführt werden kann.

Es wird zuerst das Homeverzeichnis der exportierten LeasmanWorkbench gesetzt und dann das Verzeichnis, in dem sich die zu ladenden Plugins befinden. Danach wird das Verzeichnis zum Speichern von Einstellungen und der Konfiguration gesetzt.

Der nächste Teil ist sehr wichtig: In Zeile 27 von Abbildung 31 wird als Elternclassloader der Applicationclassloader gesetzt. Dies bedeutet, dass alle Systemklassen wie *java.util.** in den Applicationclassloader geladen werden. In Zeile 29 kann festgelegt werden, welche Packages in den Elternclassloader und nicht in den Classloader des Plugins geladen werden sollen. Hier wird das Package adapter angegeben. Dadurch ist der *PBActionManager* und der *GUIManager* sowohl auf PowerBuilder-Seite als auch auf Java-Seite im Selben Classloader geladen. Dies ist notwendig, damit beide Seiten auf dieselben statischen Felder zugreifen.

6.4. Java Klassen im Leasman laden

Damit der PowerBuilder die Klassen verwenden kann, müssen ihm diese bekannt gemacht werden. Dafür gibt es die Datei pls.ini, über welche der Leasman konfiguriert werden kann. Hier werden Datenbankverbindung und ähnliches hinterlegt. Diese Konfigurationsdatei wurde bei DELTA proveris bereits früher angelegt.

Unter anderem können dort JAR Dateien angegeben werden, die vom PowerBuilder zur Laufzeit geladen werden sollen. Hier wird ein *GUIManager.jar* File angegeben, welches die vom PowerBuilder benötigten Klassen für den *GUIManager* und *Actionserver* enthält. Die zugehörigen Packages sind *adapter*, *execute*, *framework* und *logger*.

Außerdem benötigt der PowerBuilder noch die JAR Datei org.eclipse.osgi_xxxx.jar, damit die Workbench aus seinem Classloader heraus geladen werden kann, da diese Klassen von der Starter Klasse verwendet werden. Das xxxx am Ende des Namens steht für die Version des Files.

6.5. Probleme bei der Implementierung

Probleme bei der Implementierung traten bei der Zusammenführung der beiden Classloader von PowerBuilder und LeasmanWorkbench auf. Da bestimmte Klassen auf beiden Seiten benutzt werden müssen, müssen sie auch auf beiden Seiten in den Classloader geladen werden.

Beim Laden des *GUIManagers* im PowerBuilder wurde auch die *WorkbenchFactory* mit geladen, da sie von diesem benötigt wird. Die *WorkbenchFactory* verwendet aber Klassen des RCP Frameworks, welche im PowerBuilder nicht bekannt sind. Dadurch kam es zu einer *NoClassDefinitionFoundException*.

Nun hätten alle Klassen des RCP Frameworks im PowerBuilder geladen werden müssen. Dies konnte dadurch umgangen werden, dass der *GUIManager* ein Interface der *WorkbenchFactory* erhält, in dem nur Methoden benutzt werden, die entweder primitive Datentypen oder Klassen verwenden, die dem PowerBuilder bekannt sind. Dadurch wird das Interface der *WorkbenchFactory* im PowerBuilder ohne Probleme geladen.

Die konkrete Implementierung der *WorkbenchFactory* wird im Classloader der LeasmanWorkbench geladen und beim Start statisch von Java-Seite aus als Interface in den *GUIManager* gesetzt.

Dadurch kann von PowerBuilder-Seite aus ein Aufruf über den Applicationclassloader im *GUIManager* eine Funktion im Classloader der Workbench ausführen, wo alle benötigten Klassen bekannt und initialisiert sind.

Es ist auch nicht möglich, Klassen im Applicationclassloader und im Classloader der Workbench noch einmal zu laden. Dadurch kommt es zu einer *DuplicateClassLoadException*, da durch die Hierarchie die Klassen aus dem Applicationclassloader bereits bekannt sind. Hingegen von beiden Seiten im Applicationclassloader geladene Klassen, lösen keine Konflikte aus. Die Klasse wird dann nur beim ersten Mal geladen.

6.6. Testen des Prototyps

Der Stabilitätstest des Prototyps kann über die Testabteilung der DELTA proveris AG geschehen. Für den Test des Leasman stehen bereits zahlreiche Testszenarien zur Verfügung. Diese müssen in der LeasmanWorkbench abgearbeitet werden um einen möglichst großen Teil der Oberfläche zu testen. Fehler, die dabei auftreten, müssen erfasst werden, damit sie von der Softwareentwicklungsabteilung beseitigt werden können.

Fehler an der graphischen Benutzeroberfläche, die zum Beispiel Resize, Scrolling, Focus oder Anordnung der PowerBuilder Komponenten betreffen, müssen von Hand erarbeitet und erfasst werden.

Der Test der Stabilität der Kommunikation kann dahingehend automatisiert werden, dass Testszenarien einmal per Hand durchgeführt und diese dann mittels eines Robots mehrmals wiederholt werden. Bei diesen Robots werden Eingaben auf dem Bildschirm aufgezeichnet und die Abläufe gespeichert. Später können diese automatisch ablaufen.

Es sollte insbesondere getestet werden, wie sich das System verhält, wenn der Leasman oft geöffnet und geschlossen wird, sowie viele Anfragen an den Actionserver gestellt werden.

Kapitel 7
Vorgehen bei der weiteren Überführung

Dieser Abschnitt soll nicht den zukünftigen Überführungsprozess beschreiben, sondern einen Ausblick auf die weitere Vorgehensweise nach der Implementierung des Prototyps geben.

Dabei soll angesprochen werden, in welcher Reihenfolge die Komponenten überführt werden sollten und welche Herangehensweise sich für die schrittweise Überführung der graphischen Benutzeroberfläche anbietet.

7.1. Allgemeines Vorgehen

Nachdem die Kommunikation zwischen Leasman und LeasmanWorkbench realisiert ist und die erste Version des Prototyps erstellt wurde, kann darauf aufbauend die Überführung der einzelnen Komponenten der bisherigen graphischen Benutzeroberfläche beginnen.

Zu Beginn ist ein dynamisches Verhalten der Anwendung erforderlich. Dazu zählt unter anderem, dass sich alle Child-Windows innerhalb eines neuen Editors öffnen oder die Dialoge vor der LeasmanWorkbench angezeigt werden. Es muss beispielsweise auch realisiert sein, dass der Focus der Fenster beim Klick auf einen nicht aktiven Bereich, also Labels oder Hintergrund, eines internen Fensters, auf den äußeren Editor gelegt wird. Dies ermöglicht das Scrollen des Fensters und ähnliches. Solche Verhaltenseigenschaften, die bei einer eigenständigen Anwendung selbstverständlich sind, müssen bei der Implementierung des Prototyps oft langwierig von Hand erreicht werden. Dies ist notwendig, um der Anwendung das wirkliche Look and Feel einer eigenständigen Anwendung zu verschaffen. Der Anwender soll möglichst wenig in der Bedienung beeinträchtigt sein.

Ist dieses Verhalten der LeasmanWorkbench erreicht, kann mit der Überführung begonnen werden. Hierfür ist eine Analyse der unabhängigen Teile der Benutzeroberfläche erforderlich.

Da das Benutzermenü des Leasman bereits überführt ist, bietet es sich an, als nächstes die einzelnen Einstiege zu überführen und anschließend nacheinander jeden Unterpunkt, der sich in diesen Einstiegen befindet. Um sich der genauen Vorgehensweise zu nähern, sollten anfangs kleinere Einstiege im Vordergrund stehen.

Jeder Unterpunkt der Einstiege ist als kleinster überführbarer Teil anzusehen, der einzeln geplant werden kann. Dieser wird analysiert und alle zugehörigen Komponenten erfasst.

7.2. Migrationsreihenfolge

Da die neue Präsentationsschicht keine eigene Geschäftslogik enthalten soll, kann die Überführung einzelner Komponenten erst geschehen, wenn die Logik im BLS umgesetzt wurde.

Teile der Geschäftslogik sind bereits umgesetzt, wie die Vertragskomponente, die Partnerkomponente und die Kalkulationskomponente. Diese Teile könnten in der LeasmanWorkbench schon überführt werden.

Im Moment befindet sich der Leasman in der Version 6.02 und für September 2010 ist die Version 6.03 vorgesehen. Der Weitere Ausbau der Geschäftslogik im BLS ist bis zum Jahr 2014 bereits geplant. An diesem Konzept sollte sich die Überführung der Komponenten der LeasmanWorkbench orientieren.

Die Folgende Darstellung zeigt die zukünftig geplanten Umsetzungen der Geschäftslogik im BLS:

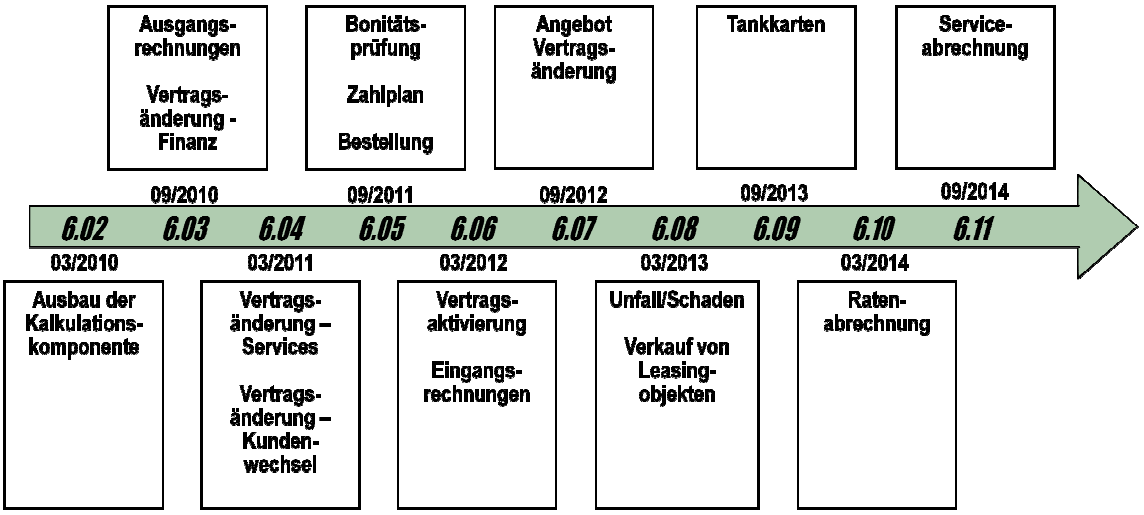


Abbildung 32: Zukünftige Umsetzungen der Geschäftslogik im BLS [Mehner08]

7.3. Ergonomie und Redesign der Benutzeroberfläche

Im Zuge der Überführung ist das Ziel, das Design der Benutzeroberfläche zu überarbeiten. Da der Leasman einen sehr langen Entwicklungs- und Wartungsprozess durchlaufen hat, ist das graphische Benutzerinterface an einigen Stellen nicht mehr „State of the Art“. Durch den Einsatz des RCP Frameworks wird dies erreicht.

Vor der Überführung sollte daher eine Analyse der bisherigen Benutzeroberfläche stehen, bei der die bestehenden Fenster auf ihr zeitgemäßes Design hin untersucht werden. Durch den Entwicklungsprozess wurden einige Fenster ständig erweitert, folglich führte das zu einer Überladung. Diese Fenster müssen durch ein Redesign überarbeitet werden.

Hierfür ist es notwendig die verschiedenen Fenstertypen zu ermittelt und mit modernen Designrichtlinien zu vergleichen. Danach ist eine Abbildung der graphischen Elemente der Fenster auf Komponenten von SWT und JFace möglich.

Für oft verwendete Fenster können Basisklassen entwickelt werden, von denen bei der Umsetzung in SWT abgeleitet wird. Diese legen das grobe Design und die Anordnung der wichtigsten Elemente eines Fensters von Vornherein fest. Dadurch wird ein Wiedererkennungswert erreicht.

Das Redesign der graphischen Elemente ist sehr wichtig, um von den modernen Richtlinien bei der Entwicklung von Benutzerinterfacen zu profitieren und die Stärken der neuen Benutzeroberfläche in möglichst hohem Maße auszunutzen.

7.4. Test der überführten Komponenten

Bei DELTA proveris wird für den Test der Benutzeroberfläche Rational Robot von IBM eingesetzt. Damit ist es möglich, Abläufe auf der graphischen Benutzeroberfläche aufzuzeichnen und anschließend automatisiert ablaufen zu lassen. Dies geschieht nicht auf Basis der Bildschirmkoordinaten, sondern auf Basis der Namen der Elemente wie Buttons und Labels. Der Ablauf dieser Tests wird in Scripts programmatisch festgehalten und kann bearbeitet werden.

Um die Tests weiterhin verwenden zu können, müssen sie für die Workbench angepasst werden, denn für Elemente, die angesprochen werden, sind Fenster anzugeben, in denen sich diese befinden. Die Fenster sind nun nicht mehr die des PowerBuilders, sondern die der Workbench.

Tests können auch verwendet werden, um die überführten Fenster des Leasman dahingehend zu testen, dass sie dieselbe Funktionalität aufweisen, wie die bisherigen PowerBuilder-Fenster. Dafür müssen neue Tests mit den überführten Fenstern geschrieben und die Eingabedaten der bestehenden Tests verwendet werden. Sind die Ausgabedaten der neuen Benutzeroberfläche identisch mit denen der bisherigen Fenster, kann davon ausgegangen werden, dass sich die neue Benutzeroberfläche gleich der bisherigen verhält. Diese Tests können dann immer wieder verwendet werden, bis die neue Benutzeroberfläche richtig funktioniert.

Später können die Tests auch eingesetzt werden um die Oberfläche bei Veränderungen an der dahinter liegenden Logik auf Funktionalität zu testen. [Robot00]

Kapitel 8
Schlussbetrachtung

Dieser Teil enthält eine abschließende Untersuchung der Diplomarbeit. Es wird ein Fazit erstellt, bei dem die Umsetzungen der Ziele dieser Diplomarbeit kritisch betrachtet werden. Abschließend erfolgt ein Ausblick auf die mögliche zukünftige Entwicklung der LeasmanWorkbench.

8.1. Fazit

Es wurde eine Analyse der bestehenden Technologie vorgenommen und ein Prototyp konzeptioniert, der alle Anforderungen für die geplante schrittweise Überführung erfüllt. Anschließend wurde das Konzept erfolgreich umgesetzt und die Problemstellung damit gelöst.

Es steht der DELTA proveris AG nun eine Möglichkeit zur Verfügung, die graphische Benutzeroberfläche des in Sybase PowerBuilder erstellten Leasman, in eine zukunftsorientierte Technologie zu überführen. Dabei ist ein schrittweises Vorgehen möglich, bei dem die bestehenden Komponenten der bisherigen Benutzeroberfläche vorerst weiter verwendet werden können, bis sie durch Neuimplementierungen abgelöst werden.

Erste Versuche ergaben, dass es ebenfalls möglich ist, nicht nur ganze zusammen gehörige Teile der bestehenden Benutzeroberfläche, die einer Aufrufkette angehören, zu ersetzen. Es ist ebenfalls möglich, einzelne Fenster innerhalb einer Aufrufkette zu überführen. Damit kann eine Aufrufkette auch von PowerBuilder-Fenstern über Java-Fenster zurück zu PowerBuilder und umgekehrt geschehen. Dies spricht für das erfolgreiche Konzept des Prototyps.

In Anbetracht des Inhaltes und der Umsetzung des Konzeptes in einen lauffähigen Prototyp, kann gesagt werden, dass die Ziele dieser Diplomarbeit erreicht wurden.

Es wurde bereits begonnen, den Prototyp weiter zu verbessern. Die meisten Einarbeitungen betreffen im Moment noch das Verhalten der graphischen Benutzeroberfläche als eigenständige Anwendung. Dadurch befindet sich die Leasman-Workbench schon in einem Entwicklungsstand, der sehr für das Konzept spricht.

Der tägliche Umgang mit der Anwendung zeigt keine gravierenden Stabilitätsprobleme. Besonders gut funktioniert das Konzept des Actionservers. Hier mussten bisher keine Änderungen vorgenommen werden.

Um die Arbeitsweise der LeasmanWorkbench zu demonstrieren, befindet sich ein Screencapture auf der beigefügten CD. Hier wird gezeigt, wie die Workbench mit den PowerBuilder-Fenstern zusammen arbeitet.

Diese Abbildung zeigt die LeasmanWorkbench in der Version 0.7:

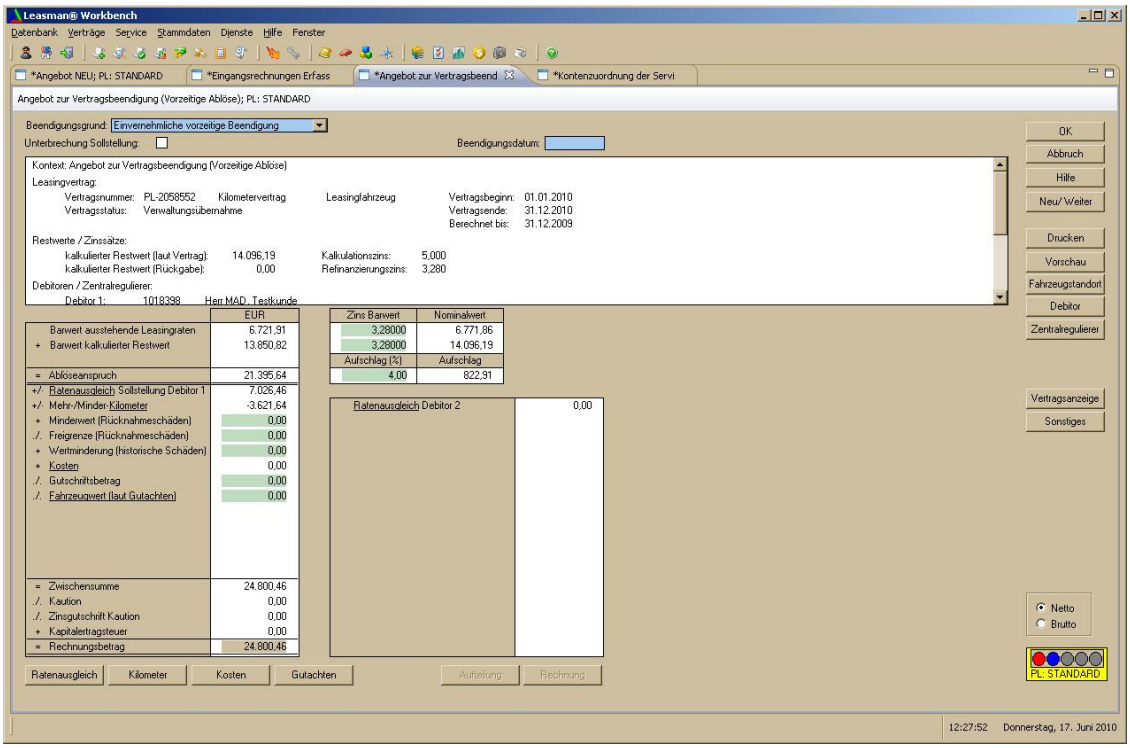


Abbildung 33: LeasmanWorkbench v0.7

Abbildung 33 zeigt die LeasmanWorkbench in ihrer aktuellen Ausbaustufe, der Version 0.7. Man kann das Hauptfenster des Leasman sehen, welches in SWT nachprogrammiert wurde. Darin befindet sich die Editor Area, in der vier Power-Builder-Fenster in Editoren geöffnet sind. Am unteren rechten Rand der Workbench befindet sich die Zeit und Datumsanzeige. Die Farbe des PowerBuilder-Fensters wurde auf die jeweilige Farbe der Window Benutzeroberfläche gestellt. Dadurch fällt es kaum auf, dass das innere Fenster nicht in SWT erstellt wurde.

8.2. Ausblick

Für die zukünftige Entwicklung der LeasmanWorkbench gibt es viele Möglichkeiten.

Durch das Pluginkonzept von Eclipse RCP ist es möglich, einzelne zusammengehörige Teile der Benutzeroberfläche jeweils als Plugin auszulagern. Dadurch könnten dem Kunden nur die Teile angeboten werden, welche zur Abbildung seiner Geschäftsprozesse notwendig sind. So kann der Kundenkreis von DELTA proveris auch auf kleinere Leasingunternehmen ausgeweitet werden.

Es wäre auch möglich die Rahmenanwendung der Workbench auszulagern und die LeasmanWorkbench als eigenes Plugin weiter zu entwickeln. Dadurch ergäbe sich die Möglichkeit, andere Anwendungen, die auch nicht Leasinggeschäft bezogen sein können, als Plugin in diese Rahmenanwendung zu integrieren. Damit stünde DELTA proveris eine Rahmenapplikation zur Verfügung, welche auch jede andere Art von Anwendung ermöglicht.

Diese Anwendungen können, falls dies notwendig ist, auch miteinander in einer Workbench untergebracht werden,.

Desweiteren könnten Lizenzen angeboten werden, die es den Kunden ermöglichen, ihre eigenen Plugins nach ihren Vorstellungen zu entwickeln.

Den Möglichkeiten, die sich durch den Einsatz der LeasmanWorkbench ergeben, sind kaum Grenzen gesetzt.

Anlagen

CD mit Anlagen

Internetquellen

Screencapture LeasmanWorkbench v0.7

Diplomarbeit als PDF

Literaturverzeichnis

[Hoffmann08] Hoffmann, Dirk W.: „Software-Qualität“, Springer-Verlag, Berlin Heidelberg 2005

[Kübeck09] Kübeck, Sebastian: „Software-Sanierung: Weiterentwicklung, Testen und Refactoring bestehender Software“, mitp/bhv, Sep. 2009

[Herczeg05] Herczeg, Michael: „Softwareergonomie“, 2. vollständig Überarbeitete Aufl., Wissenschaftsverlag GmbH, Oldenburg 2005

[Roock04] Roock, Stefan; Lippert, Martin: „Refactorings in großen Softwareprojekten“, 1. Auflage, dpunkt.verlag, Heidelberg 2004

[Wolff06] Wolff, Eberhard: „Spring, Framework für die Java-Entwicklung“ 1.Auflage, dpunkt.verlag, Heidelberg 2006

[Gunnerson00] Gunnerson, Eric: „C# - Die neue Sprache für Microsofts .NET Plattform“, Galileo Computing, September 2000

[Daum05] Daum, Berthold: „Rich-Client-Entwicklung mit Eclipse 3.1 - Anwendungen entwickeln mit der Rich Client Platform“, 1. Auflage, dpunkt.verlag, Heidelberg 2005

[Sippel08] Sippel, Heiko; Jestram, Michael; Bendisposto, Jens: „Eclipse Rich Client Platform – Entwicklung von erweiterbaren Anwendungen mit RCP“, entwickler.press, 2008

[Petri08] Petri, Jürgen: „NetBeans RCP – Das Entwicklerheft“ O’Reilly; Auflage 1, Januar 2008

[Wütherich08] Wütherich, Gerd; Hartmann, Nils; Kolb, Berndt; Lübken, Matthias: „Die OSGi Service Plattform – Eine Einführung mit Eclipse Equinox“ dpunkt.verlag, 1. Auflage, Heidelberg 2008

[Armstrong04] Armstrong, Bruce; Brown, Millard F.: „PowerBuilder 9 – Advanced Client/Server Development“, SAMS, 2004

[Java09.2007] Java Magazin: Ausgabe 09.2007

[Java02.2008] Java Magazin: Ausgabe 02.2008

[Java04.2010] Java Magazin: Ausgabe 04.2010

Quellenverzeichnis

[leasm] DELTA proveris AG; Leasman:
„Software für Leasing, Finanzierung, Fuhrparkmanagement“
http://www.depag.de/leasman.html
Letzter Zugriff: 23. Juni 2010

[ebc] ebcom AG, ebcom: „Der Lebenszyklus von IT-Anlagen“
http://www.ebcom.ch/default.aspx?navid=55
Letzter Zugriff: 23. Juni 2010

[usab] Handbuch Usability: „Definition Usability“
http://www.handbuch-usability.de/begriffsdefinition.html
Letzter Zugriff: 23. Juni 2010

[comx] Community MX - extending knowledge:
„Interview with a Usability Engineer”
http://www.communitymx.com/content/article.cfm?cid=4B44E
Letzter Zugriff: 23. Juni 2010

[Sre08] Enzyklopädie der Wirtschaftsinformatik Online – Lexikon:
Stefan Eicker: „Software Reengineering“
http://www.enzyklopädie-der-wirtschaftsinformatik.de/wi-enzyklopädie/lexikon/is-management/Integration-und-Migration-von-IT-Systemen/Software-Reengineering
Letzter Zugriff: 23. Juni 2010

[HK03] Harriet Kasper, Diplomarbeit:
„Redesign von Benutzeroberflächen durch Mittel der Navigation“
http://elib.uni-stuttgart.de/opus/volltexte/2003/1537/pdf/DIP-2081.pdf
Letzter Zugriff: 23. Juni 2010

[empr] empros GmbH - process & information management services:
„Code Smell“
http://www.empros.ch/vielfach/faustregeln/502149936a0f4bb0f/codesmell.php
Letzter Zugriff: 23. Juni 2010

[Prot] Kappel, Gerti; Nierstrasz:
“Prototyping in einer objektorientierten Entwicklungsumgebung”
http://scg.unibe.ch/archive/osg/Kapp89aPrototyping.pdf
Letzter Zugriff: 23. Juni 2010

[4GL] Naval Center for Cost Analysis:
“Fourth-Generation Languages Issue Paper”
http://www.ncca.navy.mil/services/software/4gl.pdf
Letzter Zugriff: 23. Juni 2010

[Swing] Hochschule Esslingen – University of Applied Sciences
“Swing”
http://www.it.fht-esslingen.de/~asbeck/show_room/MM-Swing-Project/
Letzter Zugriff: 23. Juni 2010

[OSGi09] OSGi Alliance
“OSGi Service Platform Core Specification – The OSGi Alliance”
Release 4, Version 4.2, Juni 2009
http://www.osgi.org/download/r4v42/r4.core.pdf
Letzter Zugriff: 23. Juni 2010

[Fitch08] Fitch, Will: “What is Web Service?”
April, 2008
http://www.willfitch.com/what-is-web-service.html
Letzter Zugriff: 23. Juni 2010

[Hib10] ITWissen – Das große Online-Lexikon für Informationstechnologie
"hibernate"
http://www.itwissen.info/definition/lexikon/Hibernate-hibernate.html
Letzter Zugriff: 23. Juni 2010

[Mehner08] Mehner, Andreas: "Leasman Release Preview 5.11"
DELTA proveris AG, 2008
Internes Dokument

[Robot00] Rational – the e-development company:
„Getting Started with Rational Robot", Version 2000.02.10
http://circe.univ-fcomte.fr/Docs/RationalRose/Rational%20Test%207/RobotGetStart.pdf
Letzter Zugriff: 23. Juni 2010

Selbständigkeitserklärung

Ich erkläre, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Mittweida, den 30. Juni 2010

_____

Thomas Richter