JOHANNES KEPLER
UNIVERSITÄT LINZ | JKU

TNF

Technisch-Naturwissenschaftliche
Fakultät

Beispielprogramm Kundenbeziehungsmanagement „Plux-CRM“ für die Plugin-Plattform Plux.NET

MASTERARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Masterstudium

SOFTWARE ENGINEERING

Eingereicht von:
Sabine Weiss Bakk. techn.

Angefertigt am:
Institut für Systemsoftware

Beurteilung:
o. Univ.-Prof. Dr. Dr. h.c. Hanspeter Mössenböck

Mitwirkung:
Mag. Dr. Reinhard Wolfinger

Linz, Mai 2010

Kurzfassung

In den letzten Jahren haben Programme mit einer komponentenbasierten Architektur in der Software-Entwicklung immer mehr an Bedeutung gewonnen. Komponentenbasierte Programme, auch plugin-basierte Programme genannt, besitzen die Eigenschaft, dass sie auf jeden Benutzer individuell zugeschnitten werden können. Bei komponentenbasierten Programmen werden von der Laufzeitumgebung nur jene Komponenten geladen, die tatsächlich benötigt werden. Dies wirkt sich positiv auf das Laufzeitverhalten der Software aus und bringt einen erheblichen Vorteil gegenüber monolithischen Programmen hervor, bei denen zum Startzeitpunkt der Software bereits alle vorhandenen Komponenten geladen werden.

Plux-CRM wurde im Rahmen dieser Diplomarbeit entwickelt und ist eine Software, die eine komponentenbasierte Architektur enthält. Sie basiert auf dem Plugin-Framework Plux.NET und wurde mit dessen Komponenten implementiert. Plux-CRM zeigt, wie eine Software mit der Plux.NET-Architektur zu programmieren ist. Der klar strukturierte Aufbau der einzelnen Komponenten mit deren definierten Schnittstellen ist in Plux-CRM von großer Bedeutung. Die Benutzeroberfläche von Plux-CRM besteht aus einer Anordnung von einzelnen Komponenten, die sich zur Laufzeit der Software verändern lassen. Der Benutzer kann selbst bestimmen, wie er seine Benutzeroberfläche konfiguriert. Speziell entwickelte Benutzersteuerelemente zeigen, wie die komponentenbasierte Plux.NET-Architektur auf Interaktionen des Benutzers reagiert.

Die Aufgabe von Plux-CRM ist neben der Einführung in die Entwicklung einer Plux.NET-Anwendung, anhand eines Kundenbeziehungsmanagementsystems das Plux.NET-Framework zu bewerten. Dabei wird speziell auf die Eigenschaften der komponentenbasierten Architektur eingegangen.

Abstract

In the last few years the acceptation of programs with a component-based architecture has increased in the software-engineering. The characteristic of a component-based program (also called plugin-based program) is, that it can be adapted individually to the requirements of every single user. The runtime-environment of such a program loads only those components, it really needs at a special time. This has positive effects to the runtime-behaviour of the software and affords the advantage over monolithic programs, whose components are loaded at starting time.

Plux-CRM was implemented for this master thesis and is a software, which contains a component-based architecture. It is based on the plugin-framework Plux.NET and was implemented with those special components. Plux-CRM shows the way to implement an application based on the architecture of Plux.NET. In Plux-CRM the structured composition of components and the well-defined interfaces are very important. The graphical user interface contains several components, which can be configured at runtime. Each user can decide, which configuration of his graphical user interface he wants. Special user controls show, how the architecture of Plux.NET reacts on the interactions with a user.

The major task of Plux-CRM is to evaluate the plugin-framework Plux.NET by means of a customer relationship management (CRM). For that matter the focus of the evaluation lies on the characteristics of a component-based architecture.

Inhaltsverzeichnis

1 Einleitung

Erweiterbare Architekturen gewinnen in der Softwareentwicklung an Bedeutung. Ein Ansatz für erweiterbare Architekturen sind *Plugin*-Komponenten. Dabei teilt man ein Programm in einen schlanken Kern und in Plugin-Komponenten, die in den Kern eingesteckt werden, ohne den Quellcode neu übersetzen zu müssen. Die Plugin-Architektur in der Softwareentwicklung nimmt einen immer höheren Stellenwert ein, weil maßgeschneiderte Software immer wichtiger wird. Aus diesem Grund entwickelte das „Christian Doppler Labor für Automated Software Engineering“ an der Johannes Kepler Universität Linz in enger Zusammenarbeit mit der Firma BMD Systemhaus GmbH eine Software, die zur Laufzeit einzelne Komponenten zu einem Programm zusammenbaut [1]. Ziel dieser Entwicklung war eine Plattform für flexible Programme. Unter flexibler Software verstehen wir in dieser Arbeit das Anpassen und Erweitern eines Programms, ohne dass das Programm neu übersetzt werden muss. Weiters soll ein derartiges Programm an den Benutzer angepasst werden können. Für diesen Vorgang soll keine neue Programmierung der Software nötig sein.

Das *Plux.NET-Komponentenmodell* erlaubt dynamische Komposition. Dabei lädt ein Programm erst dann seine Komponenten, wenn diese benötigt werden. Dadurch wird gewährleistet, dass ein Programm so klein und einfach wie möglich bleibt.

Das Plux.NET-Framework stand lange Zeit in der Entwicklungsphase. Berichte und Evaluierungen über die Verwendung des Plux.NET-Frameworks waren zu Beginn dieser Diplomarbeit großteils ausständig. Deshalb entstand die Idee, Plux.NET anhand eines Fallbeispiels zu evaluieren. Die Vor- und Nachteile einer flexiblen Architektur und somit auch von Plux.NET sollen dabei genau aufgezeichnet werden.

Diese Arbeit stellt das Anwendungsprogramm *Plux-CRM* vor. Es basiert auf den Plux.NET-Komponenten und zeigt, wie Programme mit flexibler Architektur zu entwerfen und implementieren sind. Dabei ist eine klare Strukturierung der einzelnen Komponenten und eine gute Lesbarkeit des Quellcodes erforderlich, da der Quellcode von Plux-CRM anderen Entwicklern als Vorlage dienen soll. Beim Entwurf und bei der Implementierung von Plux-CRM ist auf die Artefakte einer flexiblen Architektur einzugehen. Dazu zählt die Einteilung in spezielle Komponenten und Plugins.

1.1 Aufgabenstellung

Die im Rahmen dieser Diplomarbeit zu implementierende Software *Plux-CRM* soll ausschließlich der Evaluierung des Plux.NET-Frameworks dienen. Anhand des Fallbeispiels Kundenbeziehungsmanagement sind sowohl die Vorzüge, als auch die Nachteile einer flexiblen Architektur aufzuzeigen. Anzunehmen ist, dass zu den Vorzügen einer derartigen Software, wie Plux-CRM, die Flexibilität zählt. Zu den Nachteilen wird die Komplexität der flexiblen Architektur zählen. Diese Annahmen sind im Rahmen dieser Diplomarbeit zu überprüfen.

Das Ergebnis dieser Diplomarbeit ist neben dem Aufführen dieser Vor- und Nachteile, die Architektur und die Implementierung der Software basierend auf dem Plux.NET-Framework.

Der Entwurf und die Implementierung von Plux-CRM soll zeigen:

- wie man Programme in Plugins (Komponenten) zerlegt.
- wie man Plugins zusammen mit der Laufzeitumgebung von Plux.NET zu einem Programm verbindet.
- wie man ein Programm konfiguriert, um Plugins zur Laufzeit in das Programm einzubinden oder wieder zu entfernen.
- wie man mit den Steuerelementen der Plux.NET-Bibliothek Benutzeroberflächen baut, die sich dynamisch an geänderte Plugin-Konfigurationen anpassen.

Zusätzlich ist bei der Implementierung von Plux-CRM folgendes zu beachten: Aufgrund der individuellen Anpassungsfähigkeit verwendet jeder Benutzer nur einen kleinen Teil der Funktionen von Plux-CRM. Auf diese Weise ist auf umfangreiche Menüleisten mit zahlreichen Funktionen zu verzichten. Die Auswahl der vorhandenen Funktionen ist auf minimale Steuerelemente zu beschränken. Diese Steuerelemente sollen sich dynamisch an die Konfiguration des Benutzers anpassen. Die Plux.NET-Bibliothek enthält Steuerelemente, die bei Bedarf in Plux-CRM zu übernehmen sind. Werden neue Steuerelemente benötigt, sind diese auf gleicher Weise zu implementieren. Kommen diese Steuerelemente an mehreren Stellen vor, sind sie in das Plux.NET-Framework zu übernehmen.

Bei der Datenhaltung von Plux-CRM ist auf SQL-Datenbanken und Bibliotheken von Drittanbietern zu verzichten. Plux-CRM soll keine externen Bibliotheken oder Programme voraussetzen, weil es eine Demo-Anwendung für Plux.NET sein soll, die mit geringem Aufwand zu installieren ist. Nur Plux.NET soll für das Ausführen von Plux-CRM nötig sein. Die Abhängigkeit zu einem Datenbanksystem soll vermieden werden. Stattdessen sollen die Daten zur Laufzeit im Hauptspeicher liegen und bei Programmstart aus einer Datei geladen bzw. beim Programmende in eine Datei gespeichert werden.

Plux-CRM soll außerdem als Referenz zum Erlernen der Plux.NET-Programmierung dienen und erfordert somit eine klare Strukturierung durch eine Schichtenarchitektur und eine einfache Benutzerschnittstelle.

1.2 Funktionsumfang

Plux-CRM ist ein Programm zum Verwalten von Kundenbeziehungen (Custom Relationship Management). Damit das Programm einfach bleibt, soll sich Plux-CRM auf folgende Funktionen beschränken:

- Verwalten von Kunden: Zu Kunden zählen sowohl Einzelpersonen als auch Firmen. Dazu sind die entsprechenden Kontaktdaten, zum Beispiel Anschrift, Telefonnummer und Email zu speichern.
- Verwalten von Geschäften: Benutzer schließen mit Kunden Geschäfte ab. Ein Geschäft definiert, welches Projekt in welcher Zeit zu einen bestimmten Preis fertig zu stellen ist. Geschäfte sind entweder noch ausständig oder gehen zum Vorteil des Benutzers oder des Kunden aus.
- Verwalten von Projekten: Projekte werden mit Kunden abgewickelt. Eine Übersicht fasst beteiligte Personen, Dokumente und wichtige Notizen eines Projektes zusammen. Projekte werden ausgewählten Benutzern zugeteilt.
- Verwalten von Aufgaben: Mit Aufgaben plant ein Benutzer Tätigkeiten für sich selber oder delegiert diese an andere Benutzer. Aufgaben werden bestimmten Kategorien, wie Besprechungen oder Telefonaten zugeteilt. Fälligkeitstermine und Benachrichtigungen erinnern den Benutzer an diese Aufgaben.
- Verwalten von Notizen und Kommentaren: Notizen und Kommentare sind zusätzliche Informationen, die ein Benutzer über Kunden, Geschäfte und Projekte verfasst. Ein Journal zeigt die zuletzt hinzugefügten Notizen und Kommentare.
- Verwalten von Stichworten: Benutzer können Kunden, Projekte, Aufgaben und Geschäfte mit Stichworten versehen. Diese helfen beim Zuordnen zu Themenbereichen.
- Verwalten von Benutzern: Mehrere Mitarbeiter einer Firma befassen sich gleichzeitig mit Plux-CRM. Somit verfügt jeder Mitarbeiter über ein eigenes Benutzerkonto von Plux-CRM. Benutzerspezifische Rechte und Einstellungen bestimmen, welche Funktionen ein Benutzer verwenden darf.

- Verwalten chronologischer Aktivitäten: Das *Schwarze Brett* zeigt die wichtigsten Aktivitäten der letzten Tage. Dabei wird zwischen den zuletzt getätigten Aktivitäten eines einzelnen und aller Benutzer unterschieden. Zu den Aktivitäten zählen das Hinzufügen und Ändern von Geschäften, Notizen und das Erledigen von Aufgaben.

1.3 Kapitelübersicht

Kapitel 2 stellt das Plugin-Framework Plux.NET vor. Es zeigt die Elemente des Plux.NET-Kompositionsmodells und die Mechanismen für das dynamische Zusammenstecken.

Kapitel 3 beschreibt die Herausforderungen und Lösungsideen beim Entwurf von Plux-CRM. Das in dieser Software verwendete Datenmodell und die Architektur der Benutzeroberfläche finden in diesem Kapitel Platz. Eine ausführliche Beschreibung aller Elemente der Benutzeroberfläche und deren Abhängigkeiten ist dabei ein wichtiger Bestandteil.

Kapitel 4 nimmt Bezug auf zwei wichtige Benutzersteuerelemente von Plux-CRM. Anhand ausgewählter Beispiele wird ihre Funktionalität detailliert aufgeführt.

Kapitel 5 behandelt die speziellen Funktionalitäten von Plux-CRM. Zu diesen Funktionalitäten zählen zum einen die individuelle Konfiguration der Applikation, zum anderen die Verwendung zusätzlicher Erweiterungen. Anhand ausgewählter Szenarios werden diese Funktionalitäten genau beschrieben.

Kapitel 6 befasst sich mit der Bewertung von Plux-CRM. Der Kern liegt dabei auf den Vor- und Nachteilen, die bei der Verwendung des Plux.NET-Frameworks zum Vorschein kommen.

In Kapitel 7 beleuchten Statistiken mit technischen Daten von Plux-CRM, aus wievielen Komponenten die Software zusammen gesetzt ist. Ausgewählte Szenarien zeigen die unterschiedliche Anzahl an verwendeten Komponenten zu verschiedenen Zeitpunkten.

Kapitel 8 enthält eine abschließende Zusammenfassung über die Entwicklung von Plux-CRM und die Bewertung des Plux.NET-Frameworks.

2 Das Plugin-Framework Plux.NET

Plux.NET ist ein Plugin-Framework für die .NET-Plattform. Das Framework bildet eine Applikation mit einer flexiblen Architektur. Dazu besitzt es einen dünnen Kern und eine Sammlung von Software-Komponenten. Um in der Applikation derartige Software-Komponenten hinzuzufügen, werden speziell dafür vorgesehene Steckplätze mit diesen Software-Komponenten befüllt. Äquivalent dazu sind beim Entfernen der Software-Komponenten aus den Steckplätzen diese in der Applikation nicht mehr vorhanden. Das Hinzufügen und Entfernen der Software-Komponenten erfolgt entweder zum Startzeitpunkt oder wird zur Laufzeit vorgenommen. Abbildung 2.1 zeigt das Grundprinzip des dünnen Kerns mit den erweiterten Software-Komponenten und deren Steckplätzen.

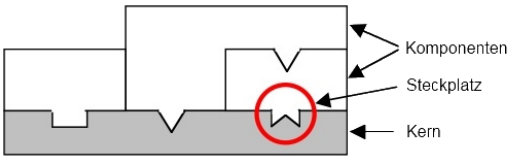

Abbildung 2.1: Grundprinzip der flexiblen Architektur von Plux.NET [2]

2.1 Plux.NET-Komponentenmodell

Das Plugin-Framework Plux.NET enthält ein Komponentenmodell (*CM - Composition model*), das es ermöglicht, einzelne Software-Komponenten von plugin-basierten Systemen während der Laufzeit in die Anwendung zu übernehmen. Eine Software-Komponente kann implementiert werden, ohne Änderungen am Komponentenmodell oder an anderen Software-Komponenten vorzunehmen. Ein Komponentenmodell ist ein System, das die Kompositionen von Software-Komponenten verwaltet. Das Komponentenmodell hat zwei Aufgaben. Zum einen legt es den Aufbau jeder einzelnen Software-Komponente fest. Zum anderen regelt es die Verbindungen zwischen den einzelnen Software-Komponenten.

Das Plux.NET-Komponentenmodell weist drei besondere Merkmale auf. Als erstes stellt es einen Kompositionsdienst zur Verfügung, der aktiv die Komposition der einzelnen Plugin-Komponenten steuert. Das zweite Merkmal bezieht sich auf die Plugin-Komponenten, die

die Aufgabe eines *Hosts* übernehmen. Ein Host stellt einen Kern dar, der um entsprechende Plugin-Komponenten erweitert werden kann. Jede Host-Komponente verwaltet seine Komponenten, um die sie erweitert wird. Das Kompositionsmodell speichert die Beziehungen zwischen Host- und Contributor-Komponenten. Als drittes Merkmal ist anzuführen, dass jede Plugin-Komponente Teil eines nachrichten-basierten Systems ist. Jede Plugin-Komponente reagiert entsprechend auf die Benachrichtungen, die der Kompositionsdienst auslöst (*dynamische Komposition*). Diese drei Merkmale ermöglichen es, dass eine Anwendung zur Laufzeit erweitert werden kann, ohne diese neu zu starten. Der Kompositionsdienst fügt zum Startzeitpunkt alle Plugin-Komponenten zu einer Applikation zusammen und ändert diese zur Laufzeit.

2.1.1 Zusammenhang von Steckplätzen und Steckern

Eine Komposition definiert, wie unterschiedliche Komponenten miteinander verbunden werden, um eine spezielle Funktion einer Software zu übernehmen. Das Verhalten dieser Funktion ergibt sich aus dem Zusammensetzen der einzelnen Komponenten. Mehrere kleinere Komponenten können zu einer größeren Komponente auf einer höheren Abstraktionsebene miteinander verbunden werden.

Das Plux.NET-Komponentenmodell fügt mehrere Komponenten zu einer einzigen Anwendung zusammen. Eine Komponente wird in Plux.NET als *Extension* bezeichnet. Sie kann sowohl andere Komponenten verwenden, als auch eigene Funktionen anderen Komponenten zur Verfügung stellen.

Bei den Extensions sind zwei Unterscheidungen zu beachten. Einerseits gibt es in Plux.NET sogenannte *Host-Extensions*. Host-Extensions sind Komponenten, die einen Steckplatz enthalten, um Contributor-Extensions zu verwenden. Andererseits verwendet Plux.NET den Begriff *Contributor-Extension*. Contributor-Extensions sind Komponenten, die einen Stecker enthalten, der der Steckplatz-Spezifikation einer Host-Komponente entspricht. Sie können wiederum ihre eigenen Steckplätze für andere zur Verfügung stellen und gleichzeitig als Host-Extension dienen. Somit ist es möglich, in einer plugin-basierten Applikation eine Hierarchie von Komponenten aufzubauen. Abbildung 2.2 zeigt, wie das Zusammenspiel einer Host-Extension mit seinem Steckplatz und einer Contributor-Extension mit seinem Stecker funktioniert.

Eine Komponente kann sowohl über Steckplätze als auch über Stecker verfügen. Ein Steckplatz wird in Plux.NET als *Slot* bezeichnet. Er definiert, wie die Funktionalität einer Komponente anhand anderer Komponenten erweitert wird. Die Bezeichnung *Plug* steht in Plux.NET für einen Stecker. Dieser legt die entgegengesetzte Richtung eines Steckplatzes fest. Ein Stecker definiert wie eine Komponente als Erweiterung einer anderen

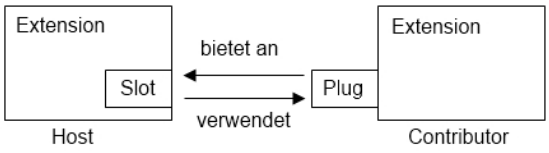

Abbildung 2.2: Zusammenspiel zwischen Host- und Contributor-Extension [3]

Komponente dient. Im Detail bestimmt ein Steckplatz die Art der Informationen, die eine Komponente erwartet, und ein Stecker bietet diese Informationen an. Die Spezifikationen eines Steckplatzes und eines Steckers müssen übereinstimmen, um deren Funktionalität zu gewährleisten. Eine derartige Spezifikation wird über eine sogenannte *Slot-Definition* geregelt. Sie wird als Schnittstelle (*Interface*) dargestellt. Eine Schnittstelle kann über zusätzliche Parameter verfügen, die einen Steckplatz genauer definieren. Schnittstellen sind immer in Host-Komponenten enthalten, hingegen kommen die Implementierungen der Schnittstellen nur in Contributor-Komponenten vor (siehe Abbildung 2.3).

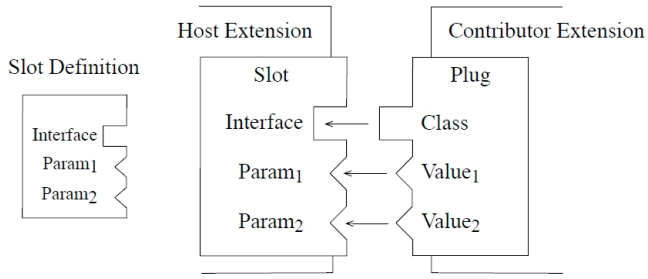

Abbildung 2.3: Zusammenspiel zwischen Steckplatz (Slot) und Stecker (Plug) [3]

In Plux.NET werden Komponenten und deren Steckplatz-Definitionen getrennt behandelt. Dabei unterscheidet man zwischen *Plugins* und *Contracts*. In einem Plugin sind mehrere Komponenten zusammengefasst, die ein Feature abdecken. Beim Hinzufügen eines Plugins in eine Anwendung werden alle im Plugin enthaltene Komponenten mitübernommen. Plugins sind unabhängig voneinander und können einzeln behandelt werden. Ein Contract enthält alle benötigten Steckplatz-Definitionen zusammen mit ihren Schnittstellen. Er ist unabhängig von Plugins, wird aber von jedem einzelnen Plugin benötigt, da er die Basis aller Plugins bildet. Komponenten, die Stecker enthalten, müssen die Steckplatz-Definitionen eines Contracts kennen. Auch Host-Komponenten benötigen diese Steckplatz-Definitionen, wenn sie Contributor-Komponenten benutzen. In diesem Fall verwenden die Host-Komponenten die entsprechenden Schnittstellen aus den Steckplatz-Definitionen. Plux.NET verwendet *Class Library Assemblies* (*DLL*) als Plugins. Eine Class Library Assembly ist in .NET die kleinste Einheit, um Komponenten zu laden. Ein Plux.NET-Plugin enthält mehrere Komponenten. Plux.NET kann ein ganzes Plugin, eine ganze Komponente oder nur einen Stecker einer Komponente integrieren.

2.1.2 Meta-Informationen

Das Plux.NET-Komponentenmodell enthält *Type Meta Elements* und *Instance Meta Elements*. Diese stellen Meta-Informationen dar, die sowohl die einzelnen Komponenten als auch ihre Verbindungen definieren. Die statischen Informationen der einzelnen Komponenten mit ihren Steckplätzen und Steckern regelt das Plux.NET über die *Meta Elements*. Diese Typen finden in allen Abstraktionsebenen Verwendung. So enthält ein Plux.NET-Contract alle Steckplatz-Definitionen (*SlotDefinition*), die wiederum sämtliche Definitionen der einzelnen Parameter enthalten (*ParamDefinition*). Ein Plux.NET-Plugin besitzt alle Typen von Komponenten (*ExtensionType*). Diese verfügen über alle Steckplatz- (*SlotType*) und Stecker-Typen (*PlugType*). Jeder Stecker-Typ enthält Parameter. Abbildung 2.4 zeigt die verschiedenen Meta-Informationen des Plux.NET-Komponentenmodells.

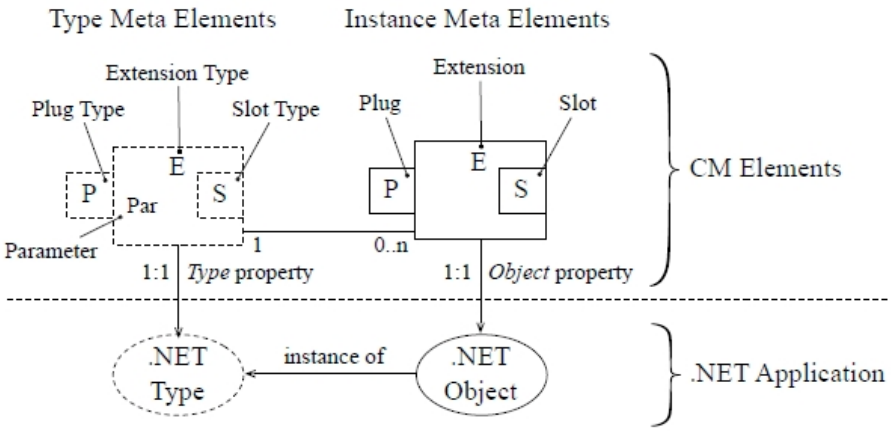

Abbildung 2.4: Zusammenhang von Meta-Informationen und .NET-Elementen [3]

Zur Laufzeit entspricht jede Komponente (*Extension*) einem .NET-Objekt. Im Plux.NET-Komponentenmodell werden diese Objekte als *Instance Meta Elements* bezeichnet. Der Typ einer Komponente (*ExtensionType*) entspricht einem .NET-Typ, von dem das .NET-Objekt erzeugt wurde. Jede Software-Komponente kann über ihren Namen oder über eine eindeutige Nummer identifiziert werden. Gibt es nur ein Objekt eines Komponententyps, so reicht die Identifikation über den Namen. Existieren mehrere Objekte dieses Komponententyps, ist der Name nicht mehr eindeutig und die Unterscheidung erfolgt über die Identifikationsnummer. Die Namen der Steckplatz-Definitionen einer Komponente müssen in der gesamten Plux.NET-Applikation eindeutig sein.

2.1.3 Beziehungen zwischen Komponenten

Wird eine neue Contributor-Komponente im Plux.NET-Komponentenmodell erkannt, muss sie einige Schritte durchlaufen bis sie in einer Applikation verwendet werden kann.

Zuerst erkennt der *Discovery*-Dienst eine Komponente. Der *Analyzer*-Dienst liest die Metadaten dieser Komponente und der *Discovery*-Dienst fügt diese Metadaten in den *TypeStore* ein. Der Kompositionsdienst erkennt die Änderung im *TypeStore* und registriert die Komponente (*registered*). Somit hat der Host die Contributor-Komponente als solche erkannt. Der Host erkennt sowohl alle Meta-Informationen als auch alle Steckplätze der Komponente und verknüpft diese mit den zugehörigen Stecker-Typen. Über einen *Qualification*-Mechanismus wird überprüft, ob der Stecker die Anforderungen einer Steckplatz-Definition erfüllt. Sind die Steckplatz-Definitionen nicht korrekt, beispielsweise gibt es mehrere Steckplätze mit demselben Namen, so wird der neue Typ ignoriert. Ansonsten wird der neue Typ dem Komponentenmodell hinzugefügt.

Bis zu diesem Zeitpunkt wurde noch kein Objekt der neuen Komponente erzeugt. Bevor eine Contributor-Komponente eingesteckt (*plugged*) wird, wird zuerst ein Metaobjekt für den Stecker erstellt und mit dem Metaobjekt des Steckplatzes verbunden. Gleichzeitig wird das mit dem Metaobjekt assoziierte .NET-Objekt erzeugt. Nach diesem Vorgang kann die Host-Komponente die neue Contributor-Komponente verwenden. In den meisten Fällen folgt nach der Registrierung einer Komponente das Einstecken in einen passenden Steckplatz. Dieser Vorgang kann bei Bedarf auch verzögert werden, sodass das Einstecken zu einem späteren Zeitpunkt erfolgt.

Je nach Definition können in einen Host-Steckplatz eine (*Single Cardinality*) oder mehrere Contributor-Stecker (*Multiple Cardinality*) eingesteckt werden. Wenn mehrere Contributor-Komponenten eingesteckt sind, wählt der Host eine dieser Komponenten als selektiert aus (*selected*) und setzt den Fokus auf diese selektierte Komponente (*Single Selection*). Es ist möglich, dass das Komponentenmodell mehrere Komponenten gleichzeitig selektiert (*Multi Selection*).

2.1.4 Kern des Komponentenmodells

Damit das Plux.NET-Komponentenmodell alle Komponenten zu einer Plux.NET-Applikation zusammenführt, besitzt es einen dünnen Kern, der um diese Komponenten erweitert wird. Dieser Kern liegt im Komponentenmodell in Form der *Core*-Komponente vor und ist eindeutig zu identifizieren. Da die Core-Komponente die Wurzel aller Komponenten ist, wird sie beim Starten einer Applikation in keinen Steckplatz eingesteckt, sondern automatisch gestartet. Sie besitzt somit keinen Stecker.

Die Core-Komponente verfügt über die zwei Steckplätze *Discovery* und *Startup*. Diese werden zu Beginn einer Applikation automatisch geöffnet. Abbildung 2.5 zeigt den Aufbau der Core-Komponente.

Der Steckplatz Discovery integriert *Discoverer*-Komponenten. Ein Discoverer erkennt Änderungen im Komponenten-*Repository*. Mit Hilfe von *Analyzer*-Komponenten liest ein

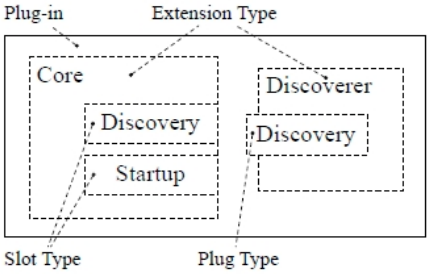

Abbildung 2.5: Aufbau der Core-Komponente [3]

Discoverer Metadaten aus Plugins und Contracts. Ein Discoverer ist nicht direkt im Komponentenmodell integriert, wird aber vom Komponentenmodell benötigt. Er wird zu Beginn einer Applikation automatisch erzeugt und in den passenden Steckplatz des Kerns eingesteckt. Der Kompositionsdienst lädt dabei zur Laufzeit alle Komponenten und integriert diese in die passenden Steckplätze. Er verwendet dazu einen Mechanismus, der zuerst in die Breite und anschließend in die Tiefe geht. Als erstes erkennt ein Discoverer jene Komponenten, die in den Steckplatz *Startup* des Kerns einzufügen sind. Dazu erzeugt er ein entsprechendes Objekt des ersten Komponententyps und integriert es in den *Startup*-Steckplatz. Bevor die nächste Komponente dem *Startup*-Steckplatz hinzugefügt wird, werden alle Contributor-Komponenten der ersten Komponente nacheinander abgearbeitet. Der Discoverer geht folgendermaßen vor: Der erste Schritt, bei dem eine Komponente eingesteckt wird (*Plug*), erfolgte bereits. Anschließend werden alle Steckplätze dieser Komponente geöffnet (*OpenSlots*). Danach folgt die Registrierung aller Contributor-Komponenten (*RegisterAll*). Dabei wird jede einzelne Contributor-Komponente registriert (*Register*) und in den Host-Steckplatz integriert (*Plug*). Somit beginnt der Vorgang wieder von vorne (siehe Abbildung 2.6).

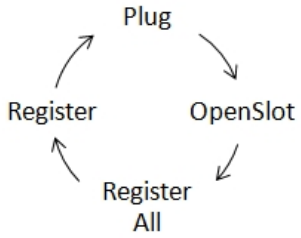

Abbildung 2.6: Aufruf des Lademechanismus

Die Abarbeitung der einzelnen Contributor-Komponenten eines Steckplatzes erfolgt nach dem *FIFO*-Prinzip (*First-In, First-Out*). Diejenigen Komponenten, die als erstes erkannt werden, werden auch als erstes dem Steckplatz hinzugefügt.

2.2 Plux.NET-Attribute

Mit dem Plux.NET-Komponentenmodell kann man beispielsweise *Rich Client Applications* implementieren. Bei der Implementierung dieser Applikationen sind einige Richtlinien einzuhalten. Zu diesen Richtlinien zählen das korrekte Verwenden spezieller Attribute. Attribute kennzeichnen zum Beispiel eine Klasse als Komponente (Extension) oder spezifizieren über Steckplatz-Definitionen welche Steckplätze und Stecker eine Komponente hat.

Im folgenden Abschnitt wird auf die Verwendung der Attribute von Steckplatz-Definitionen und Komponenten genauer eingegangen. Mit einem Beispiel von der Plux.NET Homepage [2] wird aufgezeigt, wie die Attribute der Steckplatz-Definitionen und Komponenten zu verwenden sind. Das Beispiel zeigt einen Steckplatz für Menüeinträge.

2.2.1 Attribute einer Steckplatz-Definition

Eine Steckplatz-Definition (*Slot-Definition*) legt die Schnittstelle eines Steckplatzes fest. Die Komponenten, die in einen Steckplatz integriert werden, müssen den Kriterien der Schnittstelle eines Steckplatzes entsprechen und gewisse Informationen zur Verfügung stellen. Im Detail implementiert eine Steckplatz-Definition eine Schnittstelle (*Interface*) mit einem Namen und einer Liste von Parametern als .NET-Attribute. Schnittstellen erkennt man in .NET daran, dass der Name mit dem Präfix „I“ beginnt.

Das Attribut `SlotDefinition` spezifiziert den Namen, über den die Stecker einer Komponente diesen Steckplatz ansprechen. Dieser Name muss sowohl eindeutig als auch immer vorhanden sein, um den Steckplatz korrekt zu verwenden. Die Parameter-Liste eines Steckplatzes ist optional. Jeder Parameter wird über das Attribut `Param` definiert und legt den Namen, den Datentyp und einen optionalen Standardwert fest.

Abbildung 2.7 zeigt die Definition für den Steckplatz `MenuItem`. Dieser Steckplatz stellt Platz für Erweiterungen einer Menübar in einer Applikation zur Verfügung. Als Parameter sind ein Text (*Text*) und ein Symbol (*Icon*) für einen Menüeintrag spezifiziert.

2.2.2 Attribute einer Komponente

In Plux.NET entspricht eine Komponente (Extension) einer Klasse. Eine Komponente muss die Kriterien der Schnittstelle eines Steckplatzes erfüllen. Zusätzlich enthält sie den Namen des Steckplatzes und eine vorgegebene Parameter-Liste.

Damit eine Klasse als Komponente erkannt wird, muss sie das Attribut `Extension` zusammen mit einem Namen besitzen. Über das Attribut `Plug` kombiniert mit dem Namen des entsprechenden Steckplatzes ist spezifiziert, dass diese Contributor-Komponente

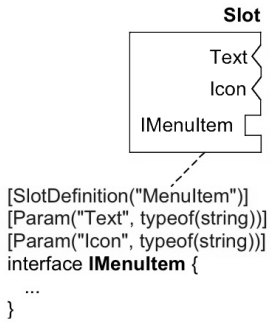

Abbildung 2.7: Beispiel der Steckplatz-Definition MenuItem [2]

im angeführten Steckplatz steckt. Es besteht die Möglichkeit, dass eine Contributor-Komponente zur Laufzeit mehrere Stecker besitzt. Jeder Stecker ist dabei über ein eigenes `Plug`-Attribut zu definieren. Übernimmt eine Komponente die Aufgaben eines Hosts, so definiert sie einen Steckplatz über das Attribut `Slot` gemeinsam mit dem Namen. Jeder Parameter, für den gewisse Informationen von einer Komponente zur Verfügung zu stellen sind, wird über das Attribut `ParamValue` zusammen mit dem Namen des Parameters und des eingesetzten Wertes festgelegt. Genauere Definitionen der Attribute einer Komponente sind [3] zu entnehmen.

Anhand der Definition des Steckplatzes `MenutItem` in Abbildung 2.8 muss ein Eintrag in der Menübar einen Text und ein Symbol enthalten. Die Contributor-Komponente `PrintItem` stellt den Befehl zum Drucken eines Dokumentes dar. Anhand der Parameterwerte „Print“ und des entsprechenden Symbols „Print.gif“ wird die Contributor-Komponente `PrintItem` in die Menübar der Benutzschnittstelle integriert.

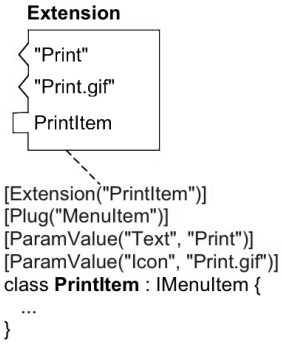

Abbildung 2.8: Beispiel der Komponente PrintItem [2]

2.3 Hot Plugging

Hot Plugging wird von einigen Plugin-Plattformen, wie Plux.NET, unterstützt. Es ist dafür verantwortlich, dass eine Applikation sowohl zum Startzeitpunkt, als auch während der Laufzeit jederzeit um Komponenten erweitert werden kann. Die Applikation muss dabei nicht neu gestartet werden.

In Plux.NET wird Hot Plugging durch *Cerberus* unterstützt. Cerberus ist eine Discoverer-Komponente, die erkennt, wenn aus einem Verzeichnis im Dateisystem Dateien in Form von Bibliotheken (*.dll*-Dateien) hinzugefügt oder entfernt werden. In Plux.NET wird dazu ein bestimmtes Verzeichnis im Dateisystem ausgewählt, das als Kommandozeilenparameter an Plux.NET übergeben wird. Erkennt Cerberus ein neues Plugin in diesem Verzeichnis, liest er die Metadaten und gibt diese an den *TypeStore* weiter [4]. Beim Start einer Applikation verwendet Plux.NET den eingebauten *Bootstrap*-Discoverer [4]. Cerberus wird verwendet, wenn man zur Laufzeit Änderungen im Dateisystem erkennen will.

3 Entwurf und Implementierung

Dieses Kapitel umfasst den Entwurf und die Implementierung von Plux-CRM. Zuerst wird auf die Herausforderungen beim Entwurf eingegangen. Dazu zählen die Zerlegung in fein-granulare Plugins und die anpassbare Benutzerschnittstelle. Anschließend wird die Datenhaltungsschicht von Plux-CRM detailliert erläutert. Diese umfasst die gesamte Struktur des Datenmodells zusammen mit seinen definierten Klassen. Als nächstes wird die Präsentationsschicht von Plux-CRM angeführt. Sie umfasst den gesamten Aufbau der Benutzerschnittstelle. Den Schluss dieses Kapitels bildet die Struktur der einzelnen Plux-CRM Plugins.

3.1 Herausforderungen beim Entwurf

Aufgrund der speziellen Anforderungen der Software, beispielsweise das Wiederverwenden von Komponenten und das individuelle Gestalten und Konfigurieren der Benutzeroberfläche, treten einige Herausforderungen beim Entwurf der Software auf. Diese inkludieren das Zerlegen in fein-granulare Plugins und die anpassbare Benutzerschnittstelle.

3.1.1 Zerlegung in fein-granulare Plugins

Um Komponenten wiederzuverwenden, ist es wichtig, die Software so in Funktionsblöcke aufzuteilen, dass diese einen gewissen Aufgabenbereich der Software abdecken. Beim Plux-CRM wird dies durch Plux.NET-Plugins realisiert. Diese Plugins sind so fein-granular wie möglich zu gestalten, um auch kleinste Funktionen der Software wiederzuverwenden.

Bei Plux-CRM ist zwischen zwei Zerlegungen zu unterscheiden. Zum einen ist es wichtig, die Anwendung grob in ihre Funktionsbereiche zu unterteilen. Darunter gehören eigene Ansichten für das „Schwarze Brett“, das Verwalten der Kunden, der Geschäfte, der Projekte und der Stichwörter. Jeder dieser Ansichten ist als eigene *View* darzustellen (siehe Kapitel 3.3.2). Eine Ansicht zusammen mit ihren Benutzersteuerelementen ist in ein eigenes Plux.NET-Plugin zusammenzufassen. Wichtig ist, dass die einzelnen Plugins gemeinsam mit ihren Ansichten unabhängig voneinander sind. Somit wird gewährleistet, dass jedes Plugin einzeln in die Software aufgenommen werden kann.

Zum anderen ist die fein-granulare Aufteilung der Benutzersteuerelemente in einer Ansicht zu beachten. Ein Großteil der erwähnten Funktionsbereiche basiert auf denselben Benutzersteuerelementen. Einige davon treten mit derselben Funktionsweise in verschiedenen Ansichten auf. Damit die Wiederverwendung dieser Benutzersteuerelemente gewährleistet ist, sind sie so in Komponenten aufzuteilen, dass man sie ohne großem Aufwand an unterschiedlichsten Stellen einsetzen kann. Aus diesem Grund ist eine Ansicht in immer kleiner werdende Elemente zu unterteilen.

Im Groben ist eine Ansicht in zwei Bereiche zu gliedern. Der *Content*-Bereich auf der linken Seite zeigt die Daten. Der *Task*-Bereich auf der rechten Seite zeigt kontextabhängige Funktionen auf diese Daten. Abbildung 3.1 zeigt die Einteilung einer Ansicht in Content und Task anhand der „Willkommen"-View. Die Herausforderung bei dieser Einteilung liegt darin, dass die Benutzersteuerelemente von einzelnen Ansichten und Plugins, in denen sie sich befinden, abhängig sind. Ist ein Plugin nicht vorhanden, so dürfen alle Benutzersteuerelemente dieses Plugins nicht in der Anwendung erscheinen. Andere Benutzersteuerelemente, die von diesen Elementen abhängig sind, müssen entsprechend auf die Veränderungen der Plugins reagieren.

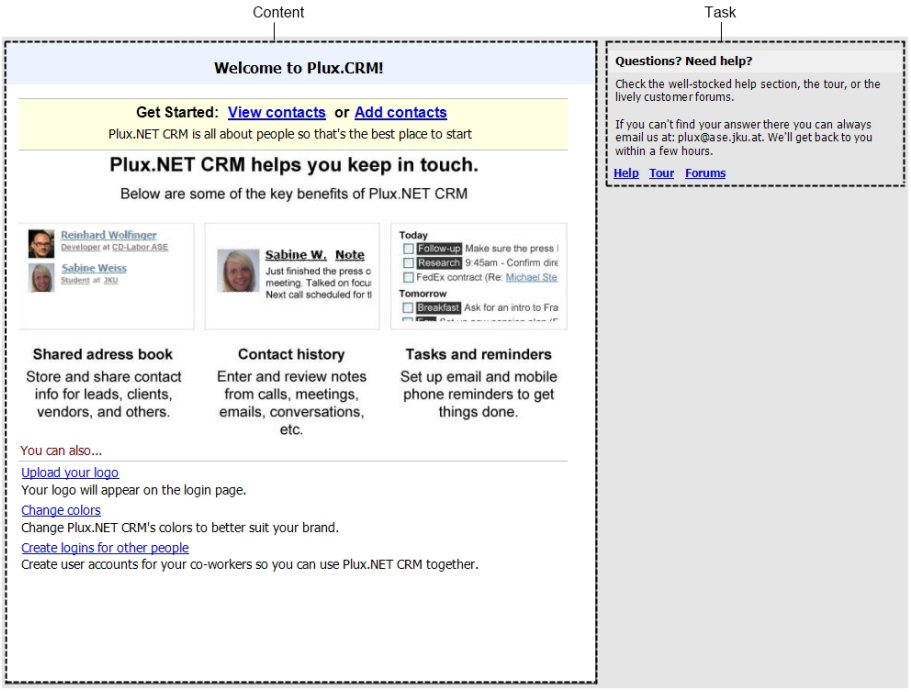

Abbildung 3.1: Einteilung einer Ansicht in Content und Task

3.1.2 Anpassbare Benutzerschnittstelle

Beim Entwurf der Benutzerschnittstelle von Plux-CRM ist darauf zu achten, dass jeder Benutzer die Möglichkeit hat, Plux-CRM so zu gestalten, wie es für seinen Gebrauch am besten ist. Die Herausforderung besteht darin, die Benutzerschnittstelle von Plux-CRM genau auf die Funktionen maßzuschneidern, die ein Benutzer für seinen Gebrauch benötigt. Dies ist nur im Rahmen der in Plux-CRM zur Verfügung gestellten Funktionen möglich. Auch bezieht sich die anpassbare Benutzerschnittstelle nur auf gesamte Ansichten und deren Teilbereiche und nicht auf persönliche Präferenzen einer Benutzeroberfläche, beispielsweise der Farbgebung eines Benutzersteuerelementes.

Jedes Plugin von Plux-CRM enthält eine eigene Ansicht zusammen mit ihren Funktionsbereichen und Benutzersteuerelementen. Die anpassbare Benutzerschnittstelle ist so definiert, dass sie sich auf das individuelle Hinzufügen oder Entfernen der einzelnen Plugins beschränkt. Ein Beispiel: Zu den Basiseinstellungen von Plux-CRM zählen die Plugins für das „Schwarze Brett“ und die Kundenansicht. Diese zwei Ansichten sieht jeder Benutzer, unabhängig von seinen zusätzlichen Einstellungen. Benötigt ein Benutzer zusätzlich die Ansichten für die Auflistung der Projekte und der Geschäfte, so sind in Plux-CRM die Plugins für die Ansicht der Projekte und der Geschäfte zu aktivieren. Alle weiteren vorhandenen Plugins sind in dieser Situation nicht aktiviert. Deshalb sind auch die zugehörigen Ansichten nicht sichtbar.

Plux-CRM soll dem Benutzer die Möglichkeit bieten, diese Einstellungen für das Aktivieren und Deaktivieren der einzelnen Plugins selbst vorzunehmen. Dabei ist zu beachten, dass diese individuellen Einstellungen ohne großen Aufwand, ohne Neuinstallationen und ohne Neustart der Anwendung zu bewerkstelligen sind.

3.2 Datenhaltungsschicht

Die Basis jeder Software bildet die zugrundeliegende Datenhaltungsschicht. Dabei ist es wichtig, sich von Anfang an eine klare Struktur zu überlegen, um spätere Umbauarbeiten zu vermeiden. Das Trennen der Datenhaltungsschicht von der Präsentationsschicht nach dem *Model-View-Controller*-Prinzip (*MVC*-Prinzip) ist von großer Bedeutung. Der Grund für diese Trennung ist die Wiederverwendung der Datenhaltungsschicht mit einer unterschiedlichen Benutzeroberfläche, ohne die Datenhaltungsschicht umschreiben zu müssen. Dies gilt auch für die Präsentationsschicht. Durch die Trennung kann sie mit einer anderen Datenhaltungsschicht eingesetzt werden und muss dafür nicht verändert werden.

In Plux-CRM wurde beim Entwurf der Software genau auf diese Merkmale eingegangen. Die Datenhaltungsschicht wurde in eigene Plugins aufgeteilt, die keinerlei Abhängigkeiten zu den Plugins der Benutzeroberfläche aufweisen. Bei der Datenhaltungsschicht unterschei-

det man zwei Bereiche. Zum einen befinden sich im Plugin `CRM.Contracts` alle Schnittstellen der Datenhaltungsschicht in Form von *Interfaces*. Zu jedem Datentyp und somit zu jeder *Klasse* der Datenhaltungsschicht existiert eine entsprechende Schnittstelle mit den Eigenschaften (*Properties*) und Methoden. in Plux-CRM erkennt man eine Schnittstelle daran, dass im Namen ein „I“ vorangestellt ist. Beispielsweise gibt es zum Datentyp `Contact` die passende Schnittstelle `IContact`.

Die einzelnen Schnittstellen der Datenhaltungsschicht sind im Pugin `CRM.DataModel` implementiert. Dieses Plugin enthält sowohl die gesamte Datenstruktur, als auch Algorithmen für das Importieren und Exportieren von Daten in Form von *CSV*- oder *VCard*-Dateien. Die Datenhaltungsschicht weist verschiedene Datentypen auf, die unterschiedliche Bereiche der Datenhaltungsschicht abdecken.

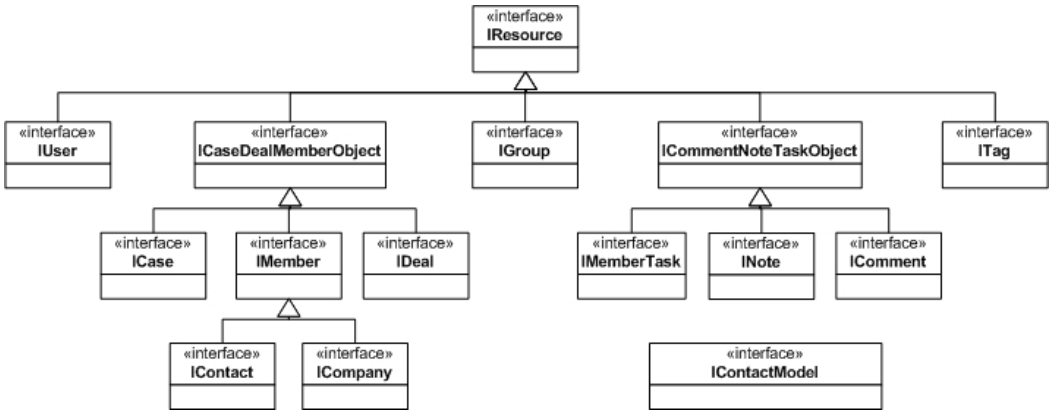

Abbildung 3.2: Hierarchie der Schnittstellen der Datenhaltungsschicht

Abbildung 3.2 zeigt die Hierarchie der Schnittstellen der Datenhaltungsschicht. Die Wurzel aller in der Datenhaltungsschicht verwendeten Schnittstellen bildet die Schnittstelle `IResource`. Zu ihr existiert keine passende Klasse. Sie bildet lediglich den gemeinsamen Nenner für alle Datentypen der Datenhaltungsschicht. `IResource` wird speziell für die Historisierung der letzten Aktivitäten eines Benutzers benötigt.

Die Schnittstellen `ICaseCustomerDealObject`, `ICustomer` und `ICommentNoteTaskObject` sind ebenfalls Ausnahmen. Zu ihnen existieren nur *abstrakte* Klassen. Sie dienen als Hilfsklassen, bei denen nur gewisse Teilbereiche implementiert sind. Alle abgeleiteten Klassen können auf diese Teilbereiche zugreifen. Abstrakte Klassen sind dann nützlich, wenn mehrere Datentypen dieselbe Funktionalitäten benötigen. Dadurch wird das Wiederverwenden dieser Funktionalitäten gewährleistet. In Plux-CRM erkennt man eine abstrakte Klasse daran, dass ihr Name mit „Abstract“ beginnt.

Die abstrakte Klasse `AbstractCaseCustomerDealObject` (siehe Abbildung 3.3), die die zugehörige Schnittstellle `ICaseCustomerDealObject` implementiert, besitzt die Eigenschaften für den Namen (`Name`), die Hintergrundinformation (`BackgroundInfo`)

und die Sichtbarkeit (`Visibility`). Die Sichtbarkeit legt fest, welcher Benutzer oder welche Benutzergruppe diesen Datensatz sehen darf. Zusätzlich enthält die Klasse `AbstractCaseCustomerDealObject` eine Bezeichnung zum Anzeigen des Datentyps (`DisplayName`). Diee Eigenschaft `Guid` sorgt für die eindeutige Erkennung eines Objekts, das von einer abgeleiteten Klasse erzeugt wird. Neben diesen Eigenschaften implementiert `AbstractCaseCustomerDealObject` die Methode `IsEmpty`, die überprüft, ob eine der genannten Eigenschaften bereits gesetzt wurde. `AbstractCaseCustomerDealObject` wird von den Datentypen für einen Kunden (`AbstractCustomer`), für ein Geschäft (`Deal`) und für ein Projekt (`Case`) benötigt, da sie dieselben Eigenschaften für den Namen, die Hintergrundinformation, die Sichtbarkeit und den Anzeigenamen haben.

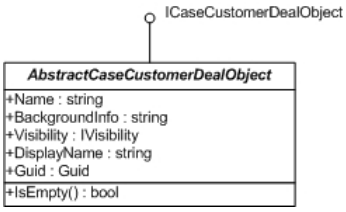

Abbildung 3.3: Die Klasse AbstractCaseCustomerDealObject

Die Schnittstelle `ICommentNoteTaskObject` wird von der abstrakten Klasse `AbstractCommentNoteTaskObject` implementiert (siehe Abbildung 3.4). Diese Klasse enthält Eigenschaften für die Beschreibung (`Description`), den Autor (`Author`) und das Erstellungsdatum (`CreatedOn`). `AbstractCommentNoteTaskObject` bildet die Basis für die Notizen (`Note`) und Kommentare eines Kunden (`Comment`) und seine geplanten Aufgaben (`CustomerTask`), da diese wiederum die angeführten gemeinsamen Eigenschaften besitzen. Auch `AbstractCommentNoteTaskObject` besitzt die Eigenschaft `Guid` und die Methode `IsEmpty`, die diesselben Funktionen wie bei `AbstractCaseCustomerDealObject` übernehmen.

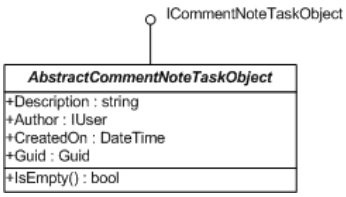

Abbildung 3.4: Die Klasse AbstractCommentNoteTaskObject

3.2.1 Benutzer - `User`

Die Klasse `User` beschreibt einen Benutzer von Plux-CRM. Jeder Benutzer verfügt über einen Benutzernamen (`Username`) und ein Password (`Password`). Außerdem wird über

einen Status (`IsAdmin`) mitgespeichert, ob der Benutzer Administratorrechte besitzt. Hat ein Benutzer diese Rechte, so kann er andere Benutzer verwalten, ihnen Administratorrechte geben oder entziehen und sie in Gruppen einteilen. Zusätzlich werden in der Klasse `User` die Kontaktdaten eines Benutzers (`Contact`) verwaltet. Diese Kontaktdaten enthalten diesselben Informationen wie zu einem Kunden (siehe Kapitel 3.2.3). Jeder Benutzer kann seine eigenen Geschäfts- (`DealTypes`) und Aufgabenarten (`TaskTypes`) festlegen und die den entsprechenden Geschäften und Aufgaben zuordnen. Bei einer Geschäftsart handelt es sich beispielsweise in Hinsicht auf eine Software um ein Design einer Benutzeroberfläche, eine Homepage oder um strategische Vorgehensweisen. Zu den Aufgabenarten zählen zum Beispiel Emails, Telefonate, Faxe und Besprechungen. Zusätzlich wird zu einem Benutzer mitgespeichert, welche Stichwörter von Plux-CRM er sich zuletzt angesehen hat (`RecentlyViewedTags`).

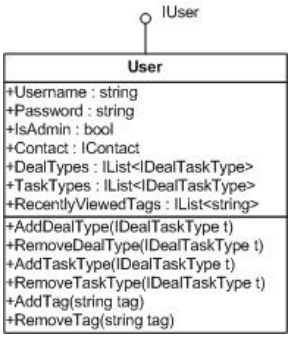

Abbildung 3.5: Die Klasse User

Aus technischer Sicht ist bei der Klasse `User` die entsprechende Schnittstelle `IUser` von großer Bedeutung. Die Schnittstelle `IUser` wird von der Klasse `User` implementiert. Sie enthält dieselben bereits erwähnten Eigenschaften wie ihre zugehörige Klasse `User`. Zusätzlich besitzen sowohl die Schnittstelle als auch die Klasse Methoden zum Hinzufügen und Entfernen der persönlichen Geschäfts- und Aufgabenarten und der zuletzt angesehenen Stichwörter (siehe Abbildung 3.5).

3.2.2 Benutzergruppen - `Group`

Die Klasse `Group` implementiert die Schnittstelle `IGroup`. Sie bezeichnet eine Gruppe, in der verschiedene Benutzer eingeteilt werden. Eine Gruppe ist sinnvoll, um beispielsweise Benutzer unterschiedlicher Abteilungen einer Firma in Plux-CRM entsprechend zu gruppieren. Zu den Eigenschaften einer Gruppe zählen der Name (`Name`), die Auflistung der Benutzer, die ihr zugeordnet sind (`Users`), und die eindeutige Erkennung eines erzeugten Objekts dieser Klasse (`Guid`). Die Klasse `Group` verfügt über zwei Methoden, eine zum Hinzufügen und eine zum Entfernen von Benutzern (siehe Abbildung 3.6).

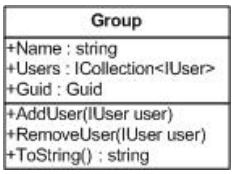

Abbildung 3.6: Die Klasse Group

Gruppen sind in Hinsicht auf Kunden, Projekte und Geschäfte von großer Bedeutung. Je nach Zugehörigkeit kann man festlegen, welche Benutzer oder Benutzergruppen entsprechende Kunden, Projekte und Geschäfte sehen dürfen und welche ihnen verborgen bleiben.

3.2.3 Kontaktpersonen und Firmen - `Contact` und `Company`

Im Zusammenhang mit den Kunden spielt die Schnittstelle `ICustomer` gemeinsam mit der abstrakten Klasse `AbstractCustomer` eine wichtige Rolle. Da sowohl Kontaktpersonen (`Contact`) als auch Firmen (`Company`) als Kunden gelten, stellt die abstrakte Klasse `AbstractCustomer` den Überbegriff *Kunde* dar. Die Klasse `Contact` ist von `AbstractCustomer` abgeleitet und implementiert die Schnittstelle `IContact`. In Plux-CRM existiert auch die Klasse `ICompany`. Sie ist ebenfalls von der Klasse `AbstractCustomer` abgeleitet und implementiert zusätzlich die Schnittstelle `ICompany` (siehe Abbildung 3.7).

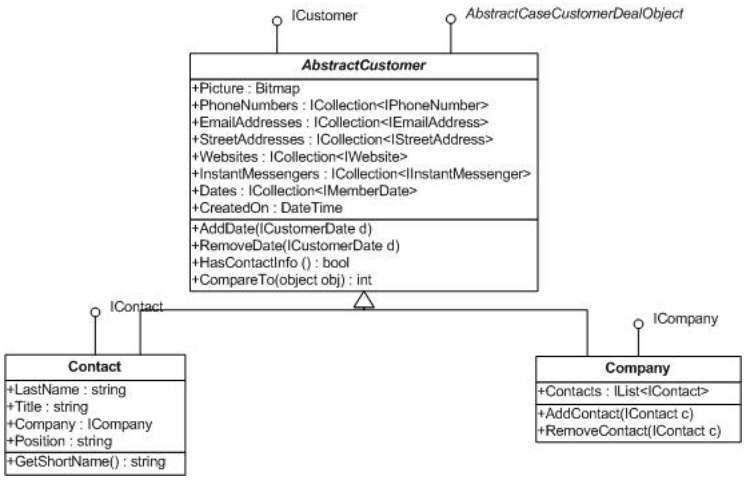

Abbildung 3.7: Die Klassen AbstractCustomer, Contact und Company

Ein Kunde verfügt über einen Namen (`Name`). Hierbei ist zu beachten, dass es sich im Fall einer Kontaktperson um den Vornamen dieser Person handelt. Im Gegensatz dazu handelt es sich bei einer Firma um den Firmennamen. Die Eigenschaft für den Namen gemeinsam mit der Hintergrundinformation und der Sichtbarkeit wer-

den aus der abstrakten Klasse `AbstractCaseCustomerDealObject` übernommen. Zu den weiteren Eigenschaften eines Kunden zählen das Kontaktbild (`Picture`), die Telefonnummern (`PhoneNumbers`), die Email-Adressen (`EMailAddresses`), die Anschriften (`StreetAddresses`), die Webseiten (`Websites`) und die Benutzernamen von *Instant-Messaging*-Programmen (`InstantMessengers`). Instant-Messaging-Programme sind Programme zum Übermitteln von sofortigen Nachrichten. Darunter zählen zum Beispiel *ICQ*, *Skype* und *Sametime*. Ein Benutzer kann Termine zu einem Kunden erstellen (`Dates`).

Eine Kontaktperson (`Contact`) verfügt zusätzlich über einen Nachnamen (`LastName`) und einen Titel (`Title`). Ist eine Kontaktperson in einer Firma beschäftigt, so wird sowohl die Firma (`Company`) als auch die Position, die die Kontaktperson dort ausübt (`Position`), mitgespeichert. In Plux-CRM wird zu einer Kontaktperson oft nur der Vorname und der erste Buchstabe des Nachnames angezeigt. Die Methode `GetShortName` liefert als Ergebnis diesen abgekürzten Namen einer Kontaktperson.

Zu einer Firma (`Company`) existiert zusätzlich eine Auflistung aller Kontaktpersonen, die in der Firma beschäftigt sind (`Contacts`).

3.2.4 Projekte - `Case`

Die Klasse `Case` steht für ein Projekt. Sie implementiert die Schnittstelle `ICase` und ist von der Klasse `AbstractCaseCustomerDealObject` abgeleitet. Aufgrund dieser Ableitung verfügt die Klasse `Case` über alle Eigenschaften (beispielsweise Name, Hintergrundinformation) und Methoden, die in `AbstractCaseCustomerDealObject` definiert sind.

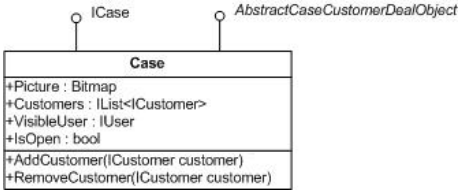

Abbildung 3.8: Die Klasse Case

Wie in Abbildung 3.8 zu sehen, verfügt auch ein Projekt über ein Bild (`Picture`). Zusätzlich können zu einem Projekt entsprechende Kunden (`Customers`) zugeordnet werden, die mit diesem Projekt in Verbindung stehen. Für dieses Zuordnen stehen in der Klasse `Case` jeweils eine Methode zum Hinzufügen und zum Entfernen dieser Kunden zur Verfügung.

3.2.5 Geschäfte - `Deal`

Ein Geschäft wird in Plux-CRM in der Klasse `Deal` definiert. Diese Klasse ist ebenfalls von `AbstractCaseCustomerDealObject` abgeleitet und enthält wie die Klasse `Case` alle geerbten Eigenschaften (Name, Hintergrundinformation) und Methoden. Die Klasse `Deal` implementiert die Schnittstelle `IDeal` (siehe Abbildung 3.9).

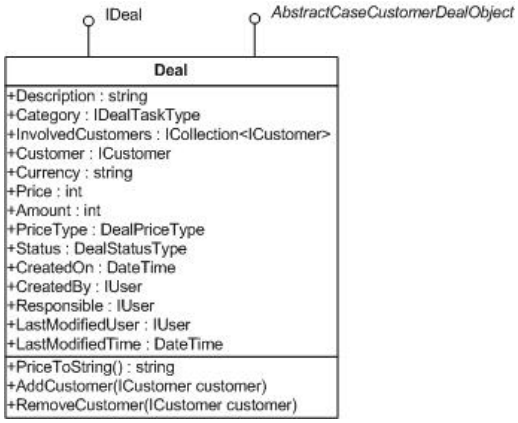

Abbildung 3.9: Die Klasse Deal

Ein Geschäft besitzt zusätzlich zu den von `AbstractCaseCustomerDealObject` geerbten Eigenschaften die Beschreibung eines Geschäfts (`Description`), das Themengebiet, welches das Geschäft behandelt (`Category`), und die Kunden, die in diesem Geschäft involviert sind (`InvolvedCustomers`). Das Hinzufügen und Entfernen eines involvierten Kunden erfolgt über die Methoden `AddCustomer` und `RemoveCustomer`. Bei den involvierten Kunden kann man einen dieser Kunden als Hauptansprechpartner festlegen (`Customer`). Zusätzlich wird ein Benutzer ausgewählt, der für die Vorgehensweise dieses Geschäfts verantwortlich ist (`Responsible`). Ein Geschäft verfügt über weitere Konditionen, beispielsweise die Finanzierung. Dazu ist eine Währung (`Currency`) und der Preis (`Price`) auszuwählen. Bei der Preisauswahl gibt es unterschiedliche Möglichkeiten, zum einen eine Pauschale und zum anderen einen Stunden-, Monats- oder Jahreslohn. Wählt man letzteres aus, so ist ein Betrag pro Stunde, Monat oder Jahr einzugeben. Zusätzlich ist festzulegen, wie oft dieser Betrag ausgezahlt wird (`Amount`). Im Hintergrund speichert Plux-CRM mit, um welche dieser Möglichkeiten es sich bei der Preisauswahl handelt (`PriceType`), damit die Endabrechnung korrekt erstellt wird. Um das Anzeigen des korrekten Preises eines Geschäfts kümmert sich die Methode `PriceToString`.

Neben der Preisauswahl ist der Status des Geschäfts auszuwählen (`Status`). Dabei gibt es drei Unterscheidungen. Legt ein Benutzer ein neues Geschäft an, so ist der Status des Geschäfts noch nicht festgelegt. Das Geschäft befindet sich in einer Art wartendem Zustand. Später kann ein Benutzer den Status so auswählen, dass das Geschäft nicht

abgeschlossen wurde und deshalb als Verlust zu werten ist. Die zweite Möglichkeit ist, dass ein Geschäft zugunsten des Benutzers ausgeht und somit einen Gewinn für den Benutzer hervorbringt.

Plux-CRM speichert zu einem Geschäft zusätzliche Informationen. Dazu zählen das Erstellungsdatum (`CreatedOn`), der Benutzer, der dieses Geschäft erstellt hat (`CreatedBy`), der Zeitpunkt, zu dem die Details eines Geschäfts geändert wurden (`LastModifiedTime`) und welcher Benutzer diese Informationen zuletzt geändert hat (`LastModifiedUser`). Diese Informationen sind nötig, um die Änderungen eines Geschäfts am „Schwarzen Brett“ anzuzeigen.

3.2.6 Aufgaben - `CustomerTask`

Die Klasse `CustomerTask` stellt in Plux-CRM eine Aufgabe dar. Sie implementiert die Schnittstelle `ICustomerTask` und ist von der abstrakten Klasse `AbstractNoteTaskObject` abgeleitet (siehe Abbildung 3.10).

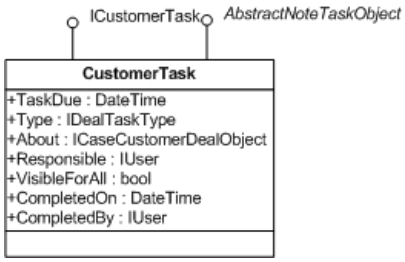

Abbildung 3.10: Die Klasse CustomerTask

Neben den von der abstrakten Klasse geerbten Eigenschaften, beispielsweise der Beschreibung und dem Autor, enthält die Klasse `CustomerTask` einen Zeitpunkt, zu dem die Aufgabe zu erledigen ist (`TaskDue`). Jeder Benutzer kann eine Aufgabe einer Kategorie (`Type`) zuweisen. Zu diesen Kategorien zählen Emails, Telefonate und Besprechungen. Diese Kategorien sind vom Benutzer selbst zu definieren (siehe Kapitel 3.2.1). Eine Aufgabe wird meist zu einem Kunden, einem Projekt oder einem Geschäft (`About`) verfasst. Es besteht auch die Möglichkeit, dass ein Benutzer sich selbst oder anderen Benutzern interne Aufgaben zuweist, ohne dass die einem Kunden, Projekt oder Geschäft zugeordnet sind.

Wie bei einem Geschäft wird auch bei einer Aufgabe ein verantwortlicher Benutzer (`Responsible`) festgelegt, der sich um diese Aufgabe zu kümmern hat. Zusätzlich wählt der Benutzer aus, ob eine Aufgabe für alle oder nur für bestimmte Benutzer oder Benutzergruppen sichtbar ist (`VisibleForAll`). Wenn eine geplante Aufgabe erledigt ist, so kann der Benutzer diese Aufgabe dementsprechend markieren. In diesem Fall speichert Plux-CRM

im Hintergrund mit, zu welchem Zeitpunkt diese Aufgabe erledigt wurde (`CompletedOn`) und welcher Benutzer die Aufgabe als erledigt markiert hat (`CompletedBy`).

3.2.7 Notizen und Kommentare - `Note` und `Comment`

Eine Notiz wird in Plux-CRM in der Klasse `Note` verwaltet. Diese Klasse implementiert die Schnittstelle `INote` und ist wie die Klasse `CustomerTask` von der abstrakten Klasse `AbstractCommentNoteTaskObject` abgeleitet. Auch `Note` besitzt weitere Eigenschaften als die von `AbstractCommentNoteTaskObject` geerbten Eigenschaften, wie der Beschreibung (siehe Abbildung 3.11).

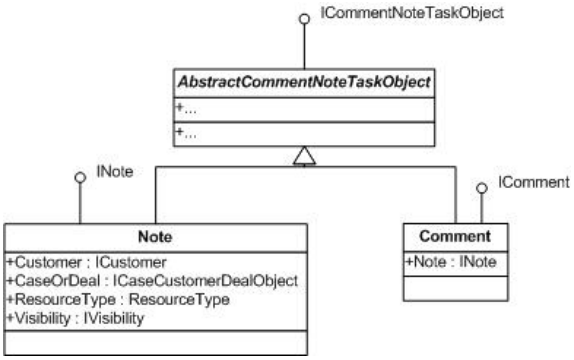

Abbildung 3.11: Die Klassen Note und Comment

Eine Notiz kann zu einem Kunden (`Customer`) und zusätzlich zu einem Projekt oder einem Geschäft (`CaseOrDeal`) verfasst werden. Mit der Eigenschaft `ResourceType` wird definiert, mit welchem Datentyp (`Customer`, `Case` oder `Deal`) die Notiz tatsächlich in Verbindung steht. Auch bei einer Notiz hat ein Benutzer die Möglichkeit, bestimmte Benutzer oder Benutzergruppen festzulegen, die Zugriff auf diese Notiz haben (`Visibility`).

Jeder Benutzer kann zu jeder Notiz seine eigenen Kommentare abgeben. In Plux-CRM sind Kommentare in der Klasse `Comment` dargestellt. Diese Klasse implementiert die Schnittstelle `IComment` und ist ebenfalls eine Ableitung der abstrakten Klasse `AbstractNoteTaskObject`. `Comment` besitzt als zusätzliche Eigenschaft die Notiz (`Note`), zu die der Kommentar verfasst wurde.

3.2.8 Stichwörter - `Tag`

Die Klasse `Tag` steht in Plux-CRM für Stichwörter. Diese Klasse verfügt über zwei Schnittstellen, zum einen die Basis-Schnittstelle `IResource` und zum anderen die C#-Schnittstelle `IComparable`, die zum Vergleichen zweier Stichwörter nötig ist (siehe Abbildung 3.12).

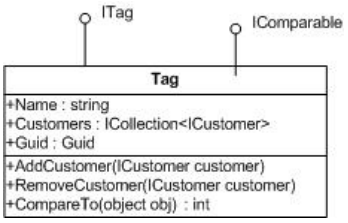

Abbildung 3.12: Die Klasse Tag

Ein Stichwort hat einen Namen (`Name`), eine Auflistung von Kunden, die mit diesem Stichwort markiert sind (`Customers`), und eine eindeutige Erkennung eines erstellten Objekts der Klasse `Tag` (`Guid`). Jeder Benutzer kann jeden Kunden mit seinen eigenen Stichwörteren markieren, um diese zu einem späteren Zeitpunkt wieder zu finden. Auch die Zusammengehörigkeit dieser Kunden kann mit Stichwörtern festgelegt werden.

Um einen Kunden mit einem neuen Stichwort zu versehen oder ein bereits vorhandenes Stichwort zu entfernen, besitzt die Klasse `Tag` zwei entsprechende Methoden (`AddCustomer`, `RemoveCustomer`). Die Methode `CompareTo` ist von der Schnittstelle `IComparable` überschrieben und vergleicht zwei Stichwörter miteinander, um ihre alphabetische Reihenfolge zu ermitteln.

3.2.9 Datenmodell - `ContactModel`

Die Verwaltung der beschriebenen Datentypen übernimmt die Klasse `ContactModel`. Sie ist als Basis des in Plux-CRM verwendeten Datenmodells zu sehen. Diese Klasse implementiert die Schnittstelle `IContactModel` (siehe Abbildung 3.13).

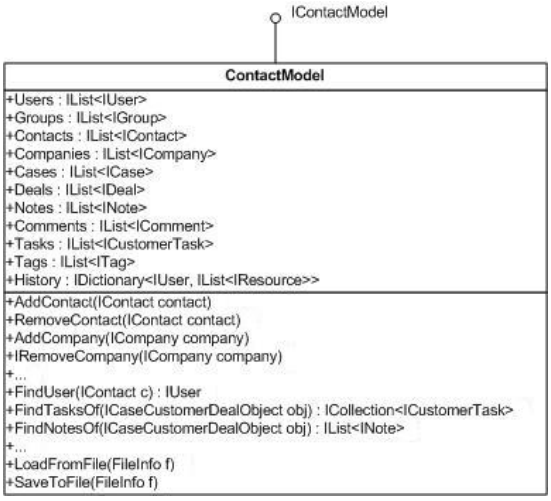

Abbildung 3.13: Die Klasse ContactModel

In der Klasse `ContactModel` sind alle Auflistungen zu den Benutzern, Benutzergruppen, Kontaktpersonen, Firmen, Projekten und Geschäften als die Eigenschaften `Users`, `Groups`, `Contacts`, `Companies`, `Cases` und `Deals` dargestellt. Zusätzlich besitzt `ContactModel` Auflistungen zu den Notizen (`Notes`), Kommentaren (`Comments`) und Aufgaben (`Tasks`), die in Plux-CRM verfasst wurden. Das Datenmodell verwaltet die gesamte Historisierung (`History`), die mitspeichert, welcher Benutzer welche Aktivitäten zuletzt in Plux-CRM ausgeübt hat. Zu diesen Aktivitäten zählen das Erfassen eines neuen Geschäfts, das Ändern des Status eines Geschäfts, das Anlegen einer neuen Notiz oder eines neuen Kommentares und das Erledigen einer Aufgabe. Diese Aktivitäten sind für jeden Benutzer am „Schwarzen Brett“ zu sehen.

Neben diesen Eigenschaften besitzt die Klasse `ContactModel` Methoden zum Hinzufügen und Entfernen von Objekten aller Datentypen des Datenmodells, wie Kunden, Projekten und Geschäften (zum Beispiel `AddContact`, `AddCompany`, `RemoveContact`, `RemoveCompany`). Das Auflisten bestimmter Objekte eines Datentyps erfolgt in der Klasse `ContactModel` über eigene Methoden, die meist mit *Find* beginnen. Beispielsweise sucht die Methode `FindUser` den Benutzer zu einer bestimmten Kontaktperson, die als Parameter `IContact c` übergeben wird. Das Laden und Speichern der vorhandenen Daten erfolgt ebenfalls in der Klasse `ContactModel` über zwei entsprechende Methoden (`LoadFromFile` und `SaveToFile`).

3.3 Präsentationsschicht

Neben der Datenhaltungsschicht spielt die Präsentationsschicht eine ebenso große Rolle. Die Benutzeroberfläche von Plux-CRM ist mit Windows Forms 2.0 [5] und den speziell von Plux.NET zur Verfügung gestellten Widgets realisiert. Aufgrund der Anforderung, die Benutzeroberfläche zur Laufzeit individuell zu konfigurieren, ist die Benutzeroberfläche in bestimmte Teilbereiche gegliedert. Hierbei gibt es zwei Unterscheidungen, zum einen die grobe, inhaltliche Gliederung und zum anderen die fein-granulare Gliederung. Auf die grobe, inhaltliche Gliederung wurde bereits im Kapitel 1.2 genauer eingegangen.

Abbildung 3.14 zeigt die fein-granulare Gliederung von Plux-CRM anhand der „Willkommen“-Seite. Diese Gliederung besteht aus den Bereichen Dashboard (1), View (2), Content (3), Task (4) und Actions (5).

3.3.1 Dashboard

Die Klasse `Dashboard` stellt das Hauptfenster von Plux-CRM dar, auf dem sich alle weiteren Benutzersteuerelemente befinden. Aufgrund der Darstellung als Hauptfenster ist die Klasse `Dashboard` von der .NET-Klasse `Form` abgeleitet. Nur die Titelleiste mit dem Na-

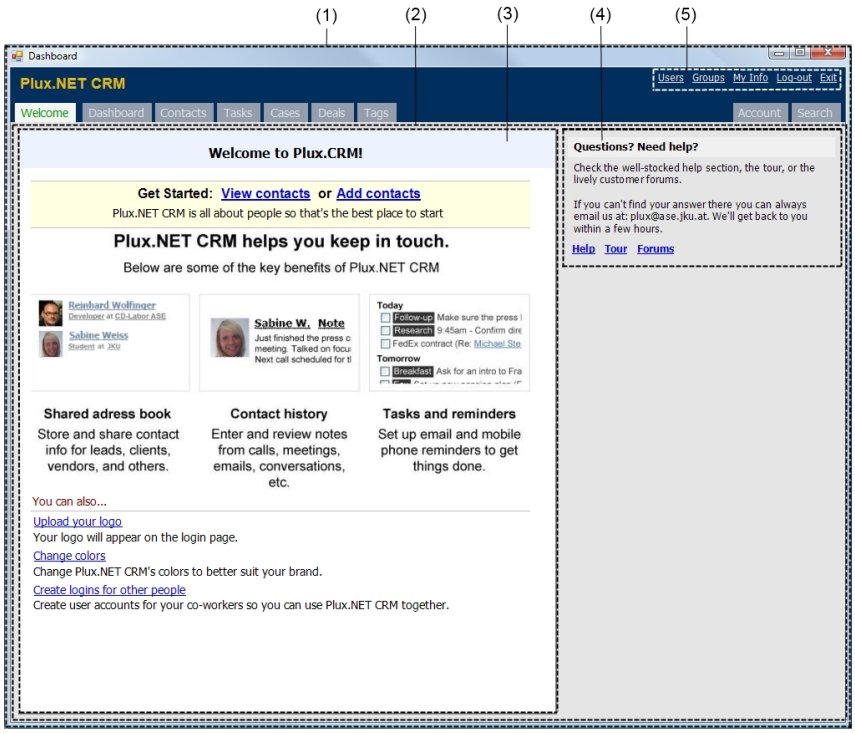

Abbildung 3.14: Anzeigebereiche in der Benutzerschnittstelle von Plux-CRM

men der Anwendung, den Standardschaltflächen zum Minimieren, Maximieren und Schließen der Anwendung und einem leeren Inhalt bilden das Design der Klasse `Dashboard`.

Damit Plux.NET das Hauptfenster beim Programmstart lädt, implementiert das `Dashboard` zusätzlich die Schnittstelle `IStartup`, die bereits in Plux.NET enthalten ist. Diese Schnittstelle verfügt über eine einzige Methode (`Run`), die zum Startzeitpunkt der Anwendung aufgerufen wird.

Neben dieser Schnittstelle implementiert das `Dashboard` eine zweite Schnittstelle, das `IDashboard`. Diese Schnittstelle besitzt vier Eigenschaften (siehe Abbildung 3.15). Dazu zählen der angemeldete Benutzer (`Identity`), die aktuell angezeigte Ansicht (`CurrentView`), alle Karteireiter, die im Titel von Plux-CRM zu sehen sind (`ViewButtons`) und alle Links, die sich im rechten oberen Teil von Plux-CRM befinden (`Actions`).

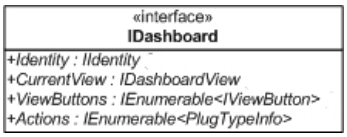

Abbildung 3.15: Die Schnittstelle IDashboard

Die Schnittstelle `IDashboard` wird benötigt, um die einzelnen Steckplätze des Dashboardes verwalten zu können. Mit dem Einstecken einer Komponente in die Steckplätze `LogonView`

und `View` des Dashboardes wird gleichzeitig die Eigenschaft `CurrentView` der Schnittstelle `IDashboard` gesetzt. Äquivalent dazu wird beim Ausstecken einer Komponente aus diesen Steckplätzen des Dashboardes die Eigenschaft `CurrentView` wieder zurückgesetzt.

In Abbildung 3.16 sind die einzelnen Steckplätze des Dashboardes zu erkennen. Zusätzlich zeigt die Abbildung, wie die Komponente *Dashboard* in der Kern-Komponente *Core* integriert ist.

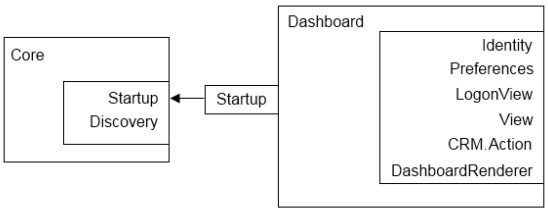

Abbildung 3.16: Die Kern-Komponente Core mit dem Dashboard

Der Steckplatz `Identity` ist für den Benutzer verantwortlich, der sich bei Plux-CRM anmeldet. Er verwaltet außerdem das Setzen der gleichnamigen Eigenschaft der `IDashboard`-Schnittstelle. Zu Beginn von Plux-CRM ist der Steckplatz `Identity` leer, da sich jeder Benutzer erst anmelden muss, bevor er die Funktionen von Plux-CRM nützen kann. Mit der erfolgreichen Anmeldung des Benutzers wird die Komponente `Identity` mit dem angemeldeten Benutzer erzeugt und in den gleichnamigen Steckplatz eingesteckt. Anhand der gespeicherten Informationen der Komponente `Persistor` aus der Plux.NET-Bibliothek, die sich im Steckplatz `Preferences` des Dashboardes befindet, ist es möglich, dass der zuletzt angemeldete Benutzer gespeichert wird. Beim nächsten Starten von Plux-CRM kommen die vorher gespeicherten Informationen zum Einsatz und der letzte Benutzer wird automatisch angemeldet. In diesem Fall kümmert sich das Dashboard um die automatische Anmeldung des Benutzers und befüllt den Steckplatz `Identity` entsprechend. Das Befüllen des Steckplatzes `Preferences` und somit das automatische Anmelden des Benutzers funktionieren nur dann, wenn Plux-CRM die Komponente `IsolatedStoragePersistor` findet. Diese Komponente ist Teil von Plux.NET und enthält den `Persistor`. Der `Persistor` speichert die Komponenten einer Plux.NET-Anwendung. Zu diesen Informationen zählen der angemeldete Benutzer oder die letzte Position und Größe einer Plux.NET-Anwendung. Anhand dieser Komponente werden beim nächsten Starten einer Plux.NET-Anwendung die gespeicherten Informationen wieder hergestellt.

Der Steckplatz `Identity` steht in direktem Zusammenhang mit den Steckplätzen `LogonView` und `View` des Dashboardes. Der Steckplatz `LogonView` wird nur dann befüllt, wenn kein Benutzer bei Plux-CRM angemeldet ist und somit der Steckplatz `Identity` des Dashboardes leer ist. Ist kein Benutzer angemeldet, so erscheint in Plux-CRM das Anmeldefenster. Meldet sich ein Benutzer erfolgreich an, so wird dem Steckplatz `Identity` eine

Komponente hinzugefügt, gleichzeitig der Steckplatz `LogonView` entleert und der Steckplatz `View` mit einer entsprechenden Komponente befüllt. Somit können die Steckplätze `LogonView` und `View` nie gleichzeitig leer oder befüllt sein. Sie sind beide vom Zustand des Steckplatzes `Identity` abhängig.

Für jede Komponente, die der Steckplatz `CRM.Action` des Dashboardes registriert, wird im rechten oberen Bereich von Plux-CRM ein Link hinzugefügt. Der Steckplatz `DashboardRenderer` verwaltet das Aussehen des Hauptfensters. In diesen Steckplatz wird eine gleichnamige Komponente eingesteckt. Diese Komponente befüllt das Hauptfenster mit der Titelleiste. Auf dieser Titelleiste befinden sich im linken oberen Bereich der Name der Anwendung, im rechten oberen Bereich die Links und im unteren Bereich die Karteireiter der verschiedenen Ansichten (siehe Abbildung 3.16).

Ein *Renderer* ist immer dann nötig, wenn ein Benutzersteuerelement dynamisch befüllt wird. Jeder Renderer hat meist seine eigene Schnittstelle, bei der gleichzeitig der Steckplatz definiert wird, in dem eine Renderer-Komponente eingesteckt wird.

Die Komponente *DashboardRenderer* implementiert die Schnittstelle `IDashboardRenderer` (siehe Abbildung 3.17). Diese enthält die Methode `Render`, die aufgerufen wird, um den Inhalt des übergebenen Dashboardes zu befüllen. Die Methode `AddViewButton` wird verwendet, wenn eine Komponente im Steckplatz `View` des Dashboardes registriert wird. Diese Methode zeichnet in der Titelleiste einen neuen Karteireiter für diese Komponente. Das Gegenteil dazu bildet die Methode `RemoveViewButton`. Sie wird aufgerufen, wenn der Steckplatz `View` die Registrierung einer Komponente aufhebt. Gleichzeitig wird in der Titelleiste der dazupassende Karteireiter entfernt. Die Methoden `AddAction` und `RemoveAction` übernehmen diesselbe Funktionen, nur für die Komponenten des Dashboard-Steckplatzes `CRM.Action` und für die Links in der Titelleiste. Die Methode `DisplayView` sorgt dafür, dass die übergebene Ansicht in Plux-CRM angezeigt wird. Die Eigenschaft `Authenticated` überprüft, ob ein Benutzer angemeldet ist. Ist kein Benutzer angemeldet, dann erscheint das Anmeldefenster und in der Titelleiste ist nur der Name der Anwendung und ein Link zum Beenden der Anwendung zu sehen. Ist ein Benutzer hingegen angemeldet, so sind zusätzlich alle Karteireiter und Links sichtbar.

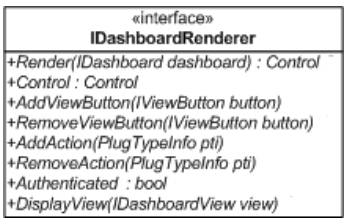

Abbildung 3.17: Die Schnittstelle IDashboardRenderer

Einsteckvorgang des Dashboardes

Wird nun Plux-CRM gestartet, so erkennt die Kern-Komponente *Core* (siehe Kapitel 2.1.4) das Dashboard als Startelement und erzeugt davon ein Objekt. Anschließend bildet es zu diesem Objekt einen entsprechenden Stecker und steckt diesen in den Steckplatz `Startup` der Kern-Komponente ein. Anschließend erzeugt das Dashboard aus allen Komponenten, die in seinen Steckplätzen registriert sind, entsprechende Objekte und Stecker und steckt diese in seine Steckplätze. Auch in diesem Fall gilt, zuerst werden in allen Host-Steckplätzen erzeugte Contributor-Komponente aufgenommen, und dann werden die Steckplätze dieser erzeugten Contributor-Komponenten befüllt (*breadth-first*). Dieser Vorgang wird solange aufgerufen, bis die letzte Contributor-Komponente erzeugt wurde, bei der keine Steckplätze zu befüllen sind.

Mit dem Einstecken des Dashboardes in den Steckplatz `Startup` der Kern-Komponente öffnet sich das Hauptfenster von Plux-CRM. Das Hauptfenster ist bereits mit seinen Ansichten und deren Benutzersteuerelementen befüllt. Je nach eingesteckter Ansicht verändert sich der Inhalt des Hauptfensters entsprechend.

3.3.2 View

Eine Ansicht (*View*) befüllt den Inhalt des Hauptfensters und nimmt somit den Hauptteil des Dashboardes ein (siehe Abbildung 3.14). Für jeden Funktionsbereich, wie in Kapitel 1.2 beschrieben, gibt es eine eigene Ansicht. Beispielsweise zeigt *DashboardView* die letzten Aktivitäten eines Benutzer, *ContactsView* listet alle Kunden auf und *AccountView* zeigt die Einstellungen zum Anpassen der Benutzeroberfläche. Jede dieser Ansichten ist nach demselben Schema aufgebaut. Auf der linken Seite einer Ansicht befindet sich der eigentliche Inhalt (*Content*). Auf der rechten Seite ist ein schmaler Streifen mit den Aufgaben einer Ansicht (*Tasks*) vorzufinden. Die einzige Ausnahme in dieser Hinsicht bildet die Ansicht für das Anmeldefenster (*LogonView*). Sie besteht nur aus einem Inhalt, der sich über die gesamte Breite des Hauptfensters ausdehnt.

Eine Ansicht ist eine Komponente, die in den Steckplatz `View` des Dashboard eingesteckt wird. In diesem Steckplatz können mehrere Ansicht-Komponenten registriert sein, eingesteckt wird jedoch immer nur genau eine Komponente. Für jede Ansicht, die registriert ist, wird ein entsprechender Karteireiter im Dashboard erzeugt. Ausnahmen sind jene Ansichten, die über einen Link in der Titelleiste aufgerufen werden. Sie bekommen keinen eigenen Karteireiter. Auch die Ansicht für das Anmeldefenster hat weder einen eigenen Karteireiter, noch wird sie im Steckplatz `View` des Dashboardes eingesteckt. Für sie ist der Steckplatz `LogonView` des Dashboardes reserviert.

Jede Ansicht ist von der abstrakten Klasse `AbstractDashboardView` abgeleitet. In dieser Klasse sind die wichtigsten Methoden und Eigenschaften bereits vorprogrammiert.

`AbstractDashboardView` implementiert zudem die Schnittstelle `IDashboardView`. Bei dieser Schnittstelle werden alle Plux.NET-Parameter definiert, die eine Ansicht-Komponente benötigt. Zusätzlich wird der Name des Steckplatzes (*CRM.DashboardView*) festgelegt, in dem alle Komponenten mit dieser Schnittstelle eingesteckt werden (siehe Listing 3.1).

Listing 3.1: Die Schnittstellen IDashboardElement und IDashboardView

```
1  [SlotDefinition("CRM.DashboardElement")]
2  [Param("OrderIndex", typeof(float), 0f)]
3  public interface IDashboardElement {
4    float OrderIndex { get; }
5    Control Control { get; }
6    bool Visible { get; }
7    ...
8  }
9
10 [SlotDefinition("CRM.DashboardView")]
11 [Param("Name", typeof(string))]
12 [Param("ViewGroup", typeof(ViewGroup))]
13 ...
14 public interface IDashboardView : IDashboardElement {
15   string Name { get; }
16   Guid Guid { get; }
17   ViewGroup ViewGroup { get; }
18   IContent Content { get; }
19   IEnumerable<ITask> Tasks { get; }
20 }
```

Zu den Eigenschaften der Schnittstelle `IDashboardView` zählt der Name der Ansicht (`Name`). Ist eine Ansicht aktiviert, so bekommt ihr zugehöriger Karteireiter in der Titelleiste seinen Namen über diese Eigenschaft. Ist eine Ansicht deaktiviert, so bekommt ihr zugehöriger Karteireiter seinen Namen über den Plux.NET-Parameter `Name` der Ansicht-Komponente. Die Eigenschaft `Guid` legt die eindeutige Erkennung einer Ansicht fest. Die Eigenschaft `ViewGroup` bestimmt, in welchem Bereich der Titelleiste sich der dazugehörige Karteireiter befindet.

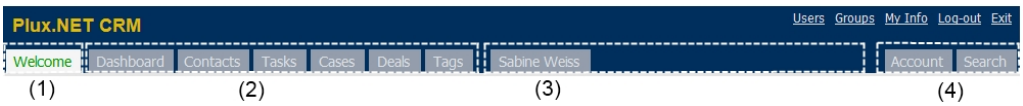

Abbildung 3.18: Einteilung der Karteireiter in der Titelleiste

Abbildung 3.18 zeigt die vier Unterscheidungen der Karteireiter-Bereiche. Links in der Titelleiste der Anwendung befindet sich der Bereich *Overview* (1), daneben der Bereich *Feature* (2), wieder daneben der Bereich *Document* (3) und am rechten äußeren Rand der Bereich *Options* (4). Nur die Karteireiter des Bereiches *Document* werden nicht von Anfang an angezeigt. Sie werden nur dann angezeigt, wenn der Benutzer eine Detail-Ansicht zu einem Kunden, einem Projekt oder einem Geschäft öffnet. Die anderen drei Karteireiter-Bereiche sind immer sichtbar. Die Eigenschaft `ViewGroup` wird ebenfalls über

den gleichnamigen Plux.NET-Parameter gesetzt. Zusätzlich enthält `IDashboardView` die Eigenschaft `Content`. Sie enthält den eigentlichen Inhalt einer Ansicht. Diese Eigenschaft wird dann gesetzt, wenn eine Ansicht, die von `AbstractDashboardView` abgeleitet ist, einen entsprechenden Steckplatz für den Inhalt enthält und dieser auch mit einer Komponente befüllt ist. Diese Komponente bildet den Inhalt der Ansicht. Neben dem Inhalt besitzt die Schnittstelle die Eigenschaft `Tasks`, die alle Aufgaben auf der rechten Seite einer Ansicht auflistet. Dieser Liste werden nur dann Einträge hinzugefügt, wenn eine Ansicht einen entsprechenden Steckplatz für die Aufgaben enthält. Jede Komponente in diesem Steckplatz ist ein Eintrag in der Liste `Tasks`.

Die Schnittstelle `IDashboardView` ist zusätzlich eine Ableitung von `IDashboardElement`. Die Schnittstelle `IDashboardElement` steht für jeden Teilbereich des Dashboardes, gleichgültig, ob es sich dabei um eine Ansicht, einen Inhalt oder eine Aufgabe handelt. Sie bildet somit die Wurzelschnittstelle aller Elemente der Benutzeroberfläche des Dashboardes. `IDashboardElement` legt die Steckplatz-Definition *CRM.DashboardElement* fest. Zusätzlich enthält sie den Plux.NET-Parameter *OrderIndex*, der die gleichnamige Eigenschaft setzt und die Reihenfolge festlegt, in der das Element in der Benutzeroberfläche vorkommt. Beispielsweise wird so bei jedem Karteireiter-Bereich die Reihenfolge der Karteireiter festgelegt. Ist dieser Parameter nicht gesetzt, so wird die Reihenfolge so zusammengestellt, wie der Dashboard-Steckplatz `View` die Ansicht-Komponenten registriert. Die Eigenschaft `Control` der Schnittstelle `IDashboardElement` stellt das Benutzersteuerelement dar, mit dem die Ansicht aufgebaut ist. Die Eigenschaft `Visible` legt fest, ob das Benutzersteuerelement angezeigt wird oder im Hintergrund verschwindet.

Beispiel einer View

Abbildung 3.19 zeigt anhand der Komponente *WelcomeView* ein Beispiel für eine Ansicht. Wie zu erkennen ist, befindet sich die Komponente im *View*-Steckplatz des Dashboardes. In der Anwendung wird die „Willkommen“-Seite angezeigt (siehe Abbildung 3.14). *WelcomeView* verfügt über drei Steckplätze. *CRM.WelcomeContent* verwaltet die Komponente mit dem Inhalt der Ansicht. Dieser Steckplatz kann genau eine Komponente aufnehmen. Im Steckplatz *CRM.WelcomeTasks* können mehrere Komponenten aufgenommen werden. Er verwaltet alle Aufgaben zu dieser Ansicht. Der Steckplatz *CRM.DashboardViewRenderer* sorgt für das Aussehen der Ansicht. Er ist über eine ähnliche Schnittstelle definiert wie der Steckplatz *DashboardRenderer* des Dashboardes. Jede Ansicht von Plux-CRM besitzt einen *CRM.DashboardViewRenderer*-Steckplatz, in den jeweils eine eigene Renderer-Komponente eingesteckt wird. Diese Komponenten sind alle vom selben Typ, sodass ein einheitliches Aussehen aller Ansichten gewährleistet wird. Zu diesem einheitlichen Aussehen zählt das Erzeugen von entsprechenden Karteireitern, das Anordnen des Inhaltes auf der linken und der Aufgaben auf der rechten Seite einer Ansicht.

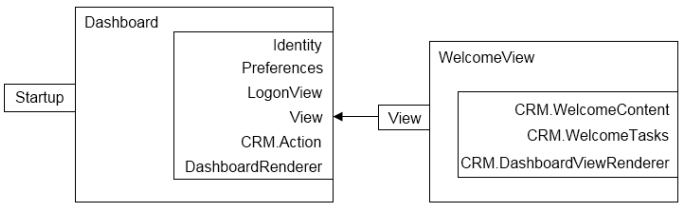

Abbildung 3.19: Die Komponente Dashboard mit der WelcomeView

3.3.3 Content

Den eigentlichen Inhalt einer Ansicht stellt der *Content* dar. Er befindet sich auf der linken Seite einer Ansicht und nimmt dort den größten Teil der Ansicht ein. Wie jedes Element der Plux-CRM Benutzeroberfläche wird auch der Inhalt in verschiedene Bereiche unterteilt. Der obere Bereich ist für den Titel reserviert. Beim Titel kann es sich entweder um einen Text oder um ein eigenes Benutzersteuerelement handeln. Der untere, größere Teil wird aus den *ContentControls* gebildet. Dabei kann eine beliebige Anzahl an ContentControls verwendet werden. Abbildung 3.20 zeigt anhand der „Willkommen“-Ansicht die Unterteilung des Inhalts. Der Titel ist in diesem Fall ein eigenes Benutzersteuerelement (1). Der untere Teil der Ansicht weist zwei unterschiedliche ContentControls auf (2a+b).

Abbildung 3.20: Gliederung des Content

Jeder Inhalt einer Ansicht ist eine eigene Komponente und bildet eine Ableitung der abstrakten Klasse `AbstractContent`. In dieser Klasse sind bereits die wichtigsten Eigenschaften für die Definitionen der Steckplätze und Stecker definiert. `AbstractContent` ist von `AbstractElement` abgeleitet und implementiert zusätzlich die Schnittstelle `IContent`. Diese Schnittstelle ist nötig, damit eine Ansicht eine Klasse als seinen Inhalt erkennt. Listing 3.2 zeigt die Schnittstelle `IContent`.

Listing 3.2: Die Schnittstelle IContent

```
1 [SlotDefinition("CRM.Content")]
2 public interface IContent : IDashboardElement {
3   string Name { get; }
4   string Title { get; }
5   Control TitleControl { get; }
6   IEnumerable<IDashboardElement> Elements { get; }
7 }
```

Wie bei den meisten Schnittstellen in Plux-CRM wird auch zu der Schnittstelle `IContent` eine Steckplatz-Definition festgelegt (*CRM.Content*). Zu erkennen ist, dass `IContent` zusätzlich von `IDashboardElement` abgeleitet ist. Diese Ableitung wird benötigt, falls eine Klasse, die einen Inhalt einer Ansicht darstellt, nicht von der abstrakten Klasse `AbstractContent` und somit auch nicht von der abstrakten Klasse `AbstractElement` abgeleitet ist. Damit die Klasse mit dem Inhalt trotzdem alle nötigen Schnittstellen besitzt und keine wichtigen Eigenschaften und Methoden untergehen, wurde die Ableitung der `IContent`-Schnittstelle von `IDashboardElement` eingeführt. Die Schnittstelle besitzt neben den von `IDashboardElement` geerbten Eigenschaften zusätzlich einen Namen (`Name`), einen Text als Titel (`Titel`) und ein Benutzersteuerelement, das alternativ als Titel dargestellt werden kann (`TitleControl`). Dabei ist folgendes zu beachten. Ist ein Benutzersteuerelement als Titel gesetzt, so wird dieses in der Ansicht angezeigt. Der Text als Titel verschwindet im Hintergrund, gleichgültig, ob die Eigenschaft gesetzt wurde. Ist hingegen das Benutzersteuerelement leer, so erscheint nur die Eigenschaft `Titel`. Jeder Text, der den Titel eines Inhaltes darstellt, wird aufgrund eines speziellen Renderers nach demselben Prinzip dargestellt. Die Darstellung der Benutzersteuerelemente variiert hingegen, da der Renderer bei ihnen keine Wirkung zeigt. Die Eigenschaft `Elements` bildet eine Auflistung aller untergeordneten Elemente (ContentControls) im unteren Teil des Inhaltes. Die Anzahl dieser Elemente ist von Ansicht zu Ansicht unterschiedlich. Sie ist abhängig von der Darstellung des Inhaltes. Je mehr Informationen auf einem Inhalt dargestellt werden, desto mehr untergeordnete Elemente werden benötigt. Die untergeordneten Elemente sind möglichst klein zu halten und sollen nur eine kleinen Teilbereich des Inhaltes übernehmen. Somit kann ein Teilbereich schnell und einfach durch einen neuen ausgetauscht werden, ohne den kompletten Inhalt neu zu gestalten. Der neue Teilbereich des Inhaltes nimmt einfach den Platz des alten ein.

Die untergeordneten Elemente (*ContentControls*) sind nach demselben Prinzip wie der Content aufgebaut. Sie haben sowohl einen Titel, der entweder nur aus einem Text oder aus einem Benutzersteuerelement besteht, und einen Bereich für den Inhalt. Dieser kann sich aus nur einem Benutzersteuerelement oder aus einer Ansammlung von untergeordneten Komponenten zusammensetzen. Die untergeordneten Komponenten sind ebenfalls vom Typ `ContentControl`. Somit kann eine beliebig tiefe Verschachtelung an untergeordneten Elementen zusammengestellt werden.

Beispiel eines Contents

Abbildung 3.21 zeigt den eingesteckten Inhalt anhand der „Willkommen“-Seite (siehe Abbildung 3.20). Zu sehen ist, dass sich die *WelcomeContent*-Komponente im *CRM.WelcomeContent*-Steckplatz der Ansicht befindet. Die *WelcomeContent*-Komponente besitzt drei Steckplätze. Der Steckplatz *CRM.WelcomeTitle* kann genau eine Komponente aufnehmen, die ein Benutzersteuerelement als Titel enthält. Mit dem Ein- oder Ausstecken dieser Komponente wird gleichzeitig die Eigenschaft `TitleControl` des Inhaltes gesetzt oder zurückgesetzt. Der zweite Steckplatz *CRM.WelcomeElements* nimmt eine unbegrenzte Anzahl an Komponenten auf. Diese Komponenten enthalten die untergeordneten ContentControls, mit denen der Inhalt aufgebaut wird. In diesem Fall wird mit dem Ein- oder Ausstecken der Komponenten die Liste der untergeordneten Elemente (`Elements`) erweitert oder reduziert. Im letzten Steckplatz, dem *CRM.ContentRenderer*, kann genau eine Komponente eingesteckt werden. Diese Komponente ist für das Aussehen des Inhaltes einer Ansicht verantwortlich. Sie ist wiederum ähnlich der Komponente im Steckplatz *CRM.DashboardViewRenderer* einer Ansicht aufgebaut. Jeder Inhalt einer Ansicht verfügt über diesen Steckplatz und nimmt seine eigene Komponente vom Typ `IContentRenderer` auf.

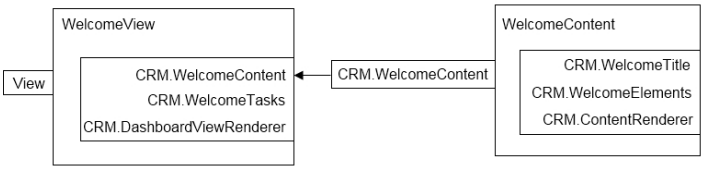

Abbildung 3.21: Die Komponente WelcomeView mit dem WelcomeContent

3.3.4 Task

Der rechte Bereich einer Ansicht weist einen schmalen Streifen mit Aufgaben auf. Eine Aufgabe einer Ansicht wird in Plux-CRM als *Task* dargestellt. Sie ist ähnlich dem Inhalt einer Ansicht aufgebaut. Auch bei einer Aufgabe gibt es unterschiedliche Bereiche. Der obere Bereich enthält den Titel. Dieser kann wiederum aus einem Text oder einem Benutzersteuerelement bestehen. Der untere Bereich wird aus untergeordneten Elementen, sogenannten *TaskControls*, zusammengestellt. Die Gliederung einer Aufgabe wird in Abbildung 3.22 anhand der Aufgabe auf der rechten Seite der „Willkommen“-Seite dargestellt. (1) zeigt den Titel, der aus einem Text besteht. (2) markiert das untergeordnete Element der Aufgabe.

Eine Aufgabe kommt in Plux-CRM in Form einer Komponente vor. Diese Komponente ist eine eigene Klasse, die von `AbstractTask` die wichtigsten vorimplementierten Eigen-

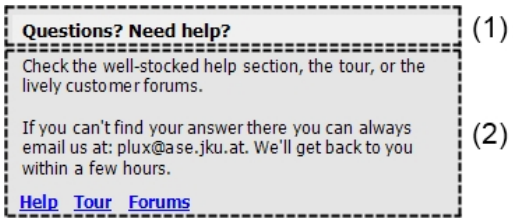

Abbildung 3.22: Gliederung eines Tasks

schaften erbt. `AbstractTask` verfügt über seine eigene Schnittstelle `ITask` (siehe Listing 3.3). Diese Schnittstelle definiert den Steckplatz *CRM.Task*, über den eine Aufgabe angesprochen werden kann. Wie jedes Element der Plux-CRM Benutzeroberfläche ist auch `ITask` eine Ableitung von der Wurzelschnittstelle `IDashboardElement`. `ITask` verfügt über zusätzliche Eigenschaften. Zu diesen zählen der Name der Aufgabe und somit auch der dazugehörigen Komponente (`Name`), der Titel in Form eines Textes (`Title`) und in Form eines Benutzersteuerelementes (`TitleControl`). Auch hier gilt wiederum: Ist ein Benutzersteuerelement für den Titel gesetzt, so verschwindet der Titeltext im Hintergrund, falls einer definiert ist. Die Eigenschaft `TaskControls` übernimmt eine ähnliche Funktion wie die Eigenschaft `Elements` beim Inhalt einer Ansicht. `TaskControls` bildet eine Auflistung mit den untergeordneten Elementen, aus denen der Inhalt einer Aufgabe zusammengesetzt wird.

Listing 3.3: Die Schnittstelle ITask

```
1 [SlotDefinition("CRM.Task")]
2 public interface ITask : IDashboardElement {
3   string Name { get; }
4   string Title { get; }
5   Control TitleControl { get; }
6   IEnumerable<IDashboardElement> TaskControls { get; }
7 }
```

Jedes untergeordnete Element einer Aufgabe ist vom Typ `ITaskControl` und enthält dasselbe Schema wie eine Aufgabe (Name, Titel als Text oder Benutzersteuerelement und eine Auflistung mit untergeordneten Elementen). Auch in diesem Fall ist eine beliebig tiefe Zusammensetzung von TaskControls möglich. TaskControls sollen wie ContentControls einen möglichst kleinen Teilbereich einer Aufgabe übernehmen, um im Bedarfsfall schnell und einfach ausgetauscht zu werden.

Beispiel eines Task

Abbildung 3.23 zeigt anhand der Komponente *WelcomeQuestionsTask*, wie eine Aufgabe in den passenden Steckplatz einer Ansicht eingesteckt wird. Die dazu in Plux-CRM angezeigte Aufgabe ist Abbildung 3.22 zu entnehmen. Da der Titel dieser Aufgabe nur

aus einem Text besteht, ist keine Komponente mit einem Benutzersteuerelement nötig. Aus diesem Grund wurde auch bei der Aufgabe auf einen eigenen Steckplatz für das Benutzersteuerelement verzichtet. Möchte man den Titel um ein eigenes Benutzersteuerelement erweitern, hat man die Möglichkeit, einen Steckplatz für den Titel hinzuzufügen. Der Steckplatz *CRM.WelcomeQuestions.TaskControl* verwaltet alle Komponenten mit den untergeordneten Elementen. In ihn kann eine beliebige Anzahl an Komponenten eingesteckt sein. Wie bereits bei jedem vorgestellten Element der Benutzeroberfläche besitzt auch eine Aufgabe einen speziellen Renderer, der das Aussehen einer Aufgabe verwaltet. Der Renderer kommt in Form einer Komponente vor und wird in den Steckplatz `CRM.TaskRenderer` eingesteckt. Jede Aufgabe in Plux-CRM besitzt diesen Steckplatz, in den jeweils ein eigene Renderer-Komponente vom Typ `ITaskRenderer` eingesteckt wird.

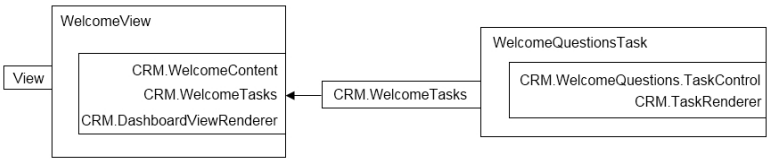

Abbildung 3.23: Die Komponente WelcomeView mit dem WelcomeQuestionsTask

3.3.5 Action

Eine *Action* hat in Plux-CRM eine Art Sonderstellung. Sie trägt nicht zum Aufbau einer Ansicht bei, sondern öffnet beim Aktivieren eine entsprechende Ansicht. Eine Action kommt in der Plux-CRM Benutzeroberfläche in Form eines Links vor, der sich in der rechten oberen Ecke des Hauptfensters befindet. Für jeden dieser Links gibt es in Plux-CRM eine eigene Action-Komponente. Eine Action ist eine Klasse, die die Schnittstelle `ICRMAction` implementiert.

Listing 3.4: Die Schnittstellen ICRMAction und IAction

```
1  [SlotDefinition("CRM.Action")]
2  [Param("MenuItem", typeof(string), "")]
3  [Param("Tag", typeof(string), "")]
4  [Param("OrderIndex", typeof(float), 0f)]
5  [Param("IsAdminAction", typeof(bool), false)]
6  public interface ICRMAction : IAction {
7    PlugInfo ActionPlug { get; }
8    string MenuItem { get; }
9    float OrderIndex { get; }
10   bool IsAdminAction { get; }
11 }
12
13 public interface IAction {
14   void Do(object sender, ActionEventArgs args);
15   bool IsEnabled(object sender, ActionEventArgs args);
16 }
```

Wie in Listing 3.4 zu erkennen, enthält `ICRMAction` neben der Steckplatz-Definition `CRM.Action` eine Reihe von Plux.NET-Parametern. Der Parameter `MenuItem` steht für den Namen, der als Link im Dashboard angezeigt wird. Der Parameter `Tag` kann mit den Werten „ONLINE“ und „OFFLINE“ belegt werden. Wird der erste Wert gesetzt, so kommt der Link nur beim Anmeldefenster vor. Wird der zweite Wert gesetzt, so kommt der Link bei allen anderen Ansichten vor. Es ist möglich, beide Werte gleichzeitig zu setzen. In diesem Fall erscheint der Link sowohl beim Anmeldefenster, als auch bei allen Ansichten von Plux-CRM. Der Parameter `OrderIndex` gibt an, in welcher Reihenfolge die Links in der Titelleiste vorkommen. Der Parameter `IsAdminAction` gibt an, ob man für diesen Link Administrator-Rechte benötigt. Ist dieser Parameter gesetzt und hat der angemeldete Benutzer keine Administrator-Rechte, so wird der Link in der Titelleiste nicht angezeigt. Für die Parameter `MenuItem`, `OrderIndex` und `IsAdminAction` besitzt `ICRMAction` gleichnamige Eigenschaften, die mit den Werten der Parameter gesetzt werden. Zusätzlich verfügt `ICRMAction` über die Eigenschaft `ActionPlug`, die den Stecker der Action-Komponente enthält. `ICRMAction` ist von der Plux.NET-Schnittstelle `IAction` abgeleitet. `IAction` verfügt über zwei Methoden. Die Methode `Do` wird aufgerufen, wenn der Benutzer auf den dazugehörigen Link klickt. Sie dient für die weitere Vorgehensweise beim Aktivieren eines Links. In den Klassen, die diese Methode implementieren, wird dort beispielsweise festgelegt, welche Ansicht zu öffnen ist. Zu einer Ansicht, die über einen Link geöffnet wird, wird kein eigener Karteireiter in der Titelleiste des Dashboardes erzeugt. Die zweite Methode der Schnittstelle (`IsEnabled`) überprüft, ob beim Aktivieren eines Links, dieser mit einer Funktion belegt ist.

Beispiel einer Action

Abbildung 3.24 zeigt anhand der Komponente *MyInfoAction*, wie in Plux-CRM eine Action und eine Ansicht zusammenspielen. Die *MyInfoAction*-Komponente ist ein Beispiel für einen Contributor, der einen Stecker für *CRM.Action* hat, aber keinen Steckplatz besitzt.

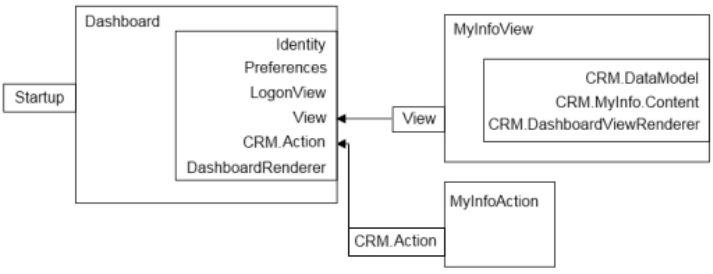

Abbildung 3.24: Die Komponente Dashboard mit der MyInfoAction und MyInfoView

Listing 3.5: Die Klasse MyInfoAction mit Code-Ausschnitten aus Plux.NET

```
1  public class MyInfoAction : AbstractAction {
2    public override void Do(object sender, ActionEventArgs args) {
3      // finds the slot "View"
4      SlotInfo viewSlot = PlugInfoEx.FindSlot(args.Plug, "View");
5      // finds the registered plugtypeinfo "MyInfoView" via the param "Tag"
6      // and creates an extension
7      ExtensionInfo groupsViewExtension = SlotInfoEx.FindRegisteredPlug(viewSlot,
8          "Tag", "MyInfoView").ExtensionTypeInfo.GetSharedExtension(true);
9      // plugs the extension into the slot
10     groupsViewExtension.PlugInfos[viewSlot.Name].Plug(viewSlot);
11   }
12 }
```

Mit dem Einstecken dieser Komponente wird gleichzeitig die dazupassende Ansicht *MyInfoView* erzeugt und in den Steckplatz *View* des Dashboardes gesteckt. Listing 3.5 zeigt die `Do`-Methode der `MyInfoAction`, die bei diesem Vorgang aufgerufen wird. Zuerst wird mittels der statischen Plux.NET-Methode `PlugInfoEx.FindSlot` der Steckplatz *View* des Dashboardes ermittelt. Danach wird über die statische Plux.NET-Methode `SlotInfoEx.FindRegisteredPlug` die registrierte Ansicht-Komponente *MyInfoView* über den Parameter *Tag* gesucht und erzeugt. Im nächsten Schritt wird die *MyInfoView*-Komponente in den Steckplatz *View* gesteckt. In der Anwendung erscheint eine Ansicht zum Ändern der persönlichen Daten eines Benutzers. Zu den persönlichen Daten zählen der Benutzername, das Passwort, die Email-Adresse und das Profilbild (siehe Abbildung 3.25).

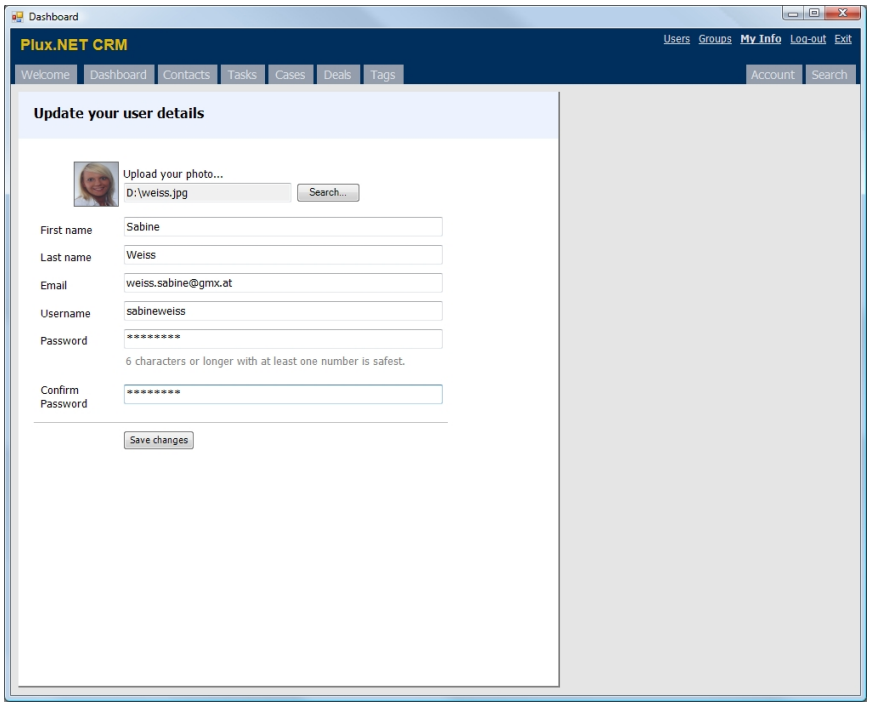

Abbildung 3.25: MyInfoAction und MyInfoView in der Plux-CRM Benutzeroberfläche

In Plux-CRM wird beim Aktivieren eines Links meist eine entsprechende Ansicht geöffnet, so wie im vorherigen Beispiel beschrieben. Die Links „Logout“ und „Exit“ bilden Ausnahmen. Aktiviert ein Benutzer den Link „Logout“, so wird keine gewöhnliche Ansicht geöffnet. Mit dieser Aktion wird die zugehörige *LogoutAction*-Komponente dem *CRM.Action*-Steckplatz hinzugefügt. Gleichzeitig meldet diese Komponente den Benutzer von Plux-CRM ab, in dem sie die *Identity*-Komponente aus dem gleichnamigen Steckplatz des Dashboardes entfernt. Wie bereits in Kapitel 3.3.1 erklärt, wird mit dem Ausstecken der *Identity*-Komponente die eingesteckte Ansicht-Komponente aus dem Steckplatz *View* des Dashboardes entfernt, der Steckplatz *LogonView* entsprechend befüllt und in der Anwendung erscheint das Anmeldefenster. Aktiviert ein Benutzer den Link „Exit“, wird ebenfalls die zugehörige *ExitAction*-Komponente in den *CRM.Action*-Steckplatz eingesteckt. Gleichzeitig schließt diese Komponente Plux-CRM (*Shutdown*). Alle eingesteckten Plux.NET-Komponenten werden somit aus ihren Steckplätzen entfernt.

3.4 Struktur der Plux-CRM Plugins

Die Basis aller in Plux-CRM verwendeten Plugins bildet das Plux.NET-Framework. Ohne dieses Framework ist Plux-CRM nicht funktionsfähig. In Plux-CRM sind die meisten Plugins für sich abgeschlossen und nicht voneinander abhängig. Da alle Plugins auf den selben Elementen basieren und sie auch teilweise ähnliche Funktionen übernehmen, lassen sich gewissen Abhängigkeiten der einzelnen Plugins nicht vermeiden. Abbildung 3.26 zeigt die Abhängigkeiten der einzelnen Plugins in Plux-CRM.

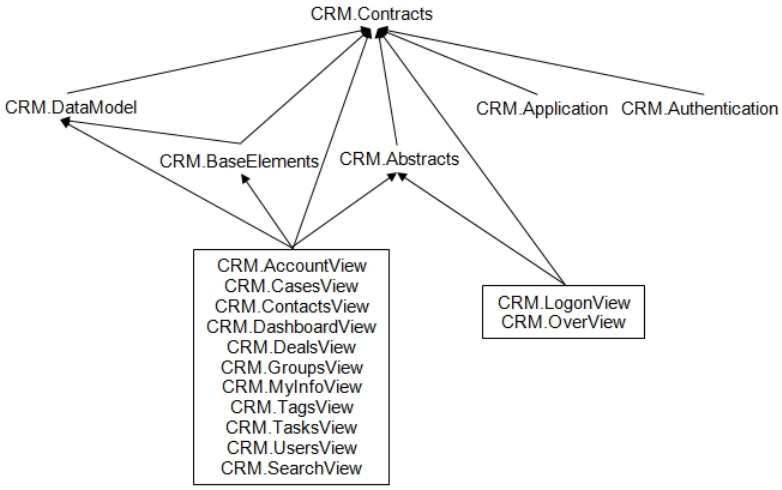

Abbildung 3.26: Abhängigkeiten zwischen Plux-CRM Plugins

Die Plugins ohne Umrandung sind für die Architektur und das Design im Hintergrund von Plux-CRM zuständig. Sie bilden den Grundstein von Plux-CRM. Die Plugins mit

Umrandung benötigen die Plugins ohne Umrandung. Alle Plugins mit Umrandung sind direkt in Plux-CRM zu sehen und bilden jeweils eine eigene Funktion. Fehlt eines dieser Plugins mit Umrandung, so ist zwar der Funktionsbereich des Plugins in der Anwendung nicht verfügbar, aber alle anderen Plugins werden davon nicht beeinflusst. Fehlt eine der Bibliothekskomponenten, können die Plugins nicht geladen werden.

In Tabelle 3.1 sind die Funktionen der Plux-CRM Plugins zusammengefasst.

Plugin	Beschreibung
CRM.Abstracts	Abstrakte Klassen der Basis-Elemente der Benutzeroberfläche
CRM.AccountView	Ansicht zum Konfigurieren der Ansichten
CRM.Application	Architektur der Benutzeroberfläche
CRM.Authentication	Authentifizierung der Benutzer
CRM.BaseElements	Basis-Benutzersteuerelemente und -Komponenten, die in mehreren Plugins verwendet werden
CRM.CasesView	Ansicht zum Verwalten der Projekte
CRM.ContactsView	Ansicht zum Verwalten der Kunden
CRM.Contracts	Steckplatz-Definitionen und Schnittstellen der Benutzersteuerelemente
CRM.DashboardView	Ansicht zum Verwalten des „Schwarzen Bretts“
CRM.DataModel	Datenmodell und Funktionen zum Importieren/Exportieren
CRM.DealsView	Ansicht zum Verwalten der Geschäfte
CRM.GroupsView	Ansicht zum Verwalten der Benutzergruppen
CRM.LogonView	Ansicht zum Anmelden eines Benutzers
CRM.MyInfoView	Ansicht zum Ändern der Benutzerdaten
CRM.SearchView	Ansicht zum Suchen von Notizen und Kommentaren
CRM.TagView	Ansicht zum Verwalten der Stichwörter
CRM.TasksView	Ansicht zum Verwalten der Benutzer-Aufgaben
CRM.UsersView	Ansicht zum Verwalten der Benutzer
CRM.WelcomeView	Begrüßungs-Ansicht

Tabelle 3.1: Überblick der Plux-CRM Plugins

Die Wurzel aller in Plux-CRM verwendeten Plugins bildet das Plugin `CRM.Contracts`. Es enthält alle wichtigen Schnittstellen der einzelnen Benutzersteuerelemente, der Komponenten zusammen mit den Definitionen ihrer Steckplätze, des Datenmodells und der Authentifizierung der Benutzer. Von diesem Plugin sind alle weiteren Plugins abhängig.

Das Plugin `CRM.Authentication` besitzt nur eine Abhängigkeit zu `CRM.Contracts`. Sein Aufgabengebiet liegt im Authentifizieren eines Benutzers zum Startzeitpunkt von Plux-CRM.

Das Plugin `CRM.Application` hat ebenfalls nur eine Referenz zu `CRM.Contracts`. Es umfasst alle Basis-Benutzersteuerelemente und deren Aufbau und Mechanismen zum korrekten Darstellen eines Elementes. Zu diesen Basis-Benutzersteuerelementen zählen alle in Kapitel 3.3 beschriebenen Elemente. Das Plugin hat die Aufgabe, diese einzelnen Benutzersteuerelemente korrekt zu einer Anwendung zusammenzubauen.

`CRM.DataModel` ist das dritte Plugin mit nur einer Abhängigkeit. Darin befindet sich das Datenmodell zusammen mit der Funktion zum Importieren und Exportieren von Kontaktdaten im CSV- oder VCard-Format.

Als letztes Plugin mit nur einer Abhängigkeit zu `CRM.Contracts` ist die Bibliotheks-Komponente `CRM.Abstracts` zu erwähnen. Es enthält alle abstrakten Klassen mit den wichtigsten vorprogrammierten Methoden, die nötig sind, um eine Ansicht aufzubauen.

Vom Plugin `CRM.BaseElements` bestehen Abhängigkeien zu `CRM.Contracts` und `CRM.DataModel`. Dieses Plugin enthält Benutzersteuerelemente und Komponenten mit gewissen Funktionen, die in mehreren anderen Plugins vorkommen. Aufgrund der Wiederverwendbarkeit dieser Benutzersteuerelemente und Komponenten sind sie in einem eigenen Plugin ausgelagert. Alle Plugins, die diese Elemente verwenden, besitzen eine Referenz auf das Plugin `CRM.BaseElements`.

`CRM.LogonView` und `CRM.OverView` sind zwei Plugins, die nur Abhängigkeiten zu `CRM.Abstracts` und `CRM.Contracts` besitzen. Das erste Plugin ist für das Anmelden eines Benutzers zuständig. Das zweite Plugin startet nach dem Anmelden des Benutzers die „Wilkommen“-Seite.

Die Plugins `CRM.AccountView`, `CRM.CasesView`, `CRM.ContactsView`, `CRM.DashboardView`, `CRM.DealsView`, `CRM.GroupsView`, `CRM.MyInfoView`, `CRM.TagsView`, `CRM.TasksView`, `CRM.UsersView` und `CRM.SearchView` decken jeweils den Funktionsbereich einer eigenen Ansicht ab. Die Beschreibung der einzelnen Ansichten findet in Kapitel 1.2 statt. Jedes dieser Plugins hat Referenzen auf die Plugins `CRM.Abstracts`, `CRM.BaseElements`, `CRM.DataModel` und `CRM.Contracts`.

4 Ausgewählte Benutzersteuerelemente von Plux-CRM

Plux.NET enthält spezielle Benutzersteuerelemente, die genau auf die Bedürfnisse einer Plux.NET-Anwendung zugeschnitten sind. Beispielsweise wurden Schaltflächen, die von den Windows Forms 2.0 Schaltflächen abgeleitet sind, so modifiziert, dass sie beim Aktivieren durch den Benutzer eine speziell definierte Komponente aus einem Steckplatz entfernen (*UnplugButton*) oder einem Steckplatz hinzufügen (*PlugButton*). Diese abgewandelten Mechanismen für Benutzersteuerelemente existieren in Plux.NET auch für Links, Auswahlfelder und Karteireiter.

In diesem Kapitel wird auf zwei spezielle Benutzersteuerelemente und deren Abhängigkeiten zu anderen Benutzersteuerlemente eingegangen, die vor der Implementierung von Plux-CRM bereits Teil von Plux.NET waren, für Plux-CRM weiterentwickelt und in die Plux.NET-Bibliothek integriert wurden.

4.1 ReplaceButton

Die Klasse `ReplaceButton` implementiert eine Schaltfläche, die eine Komponente aus einem Steckplatz entfernt und anschließend eine neue Komponente in denselben Steckplatz wieder einsteckt. `ReplaceButton` ist eine Spezialisierung der Windows Form Klasse *Button*. Sie enthält neben den von Button geerbten Eigenschaften und Methoden drei zusätzliche Methoden. Zwei Methoden davon besitzen denselben Namen `Attach`, führen diesselbe Funktion aus und unterscheiden sich nur durch ihre übergebenen Parameter. Beide `Attach`-Methoden erzeugen ein neues Benutzersteuerelement `PlugControl`. Dieses speziell für Plux.NET entwickelte Benutzersteuerelement führt beim Aktivieren durch den Benutzer eine Aktion aus, die ihm vorher übergeben wurde. Beim `ReplaceButton` wird dem erzeugten `PlugControl` eine spezielle Aktion in Form einer `ReplaceAction` übergeben. Die Aufgabe der `ReplaceAction` ist, dass sie bei ihrem Aktivieren eine Komponente aus einem Steckplatz entfernt (`UnplugAction`) und anschließend eine neue Komponente in diesen Steckplatz einsteckt (`PlugAction`). Beim Erzeugen eines Objekts dieser Klasse wird diesem Objekt der Steckplatz (`slot`), der Name eines Parameters (`param`), über den die alte und die neue Komponente identifiziert werden, und die Werte dieses Pa-

rameters von der alten und der neuen Komponente (`valueUnplug` und `valuePlug`). Dabei werden sowohl eine `UnplugAction`, die das Ausstecken der alten Komponente ausführt, und eine `PlugAction`, die das Einstecken der neuen Komponente veranlasst, erzeugt.

Beim Aktivieren des Benutzersteuerelementes `ReplaceButton` wird die Methode `OnClick` ausgeführt, die sowohl die gleichnamige Methode des Benutzersteuerelementes `PlugControl` als auch die überschriebene Methode der Basisklasse `Button` aufruft. Wird die Methode `Do` der Klasse `PlugControl` ausgeführt, so wird die vorher übergebene Aktion aktiviert. Im Fall des `ReplaceButton` wird die `ReplaceAction` aktiviert. Dabei wird die Methode `Do` ausgeführt, die die gleichnamige Methode der zuvor erzeugten Aktionen `UnplugAction` und `PlugAction` aufruft und das Aus- und Einstecken der Komponenten aus dem Steckplatz wird ausgeführt.

Die Zeilen 48-54 von Listing 4.1 zeigen die Methode `Do` der Klasse `UnplugAction`. Zu sehen ist, dass alle Stecker in Form von Plugs des zuvor angegebenen Steckplatzes durchlaufen werden und anhand von *Qualify* überprüft werden, ob der angegebene Parameter und sein Wert mit den Werten einer der Stecker übereinstimmt. Wird der entsprechende Stecker gefunden, so wird er aus dem Steckplatz entfernt.

Listing 4.1: Die Klasse ReplaceButton mit ihren verwendeten Elementen

```
1  public class ReplaceButton : Button {
2    private PlugControl ctl;
3    public void Attach(SlotInfo slot, QualifyDelegate unplug,
4      QualifyDelegate plug) {
5      ctl = new PlugControl(this, new ReplaceAction(slot, unplug, plug));
6    }
7    public void Attach(SlotInfo slot, string param, object valueUnplug,
8      object valuePlug) {
9      ctl = new PlugControl(this, new ReplaceAction(slot, param,
10       valueUnplug, valuePlug));
11   }
12   protected override void OnClick(EventArgs e) {
13     ctl.OnClick(e);
14     base.OnClick(e);
15   }
16 }
17
18 public class PlugControl : IDisposable {
19   protected Action Action = null;
20   public PlugControl(Control control, Action action) {
21     Action = action;
22     ...
23   }
24   public void OnClick(EventArgs e) {
25     Action.Do(null, null);
26   }
27   ...
28 }
29
30
31
```

```
32 public class ReplaceAction : Action {
33   private readonly PlugAction plugAction;
34   private readonly UnplugAction unplugAction;
35   public ReplaceAction(SlotInfo slot, string param, object valueUnplug,
36     object valuePlug) : base(slot, null) {
37     unplugAction = new UnplugAction(slot, param, valueUnplug);
38     plugAction = new PlugAction(slot, param, valuePlug);
39   }
40   public override void Do(object s, EventArgs args) {
41     unplugAction.Do(s, args);
42     plugAction.Do(s, args);
43   }
44   ...
45 }
46
47 public class UnplugAction : Action {
48   public UnplugAction(SlotInfo slot, string param, object value)
49     : base(slot, param, value) {}
50   public override void Do(object s, EventArgs args) {
51     for (int index = Slot.PluggedPlugInfos.Count - 1; index >= 0; --index) {
52       PlugInfo plug = Slot.PluggedPlugInfos[index];
53       if (Qualify != null && !Qualify(plug.PlugTypeInfo)) continue;
54       plug.ExtensionInfo.UnplugPlugs(Slot.ExtensionInfo);
55     }
56   }
57   ...
58 }
59
60 public class PlugAction : Action {
61   public PlugTypeInfo PlugType { get; private set; }
62   public PlugAction(SlotInfo slot, string param, object value)
63     : base(slot, param, value) {}
64   ...
65
66   public override void Do(object s, EventArgs args) {
67     var extTypes = new List<ExtensionTypeInfo>();
68     if (PlugType != null) {
69       extTypes.Add(PlugType.ExtensionTypeInfo);
70     } else {
71       foreach (PlugTypeInfo plugType in Slot.RegisteredPlugTypeInfos) {
72         if (Qualify != null && !Qualify(plugType)) continue;
73         extTypes.Add(plugType.ExtensionTypeInfo);
74         if (!Slot.SlotTypeInfo.Multiple) break;
75       }
76     }
77     foreach (var extType in extTypes) {
78       ExtensionInfo ext;
79       if (Slot.Unique) {
80         ext = extType.CreateUniqueExtension();
81         ext.PlugPlugs(Slot.ExtensionInfo);
82       } else {
83         ext = extType.GetSharedExtension(true);
84         ext.PlugPlugs();
85       }
86     }
87   }
88 }
```

4.1.1 Verwendung des ReplaceButton in Plux-CRM

Der ReplaceButton wird im Plugin *CRM.TasksView* verwendet. Jede Ansicht, bei der im rechten Bereich die geplanten Aufgaben eines Benutzers oder eines Kunden angezeigt werden, enthält eine Schaltfläche zum Hinzufügen einer neuen Aufgabe. Abbildung 4.1 zeigt diese Schaltfläche in der Ansicht der Kunden. Sie befindet sich zwischen dem Suchfenster und der Schaltfläche „Add a new person“ zum Hinzufügen einer neuen Kontaktperson. Die Schaltfläche „Add a task“ ist in Plux-CRM ein ReplaceButton. Das Benutzersteuerelement, auf dem sich diese Schaltfläche befindet, kommt in Form einer eigenen *TaskControl*-Komponente mit dem Namen *AddTaskButtonTaskControl* vor.

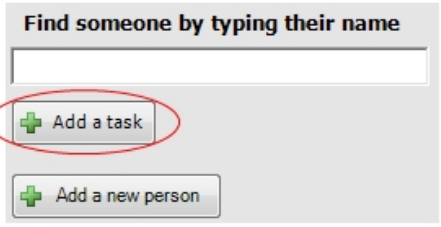

Abbildung 4.1: Schaltfläche zum Hinzufügen einer Aufgabe

Wie bereits in Kapitel 3.3.4 erwähnt, ist eine TaskControl-Komponente immer ein Contributor einer Task-Komponente. So ist in Plux-CRM die Komponente *AddTaskButtonTaskControl* Teil der Komponente *AddTaskButtonTask*. Beim Anzeigen einer Ansicht, die die Aufgaben eines Benutzers oder Kunden enthält, werden diese beiden Komponenten automatisch zusammengesteckt und die Task-Komponente dem Task-Steckplatz der aktuellen View-Komponente hinzugefügt. Abbildung 4.2 zeigt anhand der Ansicht des „Schwarzen Bretts“ (*DashboardView*), wie die erwähnten Komponenten zusammenhängen.

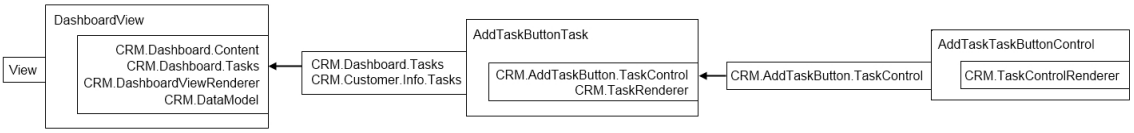

Abbildung 4.2: DashboardView-Komponente mit eingesteckten AddTaskButtonTask- und AddTaskButtonTaskControl-Komponenten

Beim Einstecken der Komponente *AddTaskButtonTaskControl* wird beim Erzeugen der Komponente die Schaltfläche zum Hinzufügen einer neuen Aufgabe in Form eines neuen Objektes der Klasse `ReplaceButton` erzeugt. Beim Anzeigen dieser Schaltfläche wird vorher dem `ReplaceButton`-Objekt der Steckplatz übergeben, in dem sich die *AddTaskButtonTask*-Komponente befindet. Im Fall von Abbildung 4.4 ist dies der Steckplatz *CRM.Dashboard.Tasks*. Neben diesem Steckplatz wird dem `ReplaceButton` der Name

eines .NET-Parameters und der Wert dieses .NET-Parameters für die alte und für die neue Komponente übergeben. Zeile 11 in Listing 4.2 zeigt, dass in diesem Fall der .NET-Parameter *Tag* übergeben wird. *Tag* ist bei jeder Komponente definiert und enthält die Bezeichnung einer Komponente. In Listing 4.2 ist der Wert für die alte Komponente *AddTaskButtonTask* und der für die neue Komponente *AddTaskTask*.

Listing 4.2: Die Klasse AddTaskButtonTaskControl

```
1  public class AddTaskButtonTaskControl : AbstractTaskControl {
2    private readonly AddTaskButtonTaskUC taskUC;
3    public AddTaskButtonTaskControl() {
4      taskUC = new AddTaskButtonTaskUC();
5    }
6    public override Control DashboardControl {
7      get {
8        ExtensionInfo taskExtension = TaskControlPlug.PluggedInSlots[0].ExtensionInfo;
9        foreach (PlugInfo plug in taskExtension.PlugInfos) {
10         if (plug.PluggedInSlots.Count > 0) {
11           taskUC.AddTaskB.Attach(PlugInfoEx.FindSlot(TaskControlPlug,
12             plug.PluggedInSlots[0].Name), "Tag", "AddTaskButtonTask", "AddTaskTask");
13           break;
14         }
15       }
16       return taskUC;
17     }
18   }
19   ...
20 }
```

Aktiviert nun der Benutzer die Schaltfläche „Add a task“ zum Hinzufügen einer neuen Aufgabe, so erscheint in Plux-CRM ein neues Formular. Anhand dieses Formulares können die Werte für eine neue Aufgabe angegeben und gespeichert werden.

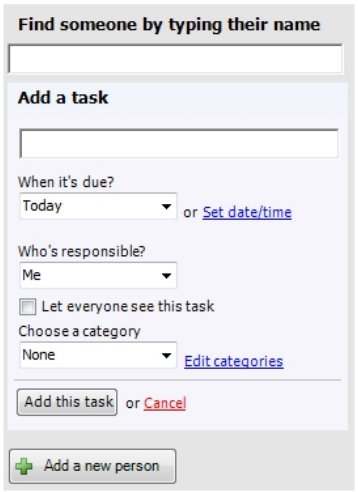

Abbildung 4.3: Formular zum Hinzufügen einer neuen Aufgabe

Abbildung 4.3 zeigt das Formular an der Stelle in der Kundenansicht, wo sich zuvor die Schaltfläche „Add a task“ befand, zwischen dem Suchfenster und der Schaltfläche zum Hinzufügen einer neuen Kontaktperson.

Beim Aktivieren der Schaltfläche „Add a task“ entfernt der zuvor erzeugte `ReplaceButton` zuerst die Komponente *AddTaskButtonTask* aus dem Task-Steckplatz der View-Komponente. Anschließend erzeugt er anhand des angegebenen Wertes *AddTaskTask* eine neue Komponente und fügt diese dem Task-Steckplatz hinzu. Die Komponente *AddTaskTask* ist ähnlich der Komponente *AddTaskButtonTask*. Auch sie verfügt über eine zugehörige TaskControl-Komponente. Diese trägt den Namen *AddTaskTaskControl* und enthält das in Abbildung 4.3 gezeigte Formular. Wie die Struktur der zu diesem Zeitpunkt eingesteckten Komponenten aussieht, zeigt Abbildung 4.4.

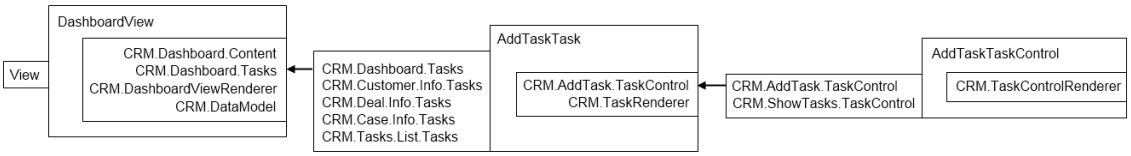

Abbildung 4.4: DashboardView-Komponente mit eingesteckten AddTaskTask- und AddTaskTaskControl-Komponenten

Das Formular der *AddTaskTaskControl*-Komponente enthält die Schaltfläche „Add this task“ zum Speichern der neuen Aufgabe. Diese Schaltfläche ist ebenfalls ein `ReplaceButton` und führt diesselbe Aktion wie die Schaltfläche „Add a task“ in entgegengesetzter Richtung aus. Das heißt, sie entfernt die Komponente *AddTaskTask* aus ihrem Steckplatz und fügt die neue *AddTaskButtonTask*-Komponente dem Steckplatz wieder hinzu. Mit dieser Aktion erscheint in Plux-CRM die in Abbildung 4.1 dargestellte Schaltfläche „Add a task“. Neben dem Umstecken der Komponenten werden die Werte im Formular der *AddTaskTaskControl*-Komponente als eine neue Aufgabe gespeichert.

4.2 PlugComboBox

Das zweite Benutzersteuerelement, das für Plux-CRM erweitert wurde, ist die in Plux.NET enthaltene `PlugComboBox`. Die `PlugComboBox` ist ein Auswahlfenster. Ihr muss ein bestimmter Steckplatz zugeordnet sein. Sie bietet alle in diesem Steckplatz registrierten Contributor-Komponenten zur Auswahl an, steckt aber nur die vom Benutzer selektierte Contributor-Komponente in den Steckplatz ein. Abbildung 4.5a zeigt eine `PlugComboBox`, bei der Kunden nach bestimmten Kriterien gefiltert werden können. Nur der Eintrag „All companies“ ist ausgewählt. Abbildung 4.5b zeigt, dass dieser Eintrag die Komponente *ContactFilterAllCompanies* darstellt und im Steckplatz *CRM.Contacts.Filter* ein-

gesteckt ist. Anhand der strichlierten Umrandung ist zu erkennen, dass die Komponenten *ContactFilterAllContacts* und *ContactFilterAllContactsAndCompanies* im Steckplatz *CRM.Contacts.Filter* registriert, aber nicht eingesteckt sind.

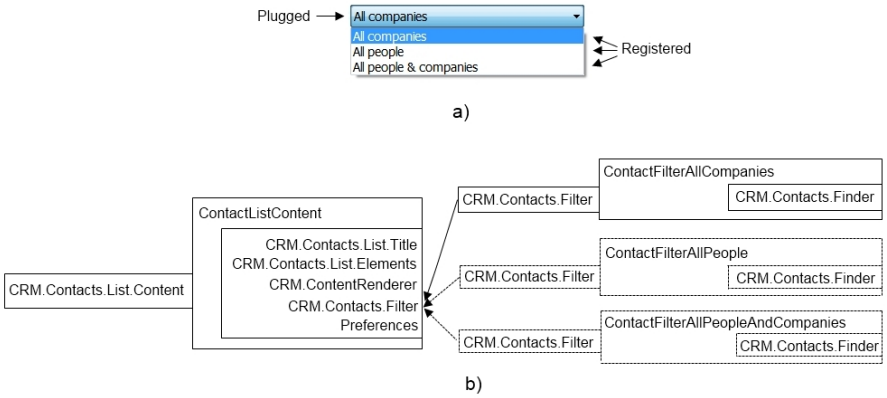

Abbildung 4.5: der Komponenten einer PlugComboBox

Die `PlugComboBox` ist in Plux.NET eine Klasse, die sowohl eine Ableitung der Windows Form *ComboBox* bildet, als auch die Schnittstelle `IAttachable` von Plux.NET implementiert (siehe Listing 4.3). `IAttachable` verfügt über die zwei Methoden `Attach` und `Detach`, die von der `PlugComboBox` implementiert werden. Die Methode `Attach` kommt beim Anzeigen des Auswahlfensters in der Anwendung zum Einsatz. Dieser Methode wird ein Steckplatz übergeben. Bei diesem Steckplatz werden Mechanismen angemeldet, die beim Auslösen bestimmter Ereignisse des Steckplatzes ausgelöst werden. Zu diesen Ereignissen zählen das Registrieren und Abmelden von Komponenten und das Umstecken einer Komponente. Zusätzlich werden alle Werte des Auswahlfensters entfernt. Die Methode `Detach` wird aufgerufen, wenn das gesamte Auswahlfenster aus der Anwendung entfernt wird. In dieser Methode werden ebenfalls alle Werte des Auswahlfensters entfernt und zusätzlich alle vorher angemeldeten Mechanismen wieder abgemeldet. Jeder dieser angemeldeten Mechanismen kommt in Form einer Methode vor. Die Methode `Registered` wird aufgerufen, wenn der übergebene Steckplatz eine Komponente registriert. Diese Komponente besitzt einen speziellen Parameter. Die Methode erkennt diesen Parameter und fügt mittels der Methode `AddItem` den dazugehörigen Wert der Liste des Auswahlfensters hinzu. Das Gegenteil passiert in der Methode `Detach`. Diese wird aufgerufen, wenn sich eine Komponente beim übergebenen Steckplatz abmeldet. In diesem Schritt wird gleichzeitig der Parameter-Wert der Komponente aus der Liste des Auswahlfensters entfernt. Die überschriebene Methode `OnSelectedIndexChanged` stammt ursprünglich aus der Basisklasse `ComboBox`. Sie kommt zum Einsatz, wenn der Benutzer einen Wert im Auswahlfenster wählt. Im gleichen Schritt wird die Komponente des zuvor ausgewählten Wertes aus dem übergebenen Steckplatz entfernt. Anschließend wird für den neu ausgewählten Wert eine Komponente erzeugt und dem Steckplatz hinzugefügt.

Listing 4.3: Die Klasse PlugComboBox und die Schnittstelle IAttachable

```
1  public class PlugComboBox : ComboBox, IAttachable {
2    public SlotInfo Slot { get; private set; }
3    public void Attach(SlotInfo slot) {
4      Slot = slot;
5      Slot.Registered += Registered;
6      Slot.Deregistered += Deregistered;
7      Items.Clear();
8    }
9    public void Detach() {
10     Items.Clear();
11     Slot.Registered -= Registered;
12     Slot.Deregistered -= Deregistered;
13     Slot = null;
14   }
15   public void Registered(object sender, RegisterEventArgs args) {
16     AddItem(args.PlugTypeInfo);
17   }
18   private void AddItem(PlugTypeInfo plugType) {
19     //adds the plugtype in the correct order to the items of the combobox
20     ...
21     Items.Add(plugTypeInfo, plugTypeInfo.ExtensionTypeInfo);
22     ...
23   }
24   public void Deregistered(object sender, RegisterEventArgs args) {
25     Items.Remove(args.PlugTypeInfo);
26   }
27   protected override void OnSelectedIndexChanged(EventArgs e) {
28     base.OnSelectedIndexChanged(e);
29     if (SelectedItem.Key == null) return;
30     if (Slot.PluggedPlugInfos.Count > 0) Slot.PluggedPlugInfos[0].Unplug(Slot);
31     PlugInfo plug = ExtensionInfoEx.CreateExtension(PlugTypeInfo)SelectedItem.Key,
32       Slot);
33     plug.Plug(Slot);
34     Slot.SelectedPlug = plug;
35   }
36   ...
37 }
38
39 public interface IAttachable {
40   void Attach(SlotInfo slot);
41   void Detach();
42 }
```

4.2.1 Verwendung der PlugComboBox in Plux-CRM

In Plux-CRM kommt die `PlugComboBox` in der Ansicht der Kunden zum Einsatz. In dieser Ansicht ist ein Filter in Form eines Auswahlfensters zu sehen. Bei diesem Auswahlfenster kann der Benutzer wählen, welche Art von Kunden angezeigt werden. Der Benutzer kann dabei unterscheiden, ob beispielsweise alle Kunden, nur Kontaktpersonen oder nur Firmen in der Liste erscheinen. Je nach vorhandenen Filtern kann der Benutzer zusätzlich wählen, ob er nur die Kunden sehen möchte, die er in den letzten Tagen betrachtet oder

hinzugefügt hat, oder ob ihn Kontaktpersonen ohne Notizen oder mit Notizen der letzten 30 Tage interessieren. Abbildung 4.6a zeigt das Auswahlfenster mit den angeführten Auswahlmöglichkeiten. Abbildung 4.6b zeigt, dass für den ausgewählten Eintrag „All people & companies“ die Komponente *ContactFilterAllPeopleAndCompanies* im Steckplatz *CRM.Contacts.Filter* eingesteckt ist. Alle weiteren Komponenten, die die restlichen Einträge im Auswahlfeld darstellen, sind im Steckplatz registriert.

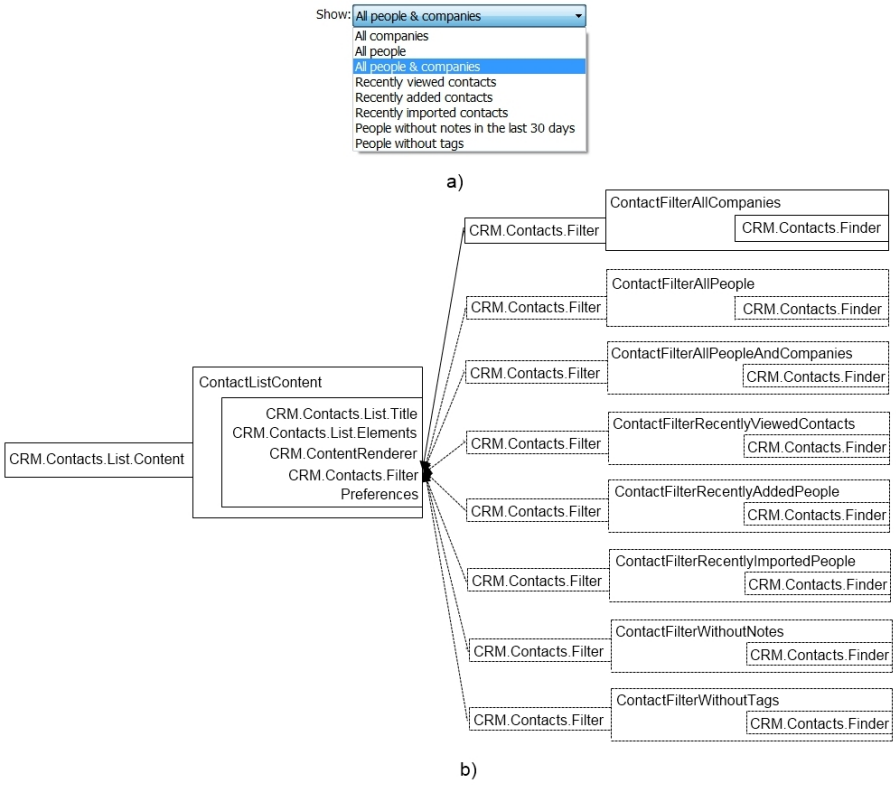

Abbildung 4.6: a) Auswahlfenster zum Filtern der angezeigten Kunden, b) eingesteckten und registrierten Filter-Komponenten

In Kapitel 5.2.2 wird anhand der Erweiterbarkeit dieser Filterfunktion genauer auf die Verwendung der `PlugComboBox` in Plux-CRM eingegangen.

5 Flexibilität von Plux-CRM

Wie bereits in Kapitel 3.1 erwähnt, lagen beim Entwurf von Plux-CRM die Schwerpunkte auf der individuell anpassbaren Benutzeroberfläche und auf der Zerlegung in fein-granulare Plugins. Diese speziellen Funktionalitäten unterscheiden Plux-CRM von monolithischen Programmen, bei denen weder der Benutzer die Möglichkeit hat, seine Benutzeroberfläche im gewissen Maß selbst zu gestalten, noch zusätzliche Funktionen zu einem späteren Zeitpunkt hinzugefügt werden können.

Im folgenden Abschnitt werden diese zwei besonderen Merkmale von Plux-CRM anhand von ausgewählten Beispielszenarien genauer erläutert.

5.1 Individuelle Konfiguration

Ein besonderer Aspekt bei Plux-CRM ist die individuelle Konfiguration der Benutzeroberfläche. Jeder Benutzer hat unterschiedliche Anforderungen an eine Software und bevorzugt somit verschiedene Funktionen. Bei Plux-CRM kann sich jeder Benutzer im Rahmen seiner Berechtigungen einen Großteil seiner Benutzeroberfläche individuell gestalten. Dabei wird zwischen Basisfunktionen und optionalen Funktionen unterschieden. Die Basisfunktionen sind bei jedem Benutzer enthalten und nicht frei zu gestalten. Die optionalen Funktionen kann sich jeder Benutzer beliebig konfigurieren.

5.1.1 Basisfunktionen

Aufgrund der Plugin-Struktur verfügt Plux-CRM über nur wenige Basisfunktionen. Durch Hinzufügen oder Entfernen optionaler Funktionen kann jeder Benutzer selbst entscheiden, welche Zusatzfunktionen er in Plux-CRM möchte. Zu den Basisfunktionen von Plux-CRM zählen die „Willkommen“-Ansicht (*Welcome*), das „Schwarze Brett“ (*Dashboard*), die Ansicht der Kunden und Kontaktpersonen (*Contacts*), die Ansicht zum Konfigurieren der optionalen Funktionen (*Account*), das Einstellen der Benutzerdaten (*My info*), das Abmelden des Benutzers (*Sign out*) und das Beenden der Anwendung (*Exit*).

Diese Basisfunktionen sind bei jedem Benutzer gleich und können nicht individuell gestaltet werden. Allerdings hat das Hinzufügen oder Entfernen der optionalen Funktionen Einfluss auf Teilbereiche der Basisfunktionen.

5.1.2 Optionale Funktionen

Neben den Basisfunktionen verfügt Plux-CRM über optionale Funktionen, die sich jeder Benutzer beliebig konfigurieren kann. Zu diesen optionalen Funktionen zählen die Ansicht der geplanten Aufgaben (*Tasks*), das Auflisten sowohl der einzelnen Geschäfte (*Deals*) als auch der Projekte (*Cases*) mit den verschiedenen Kunden, das Verwalten der Stichwörter zu den Kontaktpersonen, Kunden, Projekten und Geschäften (*Tags*) und die Suchfunktion (*Search*). Besitzt ein Benutzer Administrator-Rechte, so hat er zusätzlich die Möglichkeit, die Benutzer des Systems zu verwalten (*Users*) und sie in Gruppen einzuteilen (*Groups*).

Unter dem Karteireiter *Account* sind die Ansichten aufgeschlüsselt, zu die der Benutzer berechtigt ist. Je nach Belieben kann der Benutzer mittels „On“ oder „Off“ die Funktionen *Tasks*, *Cases*, *Deals*, *Tags*, *Search*, *Users* und *Groups* hinzufügen oder entfernen (siehe Abbildung 5.1). Diese Funktionen beziehen sich auf die einzelnen Ansichten und sind mit dem selben Namen wie die Ansichten versehen.

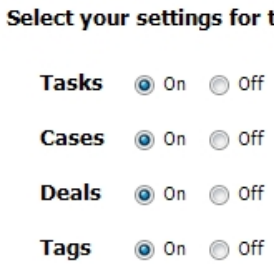

Abbildung 5.1: Aktivierung der optionalen Funktionen in der Ansicht *Account*

Beim Hinzufügen einer Ansicht erscheint in der Titelleiste der Anwendung je nach Art der Ansicht im Bereich *Feature*, *Option* oder *Action* ein Karteireiter für die entsprechende Ansicht (siehe Kapitel 3.3.2). Äquivalent dazu verschwindet beim Ausschalten einer Funktion der dazugehörige Karteireiter in der Titelleiste. Falls zusätzliche Karteireiter zu dieser Ansicht im Bereich *Document* der Titelleiste zu sehen sind, so werden auch diese Karteireiter entfernt.

Abbildung 5.2 zeigt das Beispiel zum Deaktivieren der optionalen Funktion *Tasks*. Abbildung 5.2a zeigt den linken Ausschnitt der Titelleiste mit der zusätzlich aktivierten Funktion *Tasks* (siehe Karteireiter *Tasks*). Abbildung 5.2b zeigt, dass mit dem Entfernen der Funktion *Tasks* der dazugehörige Karteireiter in der Titelleiste verschwindet.

Das Hinzufügen und Entfernen dieser Funktionen beschränkt sich nicht nur auf die entsprechenden Ansichten. Ist ein Teil einer Funktion in einer anderen Ansicht enthalten, beispielweise werden zu einer Kontaktperson im rechten Bereich der Ansicht ihre geplanten Aufgaben aufgelistet, so hat das Hinzufügen oder Entfernen der Funktion *Tasks* auf diesen Bereich der Ansicht ebenfalls Auswirkungen.

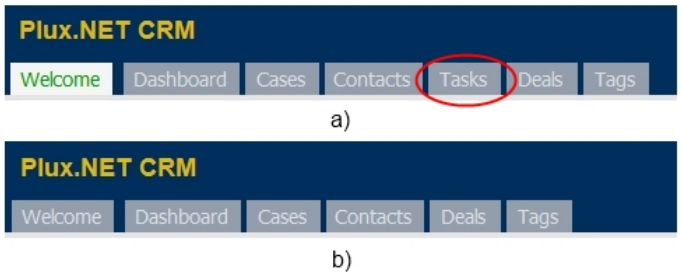

Abbildung 5.2: Linker Ausschnitt der Titelleiste mit a) aktivierter Funktion *Tasks*, b) deaktivierter Funktion *Tasks*

Wie in Abbildung 5.3a zu erkennen ist, werden beim Hinzufügen der Funktion *Tasks* die geplanten Aufgaben einer Kontaktperson und die Schaltfläche zum Hinzufügen einer neuen Aufgabe angezeigt. Entfernt man die Funktion *Tasks*, so sind sowohl die Aufgaben zu dieser Person als auch die Schaltfläche nicht mehr zu sehen (siehe Abbildung 5.3b). Analog dazu verhält sich beim Hinzufügen oder Entfernen der Funktion *Deals* das Anzeigen von Geschäften mit einem Kunden und bei der Funktion *Cases* das Darstellen der Projekte eines Kunden.

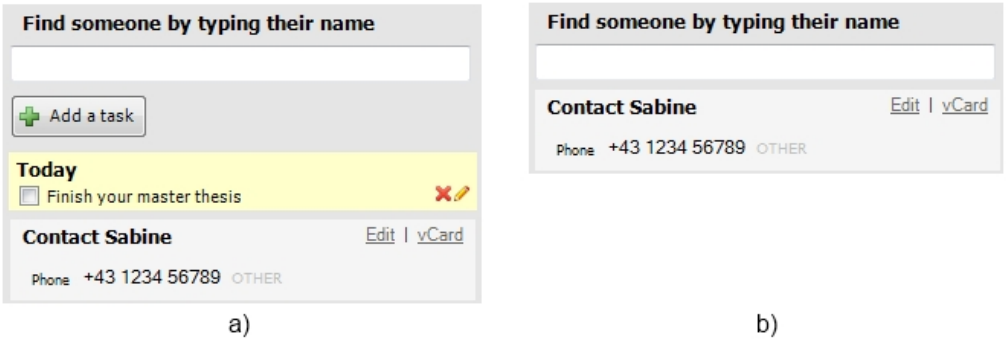

Abbildung 5.3: Tasks-Bereich der Detailansicht eines Kunden mit a) aktivierter Funktion *Tasks*, b) deaktivierter Funktion *Tasks*

Es gibt einige Ausnahmen, wo das Einstellen der optionalen Funktionen keinen Einfluss auf entsprechende Teilbereiche einer Ansicht hat. Ein Beispiel stellt hier das „Schwarze Brett“ dar. Das Auflisten der letzten Aktivitäten des Benutzers bleibt auf der linken Seite des „Schwarzen Bretts“ unverändert. Sämtliche Einstellungen der optionalen Funktionen haben keine Auswirkungen auf diese Auflistung. Der einzige Unterschied von den aktivierten zu den deaktivierten Funktionen liegt darin, dass gewisse Weiterleitungen zu Detailansichten nicht mehr funktionieren. Ist beispielsweise die Funktion *Deals* deaktiviert und ist am „Schwarzen Brett“ ein Eintrag angelegt, dass der angemeldete Benutzer ein Geschäft abgeschlossen hat, so bleibt dieser Eintrag angezeigt. Möchte man die Details zu diesem Geschäft erfahren und klickt auf diesen Eintrag, so kommt man in diesem Fall

nicht zur Detailansicht des Geschäfts. Um zu dieser Detailansicht zu gelangen, muss die Funktion *Deals* aktiviert sein.

Im Gegensatz dazu werden im rechten Bereich des „Schwarzen Bretts“ die Aufgaben des angemeldeten Benutzers angezeigt. Diese Anzeige ist abhängig von der Einstellung der optionalen Funktion *Tasks*. Sie verhält sich genau so wie im Beispiel von Abbildung 5.3 beschrieben.

5.1.3 Implementierung der optionalen Funktionen

Betrachtet man die Basisfunktionen und die optionalen Funktionen genauer, so ist in Plux-CRM sowohl jede Basisfunktion als auch jede optionale Funktion als eigenes Plugin implementiert. Beim Kompillieren jedes dieser Projekte entsteht eine eigene .dll-Datei, die gleichzeitig als Plux.NET-Plugin zu sehen ist. Bei der Auslieferung von Plux-CRM werden alle .dll-Dateien der einzelnen Basisfunktionen und optionalen Funktionen mitgeliefert. Beim Starten von Plux-CRM erkennt die Plux.NET-Laufzeitumgebung alle Plux.NET-Plugins, die im *plugins*-Ordner des Ausführungsverzeichnis liegen. Je nach den Einstellungen des Benutzers werden die Plugins aktiviert und in der Anwendung angezeigt oder bleiben deaktiviert.

Hat ein Benutzer eine optionale Funktion aktiviert, so wird im Speicher (`Repository`) der Plux.NET-Laufzeitumgebung (`Runtime`) die entsprechende .dll-Datei registriert. Alle Komponenten, die in der aktivierten .dll-Dateien liegen, werden von den zugehörigen Steckplätzen registriert. Hat ein Steckplatz eine Komponente nicht nur registriert, sondern möchte sie auch einstecken, so erzeugt vorher die Laufzeitumgebung aus der Komponente einen Stecker und fügt diesen dem passenden Steckplatz zu. Nun kann die entsprechende Funktion in der Anwendung verwendet werden.

Als Beispiel dient die optionale Funktion `Tasks`. Im aktivierten Zustand dieser Funktion ist in der Detailansicht eines Kunden eine Schaltfläche zum Hinzufügen geplanter Aufgaben zu sehen. Abbildung 5.4a zeigt die Komponente `AddTaskButtonTask`, die sich im Steckplatz `CRM.Customer.Info.Tasks` der Detailansicht eines Kunden (`CustomerInfoView`) befindet. Diese Komponente zeigt die Schaltfläche zum Erfassen einer neuen Aufgabe an. Im deaktivierten Zustand der Funktion *Tasks* verschwindet diese Schaltfläche. Abbildung 5.4b zeigt, dass in diesem Fall die zugehörige Komponente `AddTaskButtonTask` nicht mehr der `CustomerInfoView`-Komponente zugewiesen ist.

Deaktiviert der Benutzer eine Funktion, so wird das dazugehörige Plugin beim *Repository* der Plux.NET-Laufzeitumgebung, das alle Plugins verwaltet, abgemeldet. Gleichzeitig werden alle Komponenten des Plugins aus ihren Steckplätzen entfernt, in denen sie sich zu diesem Zeitpunkt befinden. Aktiviert der Benutzer diese Funktion wieder, so wird das

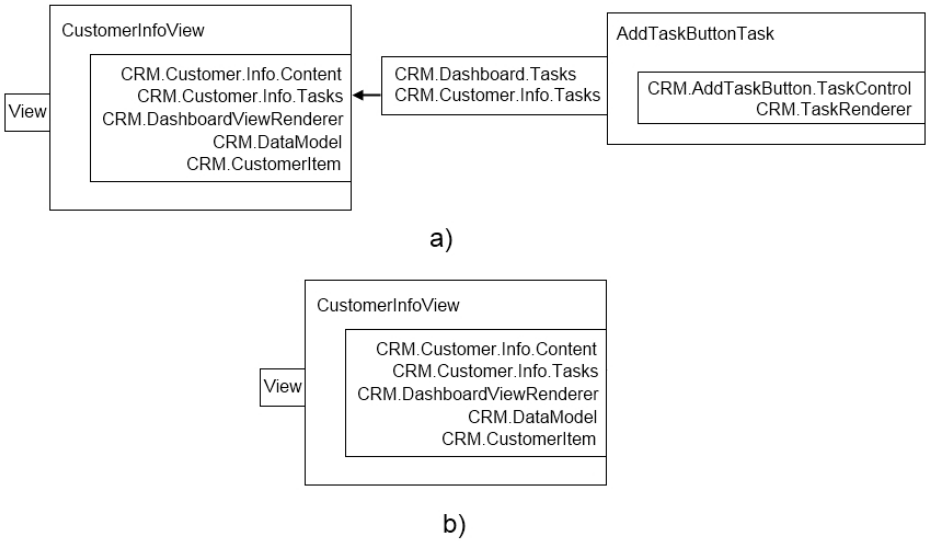

Abbildung 5.4: Die Komponente AddTaskButtonTask mit a) aktivierter Funktion *Tasks*, b) deaktivierter Funktion *Tasks*

Plugin beim *Repository* wieder angemeldet und die Komponenten dieses Plugins werden in ihre Steckplätze wieder eingesteckt. Das An- und Abmelden der Plugins beim *Repository* erfolgt während der Laufzeit. Plux-CRM muss für diese Aktionen nicht neu gestartet werden.

Ein Beispiel: Der Benutzer kann die optionale Funktion *Tasks* über die Ansicht *Account* aktivieren oder deaktivieren. Listing 5.1 zeigt das manuelle Hinzufügen und Entfernen dieser Funktion im Quellcode. Zuerst wird das Plugin über den Namen der .dll-Datei gesucht. Hat die Plux.NET-Laufzeitumgebung die Datei gefunden, so wird sie je nach den Einstellungen des Benutzers registriert (im Fall von „On“) oder abgemeldet (im Fall von „Off“).

Listing 5.1: Quellcode zum Hinzufügen und Entfernen der optionalen Funktion Tasks

```
1 //sender is the radiobutton "On", to register the plugin "CRM.TasksView.dll"
2 PluginInfo tasksPlugin = Runtime.Repository.PluginInfos["CRM.TasksView.dll"];
3 if (tasksPlugin == null) return;
4 if (((RadioButton)sender).Checked) tasksPlugin.Register(); // "On" is activated
5 else tasksPlugin.Deregister(); // "Off" is activated
```

5.2 Erweiterungen von Drittherstellern

In Plux-CRM ist es möglich, Plugins von Drittherstellern zu verwenden. Plux-CRM ist nicht nur auf die bis jetzt implementierten Funktionen beschränkt. Es können jederzeit zusätzliche Plugins programmiert und anhand der .dll-Dateien in die Software mit aufge-

nommen werden. Das Einzige, das beim Programmieren solcher Plugins zu beachten ist, sind die zu verwendenen Schnittstellen, anhand der Plux-CRM die neuen Plugins erkennt und verwenden kann.

5.2.1 Hinzufügen einer Dritthersteller-Komponente

Die Filterfunktion bei der Anzeige der Kunden zeigt das Verwendung von Dritthersteller-Komponenten. In der Ansicht *Contacts* kann der Benutzer anhand eines Auswahlfeldes steuern, welche Kunden angezeigt werden. Über den Filter „All people“ schränkt der Benutzer die Auflistung der Kunden so ein, dass nur Kontaktpersonen, aber keine Firmen erscheinen. Wählt der Benutzer „All companies“, erscheinen alle Firmen, aber keine Kontaktpersonen (siehe Abbildung 5.5a).

Nun hat der Benutzer die Möglichkeit zur Laufzeit einen dritten Filter einzubauen, bei dem beispielsweise sowohl alle Kontaktpersonen als auch alle Firmen angezeigt werden. Wichtig beim Hinzufügen eines neuen Filters ist, den Aufbau der Filterfunktion zu kennen und die vordefinierten Schnittstellen einzuhalten.

Ist der neue Filter einmal hinzugefügt (zum Beispiel mit Hilfe der Funktion `register` des Kompositionsdienstes) und wurden alle Schnittstellen korrekt eingehalten, so erkennt die Plux.NET-Laufzeitumgebung die neue Komponente und fügt sie dem Auswahlfeld hinzu (siehe Abbildung 5.5b). Wählt der Benutzer nun den neuen Filter aus, ändert sich automatisch die Auflistung der Kunden. Das Ergebnis ist das Anzeigen aller Kontaktpersonen und Firmen. Falls der Benutzer diesen Filter wieder deaktiviert (zum Beispiel mit Hilfe der Funktion `unregister` des Kompositionsdienstes), entfernt die Plux.NET-Laufzeitumgebung den Filter aus dem Auswahlfeld und der alte Zustand der Ansicht ist wieder hergestellt.

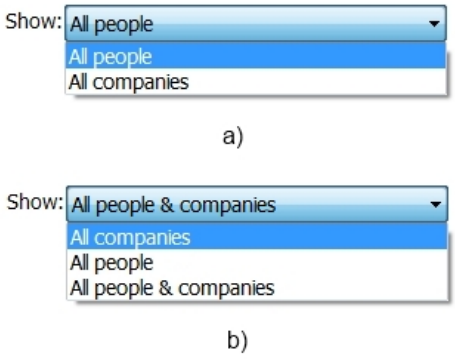

Abbildung 5.5: Auswahlfeld mit a) zwei Filtern, b) und neuem Filter „All people & companies“

5.2.2 Implementierung einer Dritthersteller-Komponente

Im Detail sieht der Aufbau der Filterfunktion wie folgt aus. Das Auswahlfeld der Filter ist eine eigene Komponente in Form des ContentControls `ContactsFilterContentControl`, das über den Steckplatz `CRM.Contacts.List.Elements` im Inhalt `ContactsListContent` der Ansicht *Contacts* steckt. Das Auswahlfeld dieser Komponente ist als `PlugComboBox` implementiert (siehe Kapitel 4.2). Ihr wird der Steckplatz `CRM.Contacts.Filter` der Komponente `ContactsListContent` zugeordnet. Dieser Steckplatz steuert die eigentliche Filterfunktion. Die `PlugComboBox` reagiert auf die Interaktionen des Benutzers in Bezug auf den ausgewählten Wert des Auswahlfeldes und verwaltet das Umstecken der entsprechenden Filter-Komponenten des Steckplatzes `CRM.Contacts.Filter`. Abbildung 5.6 zeigt das Zusammenspiel der einzelnen Komponenten.

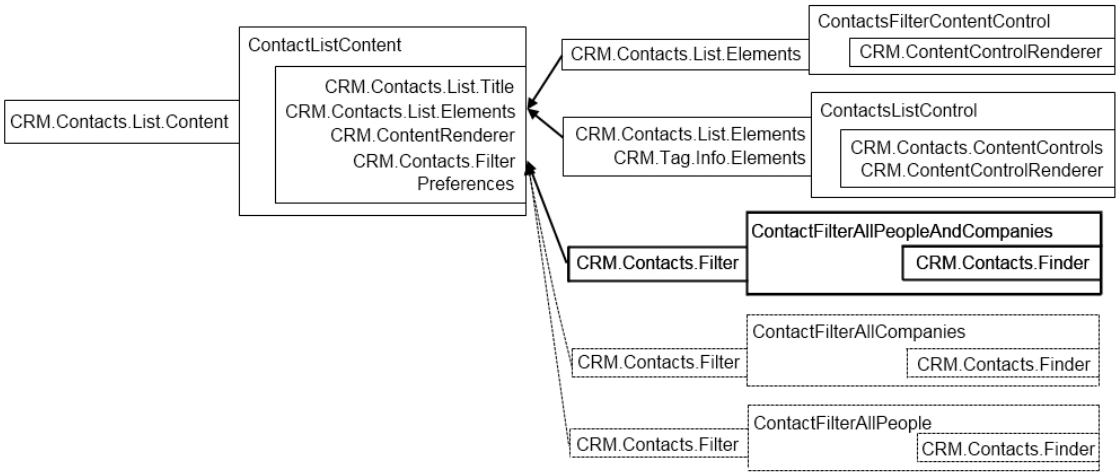

Abbildung 5.6: Die Zusammenhänge zwischen ContactsListContent, FilterAllPeopleAndCompanies, ContactsFilterContentControl und ContactsListControl

Kommt ein neuer Filter in Form einer Komponente, beispielsweise der neue Filter `ContactFilterAllPeopleAndCompanies`, über den Steckplatz `CRM.Contacts.Filter` hinzu, registriert der Steckplatz sofort diese Komponente. Die Komponente `ContactsFilterContentControl` wird ebenfalls über diesen neuen Filter informiert und stellt den neuen Filter im Auswahlfeld zur Verfügung. In Fall der Komponente `ContactFilterAllPeopleAndCompanies` erscheint im Auswahlfeld der Wert „All people & companies“ (siehe Abbildung 5.5b). In Abbildung 5.6 ist die neue Filter-Komponente `ContactFilterAllPeopleAndCompanies` fett umrandet. Die Komponenten `ContactFilterAllPeopleAndCompanies` und `ContactFilterAllPeopleAndCompanies` sind ebenfalls zu sehen. Sie sind strichliert umrandet, da sie nicht im Steckplatz `CRM.Contacts.Filter` eingesteckt, sondern nur registriert sind. Im Auswahlfenster von Abbildung 5.5b sind alle drei Filter zu sehen.

Entscheidend bei der Implementierung solch eines Filters ist, dass die zugehörige Komponente die Schnittstelle `ICustomerFilter` implementiert, damit der Filter von der Plux.NET-Laufzeitumgebung richtig erkannt wird (siehe Listing 5.2). Die Schnittstelle `ICustomerFilter` definiert die Eigenschaft `FilteredCustomers`, über die alle gefilterten Datensätze zurückgegeben werden. Zusätzlich legt die Schnittstelle über das Attribut `SlotDefinition` die Steckplatz-Definition *CRM.Contacts.Filter* fest. Über den Parameter `Text` wird der Ausgabetext definiert, der im Auswahlfeld der Kundenansicht erscheint.

Listing 5.2: Die Schnittstelle ICustomerFilter

```
1  [SlotDefinition("CRM.Contacts.Filter")]
2  [Param("Text", typeof(string))]
3  public interface ICustomerFilter {
4    IList<ICustomer> FilteredCustomers { get; }
5    string Text { get; }
6    ...
7  }
```

Listing 5.3 zeigt einen Ausschnitt des neuen Filters `ContactFilterAllPeopleAndCompanies`. Zu erkennen ist die Implementierung der Schnittstelle `ICustomerFilter` mit der Eigenschaft `FilteredCustomers` und den Kontaktpersonen und Firmen des Datenmodells als Rückgabewert. Wählt der Benutzer den angeführten Filter im Auswahlfeld aus, so entfernt die Plux.NET-Laufzeitumgebung die alte Filter-Komponente, die sich zu diesem Zeitpunkt im Steckplatz `CRM.Contacts.Filter` befindet. Zusätzlich fügt die Laufzeitumgebung die Komponente `FilterAllPeopleAndCompanies` dem erwähnten Steckplatz hinzu.

Listing 5.3: Die Klasse FilterAllPeopleAndCompanies

```
1  [Extension("ContactFilterAllPeopleAndCompanies")]
2  [Plug("CRM.Contacts.Filter", AutoPlug = false, ...)]
3  [Slot("CRM.Contacts.Finder", AutoPlug = false, Multiple = false, Unique = true,
4    OnPlugged = "FinderPlugged", ...)]
5  [ParamValue("Text", "All people & companies")]
6  public class FilterAllPeopleAndCompanies : ICustomerFilter {
7    private IContactFinder finder;
8    private IContactModel model;
9    public string Text {
10     get { return (string)Plug.PlugTypeInfo.ParamValues["Text"].Value; }
11   }
12   public IList<ICustomer> FilteredCustomers {
13     get {
14       IList<ICustomer> filteredCustomers = new List<ICustomer>();
15       foreach (IContact c in model.Contacts) filteredCustomers.Add(c);
16       foreach (ICompany c in model.Companies) filteredCustomers.Add(c);
17       if (finder != null) {
18         // filters the list with additional criteria of the ContactsFinder
19         filteredCustomers = FilterRenderer.GetFilteredList(finder, filteredCustomers);
20       }
21       return filteredCustomers;
22     }
23   }
```

```
24    public void FinderPlugged(object sender, PlugEventArgs args)  {
25      finder = args.Extension as IContactFinder;
26    } ...
27  }
```

5.2.3 Erweiterung der Dritthersteller-Komponente

Auffällig bei Listing 5.3 ist die Angabe des zusätzlichen Steckplatzes `CRM.Contacts.Finder`. Dieser Steckplatz sorgt über eine spezielle Komponente dafür, dass weitere Suchkriterien im Filter beachtet werden. Zu diesen Suchkriterien zählen beispielsweise der Name, die Anschrift, die Email-Adresse oder die Telefonnummer eines Kunden. Die Schnittstelle `IContactFinder` definiert diese erweiterten Suchkriterien (siehe Listing 5.4).

Listing 5.4: Die Schnittstelle IContactFinder

```
1  [SlotDefinition("CRM.Contacts.Finder")]
2  public interface IContactFinder {
3    string Name { get; set; }
4    string City { get; set; }
5    string State { get; set; }
6    string Country { get; set; }
7    string Zip { get; set; }
8    string Phone { get; set; }
9    string Email { get; set; }
10   string Position { get; set; }
11 }
```

Tippt der Benutzer im Formular oberhalb der Auflistung der Kunden (siehe Abbildung 5.7) entsprechende Kriterien zum Suchen bestimmter Datensätze ein, so erzeugt die Plux.NET-Laufzeitumgebung die Komponente `ContactFinder`, die von `IContactFinder` abgeleitet ist, und steckt sie in den Steckplatz `CRM.Contacts.Finder` des ausgewählten Filters ein. Verwendet der Benutzer keine zusätzlichen Suchkriterien, werden alle Komponenten aus diesem Steckplatz entfernt und er ist somit leer.

Alle bereits in Plux-CRM integrierten Filter enthalten den Steckplatz `CRM.Contacts.Finder` und führen dieselbe Funktion aus. Bei einem neuen Filter ist nur die Schnittstelle dieses Steckplatzes zu beachten. Um die Aktion auszuführen, dass eine Komponente in diesen Steckplatz ein- oder ausgesteckt wird, braucht sich der Filter nicht kümmern. Diese Funktion übernehmen bereits andere Komponenten. Dazu zählt die Komponente, die das Formular der zusätzlichen Suchkriterien implementiert (`ContactsFinderTitle`). Wichtig bei einer Filter-Komponente ist, dass sie die Eigenschaft `FilteredCustomers` berücksichtigt, sodass zusätzliche Suchkriterien anhand der Komponente `ContactFinder` möglich sind. Zeile 17-20 in Listing 5.3 zeigen diese Berücksichtigung.

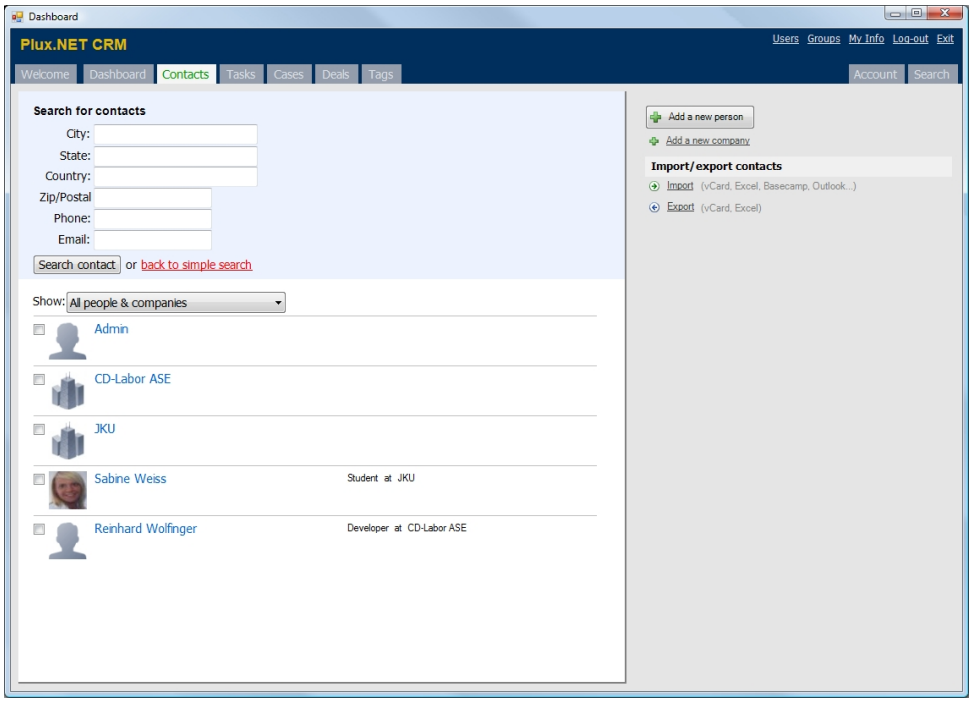

Abbildung 5.7: Formular zum Suchen von Kunden anhand bestimmter Kriterien

Die Auflistung der Kunden ist ebenfalls eine Komponente in Form des ContentControls `ContactsListControl`. Sie reagiert auf das Einstecken einer Komponente sowohl in den Steckplatz `CRM.Contacts.Filter` als auch in den Steckplatz `CRM.Contacts.Finder`. Sie reagiert auf das Einsteck-Ereignis des jeweiligen Steckplatzes (mittels eines *Listeners*). Wurde ein derartiges Ereignis ausgelöst, sucht sich die Komponente `ContactsListControl` den eingesteckten Filter und aktualisiert die Liste der Kunden anhand der Suchkriterien. Listing 5.5 zeigt einen Ausschnitt, wie sich die Komponente `ContactsListControl` an das Einsteck-Ereignis des Steckplatzes `CRM.Contacts.Filter` hängt und entsprechend darauf reagiert (`FilterPlugged`).

Listing 5.5: Die Klasse ContactsListControl

```
1  [Extension("ContactsListControl")]
2  [Plug("CRM.Contacts.List.Elements", OnPlugging = "OnPlugging")]
3  ...
4  // shows all customers of the user in a list
5  public class ContactsListControl : AbstractContentControl {
6    public void OnPlugging(object sender, CancelPlugEventArgs args) {
7      // during this control is plugging, it looks for the filter-extension
8      // in the slot "CRM.Contacts.Filter"
9      SlotInfo filterSlot = ExtensionInfoEx.FindSlot(args.SlotInfo.ExtensionInfo,
10        "CRM.Contacts.Filter");
11     filterSlot.Plugged += FilterPlugged;
12     ...
13   }
14
15
```

```
16    // if a new filter is plugged, the list of the customers has to be refreshed
17    private void FilterPlugged(object sender, PlugEventArgs args) {
18      IMemberFilter filter = (IMemberFilter)args.PlugInfo.ExtensionInfo.Object;
19      PlugContactsListControl(filter); // refreshes the list with the customers
20      OnControlChanged(this, EventArgs.Empty);
21    }
22    ...
23  }
```

6 Bewertung von Plux.NET anhand von Plux-CRM

Die Evaluierung von Plux.NET zeigt anhand des Fallbeispiels Plux-CRM sowohl die Vorteile als auch die Nachteile bei der Verwendung von Plux.NET auf. Eine Bewertung bringt immer mit sich, dass so objektiv sie erscheint, immer auch subjektive Aspekte des Bewerters mit einfließen.

Die Vorteile, die Plux.NET mit sich bringt, waren bereits zu Beginn der Plux-CRM Entwicklung bekannt. Sie wurden im Laufe dieser Evaluierung noch einmal genau betrachtet. Einige Nachteile von Plux.NET wurden ebenfalls bereits zu Beginn der Plux-CRM Entwicklung notiert. Weitere kamen im Laufe der Entwicklung von Plux-CRM zum Vorschein. Dazu zählt die hohe Einarbeitungszeit.

6.1 Vorteile von Plux-CRM

Die Vorteile von Plux.NET liegen auf der Hand. Sie treten aufgrund der komponentenbasierten Architektur im Gegensatz zu monolithischen Anwendungen auf. Bei monolithischen Anwendung sind die Funktionen von Anfang an vordefiniert und es nicht möglich, diese Funktionen zu verändern oder zu erweitern. Bei komponentenbasierten Programmen ergibt sich der Vorteil der Personalisierbarkeit aus dem individuellen Anpassen der Benutzeroberfläche an den Benutzer, wie in Kapitel 5.1 beschrieben. Das Hinzufügen von Komponenten durch eigene Entwickler oder durch Drittanbieter bringt einen weiteren Vorteil, die Erweiterbarkeit, hervor.

Es ist schwierig, die auftretenden Vorteile streng voneinander zu trennen, da sie sich gegenseitig beeinflussen und zum Teil voneinander abhängig sind. Beispielsweise folgt aus der Personalisierbarkeit und Erweiterbarkeit von Plux-CRM die hohe Flexibilität. Andererseits ist die Anwendung aufgrund der Flexibilität von Plux-CRM erweiterbar.

6.1.1 Personalisierbarkeit

Zu den größten Vorteilen von Plux-CRM zählt die Personalisierbarkeit. Plux-CRM bietet jedem Benutzer die Möglichkeit, anhand gewisser Einstellung in der Anwendung die

Benutzeroberfläche im gewissen Maß selbst zu konfigurieren. Je nach den Einstellungen des Benutzers werden unterschiedliche Komponenten in den einzelnen Steckplätzen eingesteckt. In Plux-CRM beschränken sich die Einstellungen auf das Anzeigen oder Verbergen gewisser zur Verfügung stehender Ansichten (zum Beispiel die Ansichten der Projekte, Geschäfte oder Aufgaben). Plux-CRM ist so konzipiert, dass alle Komponenten und Benutzersteuerelemente, die im Zusammenhang mit einer einzelnen Ansicht stehen, das selbe Verhalten aufweisen, wie die Ansicht selbst. Das bedeutet, dass diese Benutzersteuerelemente sichtbar sind, wenn die entsprechende Ansicht vorhanden ist, und verschwinden, wenn die zugehörige Ansicht nicht zur Verfügung steht. Diese Benutzersteuerelemente reagieren auch dann, wenn sie nicht Teil der entsprechenden Ansicht sind, sondern in einer anderen Ansicht vorkommen. Beispielsweise sind die Aufgaben eines Benutzers in mehreren Ansichten sichtbar, sind aber Teil der Ansicht „Tasks“ und reagieren auf die Einstellungen dieser Ansicht. Das Konfigurieren der Benutzeroberfläche durch den Benutzer erfolgt zur Laufzeit und erfordert keinen Neustart der Anwendung.

Die Personalisierbarkeit beschränkt sich nicht nur auf das individuelle Konfigurieren der Benutzeroberfläche durch den Benutzer. Die Entwickler von Plux-CRM können selbst die Benutzeroberfläche an den Benutzer anpassen, indem sie anhand von speziellen Befehlen (zum Beispiel über die Plux.NET-Konsolenanwendung) Komponenten ein- oder ausstecken oder diese mit speziellen Parametern versehen. Der Entwickler führt diese Befehle ebenfalls zur Laufzeit aus, ohne dass der Quellcode neu übersetzt oder die Anwendung neu gestartet werden muss. Die Einstellungen des Benutzers beschränken sich auf die Anordnung der einzelnen Ansichten. Ein Entwickler kann im Gegensatz dazu jede noch so kleine Komponente ein-, aus- oder umstecken und somit jedes einzelne Benutzersteuerelement in der Anwendung steuern. Der Discovery- und Composer-Kern des zugrundeliegenden Plux.NET-Frameworks ermöglichen diese dynamische Konfiguration durch den Benutzer oder Entwickler zur Laufzeit.

6.1.2 Erweiterbarkeit

Ein weiterer Vorteil von Plux-CRM besteht darin, dass die Software schnell und einfach um beliebige Komponenten erweitert werden kann. Diese Komponenten können zur Laufzeit der Software hinzufügt werden, ohne dass die Software neu gestartet werden muss. Die Erweiterbarkeit beschränkt sich nicht nur auf das Hinzufügen neuer Komponenten. Auch das Austauschen bestehender Komponenten ist in Plux-CRM möglich. Plux-CRM ist so implementiert, dass Benutzerinteraktionen durch An- oder Abstecken von Komponenten umgesetzt sind. Klickt zum Beispiel ein Benutzer auf eine Schaltfläche, wird im Hintergrund eine Komponente in einen Steckplatz eingesteckt. Dadurch kann jede Komponente durch eine andere ersetzt werden.

Das Aufnehmen neuer Komponenten umfasst das Hinzufügen der entsprechenden .dll-Dateien in das Ausführungsverzeichnis der Anwendung. Aufgrund des Discovery-Kerns des Plux.NET-Frameworks erkennt die Software diese neue Datei, identifiziert die enthaltenen Komponenten und fügt diese den entsprechenden Steckplätzen hinzu. Sobald eine dieser Komponenten in einem Steckplatz eingesteckt wird, reagiert Plux-CRM darauf und die neue Komponente ist in der Anwendung vorhanden. Falls es sich um ein neues Benutzersteuerelement handelt, wird dieses sofort in der Anwendung sichtbar.

Ein Plugin kann Komponenten unterschiedlicher Größen enthalten. Sie kann eine komplett neue Ansicht mit all ihren Unterkomponenten umfassen oder sie besitzt nur einzelne Komponenten, die Teilbereiche einer Ansicht oder eines Benutzersteuerelementes erweitern. Ein Beispiel für eine große Erweiterung ist eine Ansicht zum Erstellen und Bearbeiten von Rechnungen über Projekte mit den Kunden. Diese könnte detaillierte Kostenaufstellungen eines Projektes mit den involvierten Kunden und Entwicklern umfassen. Eine Erweiterung von Plux-CRM im kleinen Umfang würde eine neue Filterkomponente für das Anzeigen von Kunden darstellen (siehe Kapitel 5.2).

Für das Implementieren einer neuen Komponente muss der Entwickler nicht in den ursprünglichen Quellcode der Anwendung eingreifen. Es reicht aus, dass er die Schnittstellen für die Komponenten kennt, die er erweitern möchte. Alle Schnittstellen der einzelnen Plux-CRM Komponenten sind in einem eigenen Contract (siehe Kapitel 2.1.1) zusammengefasst. Nur das Einhalten dieser Schnittstellen benötigt der Entwickler zum Implementieren neuer Komponenten.

Das Hinzufügen neuer Komponenten ist nicht nur auf die eigenen Entwickler von Plux-CRM beschränkt. Auch neue Komponenten von Drittherstellern, die den vordefinierten Schnittstellen entsprechen, können in Plux-CRM integriert werden.

6.1.3 Flexibilität

Aufgrund der Personalisierbarkeit und Erweiterbarkeit von Plux-CRM ergibt sich der Vorteil der Flexibilität. Plux-CRM ist nicht wie andere monolithische Programme nur auf ihre vorhandenen Funktionen beschränkt, sondern reagiert flexibel auf die Konfigurationen eines Benutzers oder das Hinzukommen oder Entfernen von Komponenten.

Nicht nur in diesem Zusammenhang weist Plux-CRM ein flexibles Verhalten auf. Auch in Hinblick auf die Datenhaltung ist die Flexibilität von Plux-CRM gewährleistet. Plux-CRM ist so implementiert, dass die Anwendung beim Starten die Daten von einer eigenen Datei liest und sie während der Laufzeit im Hauptspeicher hält. Zum Programmende serialisiert sie die Daten wieder in eine Datei. Das Datenmodell, das das Laden und Speichern der Daten verwaltet, befindet sich in einem eigenen Plugin. Die Datenzugriffsschicht bleibt

somit austauschbar, da lediglich dieses Plugin durch ein neues zu ersetzen ist. Das Auslagern des Datenmodells in ein eigenes Plugin wurde deshalb berücksichtigt, damit man später die dateibasierte Datenhaltung gegen eine Datenbank austauschen kann.

6.2 Nachteile von Plux.NET

Genauso wie die Vorteile einer Software zu einer Bewertung zählen, so sind die Nachteile ebenfalls Bestandteil einer Bewertung. Bei Plux.NET sind im Laufe der Evaluierung drei Nachteile aufgetreten, die wesentlich geringer zu gewichten sind als die angeführten Vorteile.

6.2.1 Langsames Laufzeitverhalten

Ein Nachteil von Plux.NET ist auf das langsame Laufzeitverhalten zurückzuführen. Dieses Verhalten kommt aufgrund des dynamischen Verhaltens der in der Anwendung verwendeten Komponenten zustande. Zum Startzeitpunkt einer Plux.NET-Anwendung werden nur jene Komponenten geladen, die zu diesem Zeitpunkt benötigt werden. Alle weiteren Komponenten, die zur Verfügung stehen, haben zu diesem Zeitpunkt keinen Einfluss auf die Anwendung. Reagiert die Anwendung auf eine Benutzereingabe, so hat dies meist zur Folge, dass gewisse Änderungen an der Benutzeroberfläche vorgenommen werden, die wiederum bewirken, dass sich im Hintergrund die Komposition ändert. Komponenten werden demnach erst so spät wie möglich erzeugt und ihren Steckplätzen hinzugefügt. Dies hat zur Folge, dass eine Plux.NET-Anwendung im Vergleich zu konventionellen Programmen beim Start zwar schneller geladen wird, aber im laufenden Zustand verlangsamt das Umstecken den Bildaufbau. Der Grund dafür ist, dass das verspätete Laden der Komponenten auf Kosten der Laufzeit der Plux.NET-Anwendung von statten geht.

6.2.2 Spätes Auflösen von Abhängigkeiten

Das späte Laden der einzelnen Software-Komponenten hat ebenfalls zur Folge, dass das Auflösen von Abhängigkeiten zwischen Assemblies erst zur Laufzeit erfolgen. Das bedeutet, dass bei Komponenten, die voneinander abhängig sind und die gegenseitige Objekte verwenden, die Objekte erst dann auf ihre Existenz überprüft werden, wenn diese bereits benötigt werden. Es kann der Fall eintreten, dass Komponenten in der falschen Reihenfolge geladen werden. Dies kann zur Folge haben, dass eine Komponente, die von einer anderen Komponente abhängig ist, zuerst geladen wird und aufgrund des verspäteten Ladens der zweite Komponente ihre Abhängigkeit verletzt wird. Somit werden die weiteren Arbeitsschritte der ersten Komponente, die nur anhand der zweiten Komponente ausgeführt werden können, unterbrochen und zu keinem späteren Zeitpunkt ausgeführt.

Ein Beispiel hierfür ist das Plugin *CRM.DataModel* von Plux-CRM. Dieses Plugin enthält das gesamte Datenmodell und ist für die Verwalten aller Daten, wie die der Kunden, Geschäfte und Projekte, verantwortlich. Diese Komponente ist von allen weiteren Komponenten entkoppelt und nur über bestimmte Steckplätze zugänglich. Alle weiteren Komponenten von Plux-CRM zeigen meist in der Benutzeroberfläche Steuerelemente an, die bestimmte Daten des Datenmodells enthalten. Daraus folgt, dass beim Laden einer Komponente mit einem derartigen Steuerelement, die Komponente mit dem Datenmodell bereits vorhanden sein muss und sich in einem Steckplatz befinden muss.

Eine Lösung für dieses Problem ist einerseits, dass bei der Programmierung die Reihenfolge des Ladens der einzelnen Komponenten festgelegt wird. So kann gewährleistet werden, dass zu einem bestimmten Zeitpunkt gewisse Komponenten bereits vorhanden sind und andere auf ihnen aufbauen können. Andererseits ist es möglich, die Reihenfolge des Ladens zu belassen und bei der Verwendung von anderen Komponenten zu überprüfen, ob diese bereits existieren. Ist dies nicht der Fall, so muss die Komponente, die eine Abhängigkeit auf die zweite Komponente hat, solange warten, bis die zweite existiert. Erst dann kann die erste Komponente mit ihren Arbeitsschritten fortfahren.

6.2.3 Lange Einarbeitungszeit

Plux.NET hat mittlerweile einen sehr großen Umfang angenommen. Aufgrund seiner Komplexität benötigt ein Entwickler, der sich vorher mit dem Thema der plugin-basierten Struktur noch nicht auseinander gesetzt hat, einige Zeit, um die Idee dahinter zu verstehen und mit Plux.NET arbeiten zu können. Bei mir dauerte es ca. 100 Stunden.

Dabei fließt auch der Faktor der unzureichenden Dokumentation von Plux.NET mit ein. Ein Framework, das sowohl über eine ausreichende Dokumentation als auch über ein umfangreiches Benutzerhandbuch verfügt, ist wesentlich leichter zu erlernen, als ein Framework, bei dem die Funktionsweise anhand der implementierten Algorithmen zu verstehen ist. Plux.NET verfügt über nur wenig Dokumentation. Eine hilfreiche Dokumentation ist die Dissertation von R. Wolfinger [3]. Sie beschreibt die einzelnen Bestandteile von Plux.NET und zeigt detailliert die Abläufe der Plux.NET-Komposition auf.

Eine Lösung, um die Einarbeitungszeit zu reduzieren, wäre, die wichtigsten und am häufig verwendetsten Eigenschaften und Methoden von Plux.NET mit einer kurzen Dokumentation zu versehen. Zusätzliche kurze Kommentare an komplizierten Stellen im Quellcode wären für die Verwendung von Plux.NET ebenfalls hilfreich.

7 Technische Daten von Plux-CRM

Jede Software wird anhand bestimmter Kennzahlen, den technischen Daten, gemessen. Es gibt Kennzahlen, die für alle Programme interessant sind, und es gibt andere Kennzahlen, die nur bei komponentenbasierten Programmen, wie Plux.NET, interessant sind. Die Kennzahlen, die für alle Programme interessant sind, reichen vom benötigten Speicherplatz bis hin zur Anzahl der Quellcode-Zeilen. Die Kennzahlen, die nur für Plux.NET interessant sind, umfassen die Anzahl der verwendeten Komponenten, Steckplätze und Stecker.

7.1 Allgemeine Daten

Die allgemeinen Daten sind bei jedem Programm zu ermitteln. Anhand dieser Daten können alle Programme miteinander verglichen werden. Zu den allgemeinen Daten zählen der benötigte Speicherplatz, der in diesem Fall in Megabytes berechnet wird, und die Anzahl der Quellcode-Zeilen (*Lines of Code - LoC*). Tabelle 7.1 zeigt die Messwerte von Plux-CRM. Beim Messwert *Benötigter Speicherplatz* wurde zwischen dem benötigten Speicherplatz des Quellcodes und den ausgelieferten .exe- und .dll-Dateien unterschieden. Plux-CRM verfügt über eine .exe-Datei, die das auszuführende Programm darstellt, und über 19 .dll-Dateien. Bei der Anzahl der .dll-Dateien wurden nur die Plux-CRM Plugins und nicht die Plugins der Plux.NET-Laufzeitumgebung berücksichtigt. Beim Messwert *LoC* wurde zwischen Code-Zeilen und Code-Zeilen mit Kommentaren und Leerzeilen unterschieden. Beide gemessenen *LoC*-Werte umfassen sämtliche vom C#-Designer generierte Code-Zeilen.

Name der Kennzahl	gemessener Wert
Benötigter Speicherplatz des Quellcodes	1,97 MB
Benötigter Speicherplatz der .exe und .dll-Dateien	1,89 MB
LoC (nur Codezeilen)	ca. 42.600
LoC mit Kommentaren, Leerzeilen	ca. 49.700

Tabelle 7.1: Benötigter Speicherplatz und LoC von Plux-CRM

7.2 Plux.NET-spezifische Daten

Plux-CRM verfügt neben den allgemeinen Daten über eine Reihe von Plux.NET-spezifischen Daten. Die Plux.NET-spezifischen Daten reichen von der Anzahl der implementierten .NET-Projekte, über die Anzahl der zur Verfügung stehenden Klassen und Schnittstellen, bis hin zu der Anzahl der Plugins, Komponenten, Steckplätze und Stecker. Dabei ist folgende Unterscheidung wichtig: Die Anzahl der in Plux-CRM verwendeten Komponenten, Steckplätze und Stecker kann nur zur Laufzeit von Plux-CRM gemessen werden. Die Anzahl der in Plux-CRM zur Verfügung stehender Plugins und Komponenten kann auch dann ermittelt werden, wenn Plux-CRM nicht gestartet ist, da es sich bei einem Plugin um ein .NET-Projekt und bei einer Komponente um eine Klasse handelt.

Tabelle 7.2 listet die Anzahl der in Plux-CRM implementierten .NET-Projekte, Klassen und Schnittstellen auf. Sämtliche .NET-Projekte, Klassen und Schnittstellen des Plux.NET-Frameworks werden in dieser Auflistung nicht berücksichtigt. Die Werte beziehen sich rein auf Plux-CRM.

	Plux-CRM
.NET-Projekte	19
Klassen	ca. 550
Schnittstellen	ca. 150

Tabelle 7.2: Anzahl der .NET-Projekte, Klassen und Schnittstellen

Die 19 .NET-Projekte von Plux-CRM sind in 18 Plux.NET-Plugins und in einen Plux.NET-Contract einzuteilen (siehe Tabelle 7.3). Nicht alle in Plux-CRM implementierten Klassen sind auch als Komponenten zu behandeln. Das Datenmodell im Plugin *CRM.DataModel* enthält beispielsweise einige Klassen, die keine Komponenten darstellen. Tabelle 7.3 zeigt zusätzlich die Anzahl der in Plux-CRM zur Verfügung stehenden Komponenten (Extensions).

	Plux-CRM
Plugins	18
Contracts	1
Extensions	335

Tabelle 7.3: Anzahl der Plugins, Contracts und Extensions

7.2.1 Vergleich von Plux.NET-spezifischen Daten zu bestimmten Zeitpunkten

Damit man ein Gefühl bekommt, wie unterschiedlich die Anzahl der verwendeten Komponenten, Steckplätze und Stecker zu gewissen Zeitpunkten in Plux-CRM sein kann, wurden zwei bestimmte Szenarien ausgewählt und miteinander verglichen.

Bei Szenario 1 handelt es sich um einen Zeitpunkt, bei dem eine geringe Anzahl an Komponenten, Steckern und Steckplätzen zum Einsatz kommt. Dieses Szenario stellt das Anmeldefenster von Plux-CRM dar. Das Anmeldefenster verfügt über zwei Eingabefelder für den Benutzernamen und das Passwort, einer Schaltfläche zum Anmelden und zwei Links, einer zum Rücksetzen des Passwortes und einer zum Beenden der Anwendung (Abbildung 7.1).

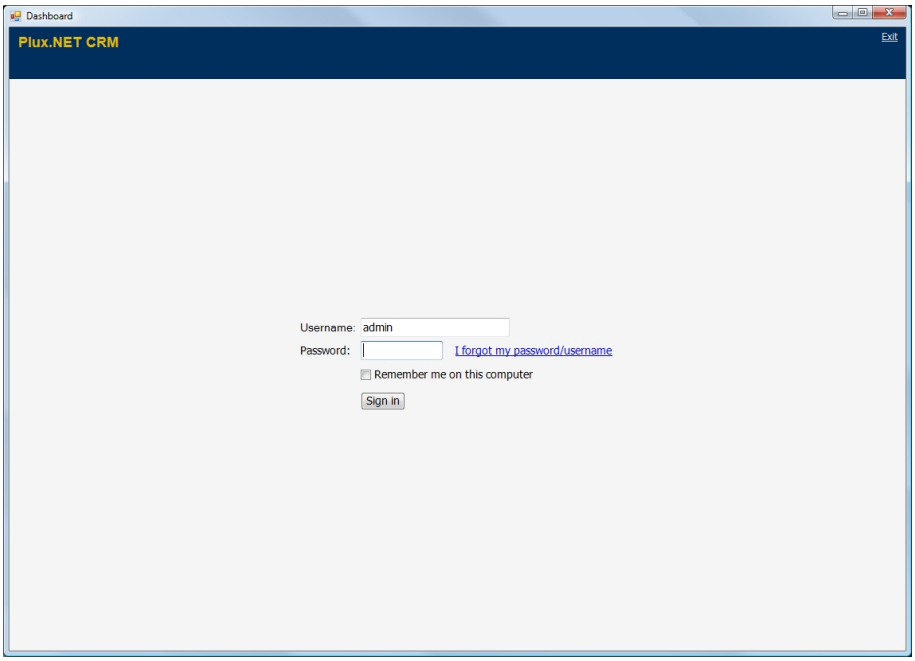

Abbildung 7.1: Szenario 1 in Plux-CRM

Abbildung 7.2 zeigt den umgekehrten Fall. Hier wurde ein Zeitpunkt gewählt, an dem eine hohe Anzahl an Komponenten, Steckern und Steckplätzen in Plux-CRM vorkommt. Dieses Szenario zeigt das „Schwarze Brett“, bei dem auf der linken Seite 10 Einträge zu sehen sind. Vier dieser Einträge beziehen sich auf das Abschließen neuer Geschäfte und das Ändern eines Geschäftsstatus. Zwei Einträge zeigen erledigte Aufgaben und vier Einträge zeigen das Verfassen neuer Notizen. Auf der rechten Seite des „Schwarzen Bretts“ befinden sich neun geplante Aufgaben des angemeldeten Benutzers. Davon sind vier Aufgaben bereits überfällig („Overdue“), zwei Aufgaben dem heutigen Tag zugeordnet („Today“) und drei Aufgaben nächste Woche zu erledigen („Next week“).

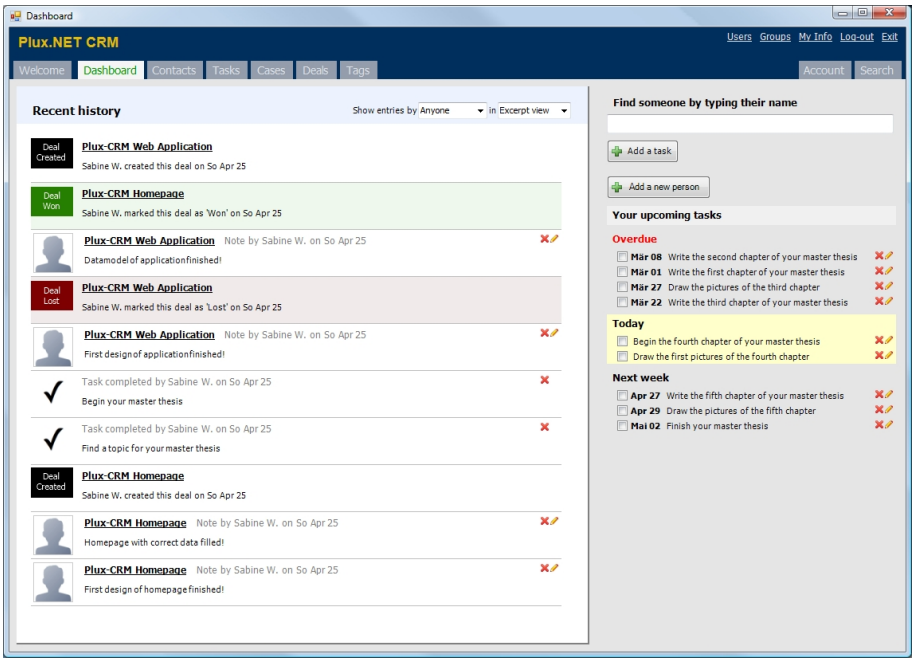

Abbildung 7.2: Szenario 2 in Plux-CRM

Betrachtet man nun die Werte in Tabelle 7.4, bei der die Anzahl der verwendeten Komponenten (Extensions), Steckplätzen (Slots) und Steckern (Plugs) zu den beiden Zeitpunkten dargestellt werden, erkennt man deutliche Unterschiede.

Die Werte in Tabelle 7.4 beziehen sich auf die in Plux-CRM implementierten Komponenten zusammen mit der benötigten Kern-Komponente.

	Szenario 1	Szenario 2
Verwendete Extensions	4	ca. 130
Verwendete Slots	10	ca. 110
Verwendete Plugs	3	ca 120

Tabelle 7.4: Plux.NET-spezifische Daten der Plux-CRM Komponenten zu bestimmten Zeitpunkten

8 Zusammenfassung

Das Ergebnis dieser Diplomarbeit ist das Kundenbeziehungsmanagementsystem Plux-CRM, das zeigt, wie Anwendungen basierend auf dem Plux.NET-Framework zu implementieren sind. Der Quellcode von Plux-CRM steht für Entwickler zur Verfügung, die die Plux.NET-Programmierung erlernen möchten. Es dient als Referenzimplementierung für Plux.NET-Programme. Plux-CRM weist somit eine klare Struktur und Lesbarkeit des Quellcodes auf. Zusätzlich zeigt Plux-CRM die Vor- und Nachteile auf, die die Verwendung von Plux.NET in Plux-CRM aufgrund seiner komponentenbasierten Architektur mit sich bringt.

Alle wesentlichen Bestandteile des Datenmodells und der Benutzeroberfläche von Plux-CRM bestehen aus Komponenten, die austauschbar sind. Aufgrund der flexiblen Benutzeroberfläche von Plux-CRM kann der Benutzer individuelle Konfigurationen an der Benutzeroberfläche vornehmen. Diese Konfigurationen umfassen das Anzeigen von neuen und das Entfernen von alten Ansichten. Vorhandene Komponenten für die Benutzeroberfläche können jederzeit durch neue Komponenten ersetzt werden. Komponenten können entweder durch andere Komponenten mit gleicher Funktion und anderem Aussehen, oder durch Komponenten mit unterschiedlicher Funktion ersetzt werden. Mit Plux-CRM ist es möglich, Komponenten mit neuen Funktionen oder Benutzersteuerelementen von Drittanbietern in die Anwendung aufzunehmen.

In Plux-CRM können Komponenten hinzugefügt, entfernt oder ersetzt werden, ohne das Programm neu zu starten. Um Plux-CRM auszuführen, müssen keine externen Programme oder Biblitoheken installiert werden. Nur das .NET-Framework ist für die Verwendung von Plux-CRM zu installieren.

Literaturverzeichnis

[1] Wolfinger, R., Reiter, S., Dhungana, D., Grünbacher, P., and Prähofer, H.: Supporting Runtime System Adaptation through Product Line Engineering and Plug-in Techniques. 7th IEEE International Conference on Composition-Based Software Systems, ICCBSS 2008, Madrid, Spain, February, 25-29, 2008.

[2] Christian Doppler Laboratory for Automated Software Engineering: Plux.NET: A Platform for Building Plug-in Systems Under .NET. URL: `<http://ase.jku.at/plux/index.html>` (Oktober 2009).

[3] Wolfinger, R.: Dynamic Application Composition with Plux.NET. Dissertation, Institute for System Softwarem Johannes Kepler University, Linz, Austria, January 2010.

[4] Christian Doppler Laboratory for Automated Software Engineering: Plux.NET: Logger Tutorial. URL: `<http://ase.jku.at/plux/Logger.html>` (Oktober 2009).

[5] Sells, C., Weinhardt, M.: Windows Forms 2.0 Programming. Microsoft .NET Development Series, Addison-Wesley Professional, 2. Edition, 2006.

[6] Jahn, M.: Entwurf und Implementierung eines Cross-Compilers von Delphi nach C#. Master thesis, Institute for System Software, Johannes Kepler University, Linz, Austria, March 2009.

[7] Wolfinger, R.: Plug-in Architecture and Design Guidelines for Customizable Enterprise Applications, OOPSLA 2008 Doctoral Symposium, OOPSLA 2008, Nashville, Tennessee, October, 19-23, 2008.

[8] Wolfinger, R., Dhungana, D., Prähofer, H., Mössenböck, H.: A Component Plug-in Architecture for the .NET Platform. Modular Programming Languages, Lightfoot, David; Szyperski, Clemens (Eds.), Lecture Notes in Computer Science , Vol. 4228, Proceedings of 7th Joint Modular Languages Conference, JMLC 2006, Oxford, UK, September 13-15, 2006.

[9] Wolfinger, R., Prähofer, H.: Integration Models in a .NET Plug-in Framework. SE 2007 - the Conference on Software Engineering, Hamburg, Germany, March 27-30, 2007.

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Lebenslauf

Persönliche Daten

Name:	Weiss
Vorname:	Sabine
Akademischer Grad:	Bakk. techn.
Anschrift:	Hydenstraße 16 4600 Wels
Geburtstag:	19. Oktober 1985
Geburtsort:	Hilden, Deutschland
Staatsangehörigkeit:	Österreich

Bildungsweg

Sept. 1992 - Juli 1996:	Volksschule an der VS 8 in Wels
Sept. 1996 - Juni 2004:	Bundesrealgymnasium Dr. Schauerstraße mit Schwerpunkt Informatik in Wels, Matura Abschluss im Juni 2004
Sept. 2004 - April 2008:	Bachelorstudium Informatik an der Johannes Kepler Universität in Linz, Abschluss mit dem akademischen Grad „Bakkalaurea der technischen Wissenschaften“ im April 2008
seit April 2008:	Masterstudium Software Engineering an der Johannes Kepler Universität in Linz

Karriere

seit September 2009:	Mitarbeiterin der Firma Programmierfabrik GmbH in Linz

Eidesstattliche Erklärung

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

Linz, am 2. Mai 2010

Sabine Weiss