DIPLOMARBEIT

Portierung und plattformübergreifende Reimplementierung von Ärztesoftware

ausgeführt am

Institut für Softwaretechnik und interaktive Systeme
Information & Software Engineering Group
der Technischen Universität Wien

unter der Anleitung von
ao.Univ.Prof. Dr. Gerald Futschek
und
Univ.Ass. Dr. Alexander Schatten
als verantwortlich mitwirkendem Assistenten

durch

Christian Mader
Matr.Nr. 9725449
Grenzgasse 27, 3950 Gmünd

Wien, am 14. Oktober 2005

_____

Abstract

This thesis deals with porting a commercial application from Microsoft Windows to the GNU/Linux platform. Subject of the port is a medical application developed at the company CSP[4]. It assists doctors and health care personnel in performing daily tasks like registering patients, prescribing medication, filling out vouchers an much more. The product's name is *CSPmed* and it is split into a client and a server part. The user interface of the server application is text-based and looks rather outdated, so in early 1998 the Microsoft Windows client (*CSPeasymed* aka *CspWin*) was developed to provide a modern interface supporting windows, buttons and all the stuff an average-skilled computer user won't like to miss. Basically the client task is to turn the text-based output of the server application into pretty Windows-widgets on a computer connected via LAN to the server. (Re)Implementing this at the GNU/Linux platform will be the main focus of this thesis, other client-functionality which makes the project even more complex has to be ported separately. The server is a native Unix application which already runs under SCO Unix or GNU/Linux and communicates with the Microsoft Windows-based client via Remote Procedure Call (RPC). The server software will not and must not be modified because these applications are well tested and work in a productive environment at our customers. The client on the other hand is subject of extensive change during reimplementation and aims towards complete substitution of the so far used Microsoft Windows client.

Danksagung

An dieser Stelle ist es mir ein Anliegen all jenen Personen Dank zu sagen ohne die die vorliegende Diplomarbeit wohl nicht entstehen hätte können. Meinen Eltern danke ich nicht nur für die finanziellen Zuwendungen während der Zeit meines Studiums sondern auch für ihre moralische Unterstützung. Großer Dank gebührt auch allen beteiligten Institutsmitarbeitern, insbesondere Herrn Alexander Schatten für seinen Einsatz und seine Zeit sowie Herrn Max Helletsgruber und Dietmar Spandl von der Firma CSP, die es mir ermöglichten diese Arbeit zu verfassen.

Nicht zuletzt danke ich all meinen Freunden und Studienkollegen mit denen ich in den letzten Jahren viel Zeit zusammen verbringen durfte für ihre Unterstützung und Motivation.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Algorithmenverzeichnis

1 Begriffsdefinitionen

Das gesamte Softwarepaket, also Client und Server sowie diverse Zusatzapplikationen wie in 2 beschrieben, wird im folgenden Text als *CSPmed* bezeichnet. Der im Zuge dieser Arbeit zu implementierende Client wird fortan *CSPmedclient* beziehungsweise *CSPmed Client*, analog dazu der Server *CSPmed Server* genannt. Beide sind oft auch nur als „Client" oder „Server" bezeichnet.
Der Begriff „freie Software" bezieht sich in dieser Arbeit nicht auf den Preis einer Software sondern darauf, dass sie unter einer offenen Lizenz (etwa einer der in 5.1.3 behandelten) erhältlich ist, die unter Anderem die Verbreitung des Quellcodes miteinschließt, der somit für jedermann einsehbar ist. Richard Stallman ([13]) prägte in bezug auf freie Software den Ausspruch

> „Free as in freedom, not as in free beer"

um die Natur derartiger Software auszudrücken. Eine Definition freier Software ist unter [15] zu finden.
Ist in dieser Arbeit von einem Rechner die Rede, auf dem der Linux Systemkern (Linux Kernel, oftmals auch bloß als „Linux" bezeichnet) unterstützt von freier Software eingesetzt wird, wird dies konsequenterweise als *GNU/Linux System* bezeichnet, da die grundlegenden Tools, die gemeinsam mit Linux das Betriebssystem bilden, aus dem GNU Projekt [2] stammen.
Elemente einer Benutzerschnittstelle, die auf Eingaben des Benutzers reagieren, wie zum Beispiel Schaltflächen (Buttons) oder Textfelder werden im Folgenden auch als *Widgets* bezeichnet.
Als *Plattform* wird in diesem Dokument die Kombination von Betriebssystem und einem bestimmten Computertyp oder einem Computer einer bestimmten Architektur bezeichnet. *Plattformunabhängig* beziehungsweise *plattformübergreifend* ist eine Applikation dann einsetzbar, wenn sie sich auf mindestens zwei Plattformen deren Betriebssystem unterschiedlich ist, übersetzen und ausführen lässt.

2 Übersicht

Das Ziel der Portierung ist es, die grafische Oberfläche des CSPmedclients größtenteils plattformunabhängig auszuführen. Größtenteils deshalb, weil heutzutage eine unüberschaubare Anzahl an Plattformen exisitert und es mit vertretbarem Aufwand so gut wie unmöglich ist wirklich jede dieser Plattformen zu unterstützen. Das Programmpaket CSPmed besteht aus einem Client und einem Server. Der Client erhält vom Server per RPC (Remote Procedure Call) Kommandos, die ihn dazu veranlassen, entsprechende Menüs, Eingabefelder, Schaltflächen und andere Interaktionselemente darzustellen. Im Prinzip soll die Oberfläche des Programms am Server (die dort Textmodus-basiert ist) auf eine moderne Benutzerschnittstelle abgebildet werden. Zur Zeit funktioniert das jedoch nur mit dem CSPmedclient für Microsoft Windows (auch *CspWin* genannt), der, wie oben schon erwähnt, nun so angepasst

beziehungsweise reimplementiert werden soll dass er unter verschiedenen Plattformen lauffähig ist.
Das CSPmed Programmpaket besteht auch aus mehreren „externen" Applikationen, die zur Zeit als Microsoft Windows-Programme implementiert sind. Hierzu zählen zum Beispiel das CSP Grafikprogramm oder eine Terminverwaltung. Diese Applikationen sollen im Rahmen dieser Arbeit nicht portiert beziehungsweise reimplementiert werden. Weiters existieren auch Schnittstellen zu Software, die nur auf der Microsoft Windows Plattform verfügbar ist, wie zum Beispiel zu Microsoft Word, Wordpad oder zur Spracherkennung. Die Implementierung der Funktionalität dieser Schnittstellen soll ebenfalls nicht Gegenstand dieser Arbeit sein, es sollte jedoch in vielen Fällen möglich sein, dies ohne großen Aufwand an Personentagen nachzuholen.
Wichtig ist, dass bestehender Code von Serverprogrammen ohne Änderungen lauffähig bleiben muss, alleine der Client darf verändert werden. Ebenso muss natürlich auch das Netzwerkprotokoll und die RPC Schnittstelle gleich bleiben.
Die Portierung des CSPmedclients, die im Zuge dieser Arbeit erstellt wird, hat das Ziel zu zeigen, dass ein solches Unterfangen prinzipiell möglich ist und soll bereits eine Untermenge all jener Arbeitsschritte, die auch direkt am Server (beziehungsweise per Telnet oder SSH Verbindung dorthin) ausgeführt werden können, unterstützen. Dabei sollen grundlegende Funktionen, wie etwa die Darstellung und (eingeschränkte) Benutzung einiger Stammdatenprogramme beziehungsweise der Ordinations- und Anmeldungsprogramme zur Verfügung stehen und somit eine Ausgangsbasis für weitere Entwicklungen und Verbesserungen geschaffen werden. Details zum Umfang der Implementierung sind in 4 zu finden, da es zu umfangreich wäre, hier alle Funktionen anzuführen, die portiert werden sollen.
Die fertige Applikation soll sowohl unter GNU/Linux als auch unter Microsoft Windows kompilierbar und lauffähig sein, wobei besonderes Augenmerk darauf liegt, die Versionen für beide Betriebssysteme aus ein- und demselben Quellcode erzeugen zu können, um den Wartungsaufwand in Grenzen zu halten. Durch die Lauffähigkeit des CSPmedclients unter GNU/Linux wird bereits ein Großteil der verfügbaren Plattformen abgedeckt, sodass auf diese Portierung in der vorliegenden Arbeit das Hauptaugenmerk gelegt wird.

3 Ausgangssituation

3.1 Softwarebasis

3.1.1 Server

Aktuell wird für den Betrieb des CSPmed Systems serverseitig bereits ein Unix-Betriebssystem verwendet. Hierbei handelt es sich entweder um SCO-Unix oder SuSE Linux, wobei bei Neuinstallationen nur mehr SuSE Linux eingesetzt wird. Datenbankserver ist eine Oracle Datenbank in der Version 5 oder 8.

Die Applikationen werden in Standard-C implementiert, es existiert eine eigene Bibliothek *CSPclib*, die oft benötigte Funktionalität wie zum Beispiel verkettete Listen und sichere Kopierfunktionen, die ein unbeabsichtigtes Überschreiben von Speicherbereichen verhindern sollen, zur Verfügung stellt. Weiters wird die Programmentwicklung am Server durch das *Window Manangement System (WMS)* unterstützt. Dies ist ebenso wie die CSPclib eine Sammlung von Funktionen, die Eigentum der Firma CSP sind. Die WMS-Bibliothek stellt Methoden zur Implementierung eines Benutzerinterfaces zur Verfügung, also zum Beispiel das Einlesen von Bildschirmmasken oder die Ausgabe von Listboxen oder Hinweisfenstern. Diese Elemente werden dann je nach Aufruf der Applikation in einem Textterminal-Fenster dargestellt oder mit Hilfe des CSPmed Clients in Form einer grafischen Oberfläche am jeweiligen Client-Rechner.
Serverseitig werden zur Entwicklung Tools eingesetzt, die zum WMS gehören und bereits für Unix und GNU/Linux verfügbar sind. Hierzu zählen zum Beispiel der Formcompiler und der Listcompiler. Ersterer wird verwendet, um aus den Beschreibungsdateien für Eingabemasken entsprechende Headerdateien und Binärdateien zu erzeugen, die in den entsprechenden C-Code eingefügt werden. Mit dem Listcompiler verhält es sich ähnlich, nur dass dieser auf Beschreibungsdateien für Druckformulare angewandt wird, deren Syntax von der Formularbeschreibung leicht abweicht.
Die Client-Server Kommunikation erfolgt mittels in C implementierten RPC Methoden, die in Form vom Bibliotheken für SCO Unix und Linux vorliegen. Gegen diese Bibliotheken werden die erstellten Applikationen am Server gelinkt, der Programmierer einer Serverapplikation braucht sich um die Netzwerkfunktionalität also keine Gedanken zu machen.
Es sei an dieser Stelle noch einmal deutlich erwähnt, dass sich der CSPmedclient ausschließlich um die grafische Umsetzung der Benutzeroberfläche der Serverapplikation an einem per Netzwerk verbundenen Windows-Rechner kümmert. Er enthält darüber hinausgehend keinerlei Funktionalität, die jene der Applikation die am Server läuft erweitert. Im Prinzip könnte man den CSPmed Client daher als eine Art Terminalemulation betrachten, siehe dazu auch die Abbildungen in 3.3. Die Binärdateien die am Server laufen, sind exakt die gleichen, egal ob die Ausgabe im Terminal erfolgt, oder mittels CSPmedclient. Da im Terminal Funktionen, wie etwa das Darstellen von Buttons mit Bildern als „Beschriftung" nicht möglich sind, ist es daher notwendig, innerhalb der Applikation abzufragen, ob der CSPmedclient oder das Terminal zur Ausgabe verwendet wird. Man kann dann bei der Implementierung der Oberfläche im C-Quellcode auf die jeweiligen Darstellungsmöglichkeiten Rücksicht nehmen. Von Seite der Firma CSP werden Neuentwicklungen schon seit einiger Zeit nur mehr für die Ausgabe mit Hilfe des CSPmedclients ausgelegt, die Darstellung der Oberfläche im Terminal wird nicht mehr unterstützt.

3.1.2 Client

Momentan wird als Client-Betriebssystem Microsoft Windows in den Versionen 98 und XP eingesetzt. Der CSPmedclient ist in der Programmiersprache C++ mit Hilfe von Microsoft Visual C++ implementiert. Dabei werden für die Ausgabe der Benutzerinterfaceelemente die *MFC* verwendet (*Microsoft Foundation Classes*). Hierbei handelt es sich um eine Vielzahl verschiedenster Klassen, deren Aufgabe vor allem die einfache Entwicklung von Benutzerschnittstellen unter Microsoft Betriebssystemen ist, die aber auch andere Funktionen, wie zum Beispiel Unterstützung bei der Implementierung von Netzwerkfunktionalität, zur Verfügung stellen. Wie die Abkürzung MFC schon nahelegt, sind dies Klassen Eigentum von Microsoft und stehen nicht für andere Betriebssysteme als Windows zur Verfügung. Sie liegen allerdings im Quellcode vor, was ein Kompilieren unter GNU/Linux theroretisch ermöglichen würde. Nähere Informationen zu diesem Thema sind in diesem Dokument unter 5.2.1 zu finden.
Die CSPmed Clientapplikation greift ebenfalls auf die in 3.1.1 genannten RPC Methoden zurück, diese liegen in Form von Microsoft Windows Bibliotheken vor, gegen die der CSPmedclient gelinkt ist.

3.1.3 Kommunikation des Clients mit dem Server

Die *libcsprpc* Bibliothek ist eine CSP-eigene Implementierung des RPC-Protokolls. Laut Dokumentation in den Quellcodedateien wurde sie 1995 begonnen zu implementieren, jedoch existieren keine Informationen mehr darüber, aus welchem Grund diese Bibliothek implementiert wurde und nicht einfach die in jedem Unix Betriebssystem bereits vorhandenen RPC-Bibliothken verwendet wurden. Für die Darstellung der Applikation am Terminal wird das sogenannte WMS (Window Management System) Toolkit verwendet, ebenfalls eine CSP Eigenentwicklung, die dazu dient, Programmoberflächen im Textmodus auf einem Terminal beziehungsweise in einer Terminalemulation anzuzeigen (siehe dazu auch 3.1.4). Dieses Toolkit verwendet die libcsprpc Bibliotheken zum Übertragen von Strukturen des in Algorithmus 1 dargestellten Typs, der in `/udsk_ent/csp/csp_sys/mask/pgm/CSPm_wwms.x` definiert ist.
Dabei kann z.B. `iCommand` ein Kommando zum Einlesen eines Formulars bezeichnen, dessen Parameter in den `sVal` und `iVal` Feldern angegeben sind. RPC wird also allem Anschein nach bloß zur Übergabe dieser Information benötigt. Daher sollte es unter Umständen auch möglich sein, die csprpclib Bibliothek mit der Standard-RPC Implementierung der jeweils verwendeten Betriebssysteme zu ersetzen, oder andere Wege zu finden, eine Datenstruktur über das Netzwerk auszutauschen. Natürlich müsste man dann das WMS-Toolkit entsprechend umändern, alle Serverprogramme die ja gegen die WMS Bibliotheken gelinkt sind, neu kompilieren und auch den entsprechenden Code im CSPmedclient anpassen. Ein Kriterium bei der Entwicklung der plattformunabhängigen Version des CSPmedclients war, dass er als Ersatz für die momentan verwendete MFC-Version fungiert.

Algorithm 1 Struktur zum Datenaustausch zwischen Client und Server

```
struct tag_wwmscmd
{
    TYP_CSP027CLASSTYPE eClass;
    int iCommand;
    int hForm;
    int hWindow;
    string sVal1<>;
    string sVal2<>;
    string sVal3<>;
    int iVal1;
    int iVal2;
    int iVal3;
};
```

Dies war jedoch nur dadurch zu bewerkstelligen, dass er mit dem gleichen RPC-Protokoll wie das aktuell verwendete kommuniziert und somit eine Verwendung der libcsprpc Bibliothek unvermeidlich. Um möglichst schnell zu einem Ergebnis zu gelangen, erschien es sinnvoll, den aktuell verwendeten Quellcode der MFC-Version des CSPmedclients, der die Kommunikation mit dem Server implementiert, anzupassen um ihn plattformunabhängig einzusetzen. Dies konnte auch in relativ kurzer Zeit erreicht werden, wodurch die Verwendung der C++ Programmiersprache als die am schnellsten eine Lösung versprechende Variante nahelag.
Ein möglicher anderer Weg wäre sicherlich auch Java gewesen, hierbei hätte man allerdings höchstwahrscheinlich eine Alternative zur libcsprpc Bibliothek implementieren müssen, mit den oben erwähnten Konsequenzen für das WMS und die Serverprogramme. Die Möglichkeit einer Einbindung der bestehenden libcsprpc Bibliothek in Java Programme müsste geprüft werden, jedoch ist der dafür notwendige Aufwand (falls es prinzipiell möglich wäre) sicher nicht geringer einzuschätzen als eine komplette Neuimplementierung des Kommunikationsprotokolls. Es sollte dennoch kein besonders hoher Aufwand sein, den CSPmedclient auf ein anderes Kommunikationsprotokoll umzustellen, als Fixpunkt der Implementierung kann die oben angegebene Struktur angesehen werden. Aufgrund ihres Inhalts, den der CSPmedclient über das Netzwerk empfängt, werden sämtliche Aktionen durchgeführt. Hat man also vor, das Protokoll umzustellen, ist es „nur” notwendig, diese Daten der entsprechenden verarbeitenden Klasse im CSPmedclient zur Verfügung zu stellen und alles sollte wie gewohnt ablaufen. Die Rückmeldung vom CSPmedclient an den Server erfolgt in ähnlicher Weise, und zwar als Liste oben angegebener Strukturen an den Server. Auf die weitere Funktionsweise der libcsprpc Bibliothek soll im Folgenden nicht mehr näher eingegangen werden, außer es ist für das Verständnis der Funktion des CSPmedclients unbedingt erforderlich. Die libcsprpc Bibliothek soll als eine Art „Black Box" angenommen werden, die fortlaufend Datenstrukturen des Typs `tag_wwmscmd` liefert.

Algorithm 2 Beschreibungsdatei eines Fensters

```
.s t=09 b=15 l=23 r=61
.a H
```

3.1.4 Erstellung einer Benutzerschnittstelle

Dieser Abschnitt geht auf die einzelnen Schritte zur Erstellung einer Benutzerschnittstelle mit Hilfe des WMS und des CSPmedclients ein. Das WMS (Window Management System) ist eine von der Firma CSP geschriebene Applikation mit deren Hilfe es möglich ist, eine Terminalapplikation in optisch ansprechender Form darzustellen. Damit ist zum Beispiel eine optische Anzeige der verfügbaren Funktionstasten gemeint, oder die Anzeige von Eingabefeldern in den entsprechenden Eingabemasken, welche umrahmt und teilweise mit Titelzeile dargestellt werden können. Die Ausgabe kann in einem Terminal beziehungsweise einer Terminalemulation erfolgen, aber auch an einen beliebigen anderen Rechner im Netzwerk per RPC weitergeleitet werden, auf dem der grafische CSPmedclient läuft. Näheres dazu und zum verwendeten Protokoll wurde bereits in 3.1.3 angeführt.

Definition der Fenster und Formulare

Da das WMS ursprünglich nur für eine Darstellung von Eingabemasken im Textmodus gedacht war, ist auch das dahinterliegende Konzept auf eine zeilen- und spaltenweise Angabe der Position der Maskenelemente ausgerichtet. Grundsätzlich wird im WMS zwischen einem Fenster und einem Formular (manchmal auch Maske genannt) unterschieden. Ein Fenster stellt demnach sozusagen den Rahmen eines Formulars dar, kann frei am Bildschirm positioniert werden und seine eigenen Funktionstastendefinitionen haben. Ein Formular ist einem Fenster zugeordnet und beinhaltet sämtliche Formularelemente wie Texteingabefelder, Comboboxen, Checkboxen und Listviews. Das Layout von Fenstern und Formularen wird jeweils über eine eigene Textdatei festgelegt, in Algorithmus 2 sind die wesentlichen Eintragungen in Beschreibungsdatei eines Fensters ersichtlich.
Dabei bezeichnen die Werte in der ersten Zeile die Position des Fensters in Zeilen und Spalten (ein Standard-Terminalfenster hat 25 Zeilen und 80 Spalten), also oberer Rand (Top) in der 9. Zeile, unterer Rand (Bottom) in der 15.Zeile, linker Rand (Left) in der 23. Spalte und rechter Rand (Right) in der 61. Spalte.
Die zweite Zeile ist das Videoattribut (`.a`) des Rahmens, das nur für die Darstellung im Terminal von Bedeutung ist, mögliche Werte wären hier `H` für hohe Intensität, `N` für normale Intensität oder `U` für Unterstrichen. Die Layoutdefinition eines Formulares erfolgt in einer Textdatei mit ähnlichem Aufbau, dargestellt in Algorithmus 3.
Mittels `.f` wird in der ersten Zeile der Datei ein sogenanntes Fixtextfeld definiert. Dies ist ein Feld, dessen Inhalt schon bei der Erstellung des Formulares feststeht und im Programm nicht mehr verändert werden kann. Solche Felder sind zum Beispiel zur Beschriftung von Texteingabefeldern sinnvoll. Die Parameter `r=` und `c=`

Algorithm 3 Beschreibungsdatei eines Formulares

```
.f r=00 c=01 a=N p=Bitte w{hlen Sie den Kartenleser
aus:
.F r=000 c=010
.v r=01 c=01 n=list p=X(1)
.V r=010 c=010 typ=6
.v r=03 c=01 n=merk p=X(1)
.V r=050 c=010 typ=7 size=150
```

legen die Zeile (row) und Spalte (column) fest, in der sich das Fixtextfeld im Formular befinden soll. Hier ist es ebenfalls wieder möglich, ein Videoattribut anzugeben und natürlich auch den Text (mittels `p=`), der auf dem Formular erscheinen soll.
Die Koordinaten, die mittels `.F` in der zweiten Zeile angegeben sind, beschreiben die Position des Fixtextfeldes im CSPmedclient. Da dieser Client auf eine grafische Oberfläche zurückgreifen kann, ist es möglich, die Formularelemente genauer zu positionieren, es stehen bei der gegenwärtigen Implementierung 10 zusätzliche Abstufungen zur Positionsangabe im Vergleich zum Terminal zur Verfügung. Variable Felder werden auf die gleiche Art wie Fixtextfelder positioniert, im Gegensatz dazu aber mit `.v` beziehungsweise `.V` definiert. Ein variables Feld kann sowohl ein Feld zur Eingabe von Text sein (der häufigste Fall), aber auch eine Checkbox oder eine Combobox sein. Das konkrete Aussehen legt der `typ=` Parameter fest, im vorliegenden Fall ist dieser 6, also ein Listview Control. Weitere mögliche Werte wären 5 für eine Combobox oder 7 für eine Checkbox.
Zusätzlich müssen im Quellcode der Serverapplikation nach dem Erzeugen des Formulars noch die endgültige Position und zusätzliche Parameter von Check- und Comboboxen angegeben werden, Details dazu später. Der Parameter `p=` legt bei variablen Feldern zum Einen fest, wieviele Zeichen das Feld aufnehmen soll (was ident mit der Länge am Terminalbildschirm ist, im vorliegenden Fall jeweils ein Zeichen) und zum Anderen von welchem Typ die Zeichen sind. Der Typ ist in diesem Fall `X`, also Buchstaben, Zahlen und Leerzeichen. Alternativ wäre es möglich, die Eingabe auf Zahlen zu beschränken (`p=9(1)`) oder mittels `p=Y` ein Datumsfeld zu definieren. Ein solches Feld würde ein Datum in der Form DD.MM.JJJJ aufnehmen und bereits bei der Eingabe sowohl im Terminal als auch im CSPmedclient eine Kontrolle der Benutzereingabe vornehmen. Wird also ein unmögliches Datum angegeben, würde eine entsprechende Meldung automatisch ausgegeben werden, ohne dass man bei der Programmierung der Serverapplikation darauf Rücksicht nehmen müsste. Mit Hilfe des Parameters `size=` ist es möglich, die tatsächliche Größe eines Feldes bei der Darstellung durch den CSPmedclient anzugeben. Da das GUI-System mit proportionalen Fonts arbeitet, die Berechnung der Größe eines Controls aber aufgrund der Anzahl der Zeichen erfolgt, kann es zu Überschneidungen kommen, die man mit diesem Parameter beseitigen kann.
Sind die Textdateien der Fenster und Formulare fertig definiert, müssen die Fenster

mit dem Windowcompiler und die Formulare mit dem Formularcompiler übersetzt werden. Die Compiler übersetzen die Textdateien in ein kompaktes binäres Format, um einen wesentlich schnelleren Zugriff auf die Maskenbeschreibung gewährleisten zu können. Außerdem werden syntaktische Fehler in den Beschreibungsdateien erkannt und dem Benutzer gemeldet. Zum Schreiben und Lesen des Formulars im Serverprogramm wird vom Formcompiler eine Includedatei mit einer Struktur die alle Formularfelder als character-Array enthält, erzeugt.

Grundlegende UI-Funktionsaufrufe im Serverprogramm

Beim ersten Aufruf des Serverprogramms muss das Fenster in den Hauptspeicher geladen werden. Das passiert mittels

```
Aw838_win = Ldwin("Aw838");
```

Hierbei ist `Aw838_win` der von Ldwin zurückgegebene eindeutige Windowdescriptor und `Aw838` die vom Windowcompiler erzeugte Binärdatei des Fensters. Das Formular wird folgendermaßen in den Hauptspeicher geladen:

```
Al838_msk = Ldform("Al838", Aw838_win);
```

Wiederum wird ein Formdescriptor zurückgegeben und das Formular gleich dem vorher erzeugten Fenster zugeordnet. Der Aufruf von

```
Ptform(Al838_msk, 0, 0);
```

schreibt die Fixtextfelder des Formulars auf den Bildschirm, und zwar realtiv zum linken oberen Rand des beim `Ldform` zugeordneten Windows (in diesem Fall sind beide Koordinaten 0). Mittels

```
WxOpenwinModal(Aw838_win, "Auswahl");
```

kann man dann auch schon das Fenster am CSPmedclient darstellen, und zwar als modales Fenster mit dem Text `Auswahl` in der Titelzeile. Dies funktioniert allerdings nur bei der Darstellung durch den CSPmedclient, am Terminal müsste man folgende Funktion verwenden:

```
Openwin(Aw838_win);
```

Eingelesen wird ein Formular mit Hilfe des folgenden beispielhaften Funktionsaufrufes:

```
Rdform(Al838_msk, "*", &Al838, 0);
```

Der Parameter `*` bedeutet dabei, dass alle Felder eingelesen werden sollen, der Eingabecursor also in jedes Feld springen soll. Alternativ ist es möglich, nur definierte Felder anzuspringen, etwa wenn man statt `"*"` den String `"feld1, cbox"` angibt. Dann ist es nur möglich, in die in der Formularbeschreibung als `n=feld1` und `n=cbox` definierten Felder Werte eintragen. Der dritte Parameter der `Rdform()` Funktion gibt die Struktur an, die vom Formularcompiler erzeugt wurde und enthält die vom Benutzer in das Formular eingegebenen Daten. Rückgabewerte der `Rdform()` Funktion sind zum Beispiel die Integerwerte `WMS_QUIT`, falls die Escape Taste vom Benutzer gedrückt wurde, und die Eingabe abgebrochen

werden soll. Bei Druck der Abschlusstaste (F12) wird `WMS_END` von `Rdform()` zurückgegeben. Will man umgekehrt Werte von der Serverapplikation in das Formular schreiben, füllt man den entsprechenden Datenpuffer des Formulars (in unserem Beispiel die Struktur `Al838`) und ruft die Funktion `Wrform()` auf. Um also etwa die Checkbox die mit `n=merk` in der Formularbeschreibungsdatei definiert ist zu setzen, muss man in diesem Fall mittels

```
Al838.merk[0] = 'X';
```

den Wert setzen und danach

```
Wrform(Al838_msk, "merk", &Al838);
```

ausführen. Dadurch wird das Feld `merk` am Bildschirm aktualisiert und dementsprechend ein Häkchen in die Checkbox gesetzt. Auch hier wäre es wieder möglich, statt einem Feld alle Felder mittels `*` als zweiten Parameter der Funktion `Wrform()` anzugeben.
Die oben beschriebenen Aufrufe der Methoden funktionieren sowohl im Terminal als auch unter Benutzung des CSPmedclients, man kann ein- und denselben Quellcode also dazu verwenden, um Ausgaben im Textmodus als auch in der grafischen Benutzeroberfläche zu realisieren. Falls man im Programm ermitteln möchte, ob die Applikation im CSPmedclient oder im Terminal gestartet wurde, kann man die Methode `WxIsGUI()` verwenden, die entsprechend `TRUE` oder `FALSE` zurückgibt, falls die grafische Benutzeroberfläche, also der CSPmedclient, für die Darstellung dieser Applikation verantwortlich ist. Letztgenannte Methode ist auch recht hilfreich, falls für eine Aufgabe im CSPmedclient und im Terminal zwei verschiedene Methoden zur Verfügung stehen, wovon eine auf die Ausgabe über den CSPmedclient optimiert ist.
Eine moderne Applikation bietet jedoch nicht bloß einfache Texteingebefelder zur Benutzerinteraktion an, wie das im Terminal der Fall ist. Hier möchte man Eingaben auch komfortabel in Comboboxen, Checkboxen oder Listviews vornehmen, daher war es notwendig, zusätzlich zu den bestehenden WMS-Methodenaufrufen auch solche einzuführen, die die neuen Möglichkeiten des grafischen CSPmedclients ausnutzen können. Beispielhaft sei hier die Methode `WxComboBox()` erwähnt. Möchte man in seinem Formular eine Combobox darstellen, definiert man in der Formularbeschreibungsdatei ein variables Feld wie gewohnt mit Hilfe der Koordinaten. Im Serverprogramm platziert man jedoch noch zusätzlich den Aufruf von `WxComboBox()` vor dem ersten `Ptform()` Aufruf. Folgendes Beispiel macht aus einem „gewöhnlichen" Texteingabefeld im Terminal eine Combobox im CSPmedclient:

```
WxComboBox(Al25_n_msk, "art", "B Bereitschaftsdienst"
    "K Erkrankungsregelung"
    "U Urlaubsregelung "
    "E Erste Hilfe "
    "C E-Card ",
    "0,2", strlen("B Bereitschaftsdienst"), 0, 0 );
```

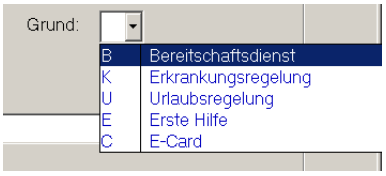

Abbildung 1: Combobox-Darstellung des MFC basierten CSPmedclients

Die genaue Bedeutung der Parameter kann man der WMS Dokumentation entnehmen, prinzipiell ist jedoch zu erkennen dass man wiederum das Feld angeben muss, das in eine Combobox „umgewandelt" werden muss (in diesem Fall das Feld `art`) sowie die Auswahlmöglichkeiten, die zur Verfügung stehen sollen. Der Screenshot in Abbildung 1 zeigt die solcherart definierte Combobox.
Ähnlich funktioniert die Darstellung einer Checkbox im CSPmedclient:

```
WxMsg(WMSM_DEFINE_CHECKBOX, Aw838_win, Al838_msk, 0,
    "merk","Kartenleser merken","X ",0,0);
```

Die `WxMsg()` Methode ist eine zentrale und relativ einheitliche Möglichkeit, mit dem CSPmedclient zu kommunizieren, sie wurde eingeführt, um die WMS-Library überschaubar zu halten und ohne großen Änderungsaufwand neue Funktionen über Messages implementieren zu können. Man kann mit ihrer Hilfe etwa auch Listviews erzeugen oder den Fokus manuell auf bestimmte Formularelemente setzen.

3.2 Hardwarebasis

Die von den Kunden eingesetzten Systeme werden ebenfalls von der Firma CSP geliefert und administriert. Für die Server kommt die Intel x86 Architektur zum Einsatz, die auch auf fast allen Clients verwendet wird. Einige Kunden verwenden als Clients immer noch Terminals, die nur zur Textdarstellung fähig sind. Aufgrund der Tatsache, dass Neuentwicklungen diese Terminals nicht mehr unterstützen, kommt es auch bei den Clientgeräten dieser Kunden zu einer langsamen Migration in Richtung x86 beziehungsweise deren 64-bit Erweiterungen.

3.3 Screenshots

Zur Verdeutlichung der obigen Beschreibung sind im Folgenden Abbildungen (2, 3 und 4) angeführt, die einen möglichen Arbeitszustand der Applikation zeigen.

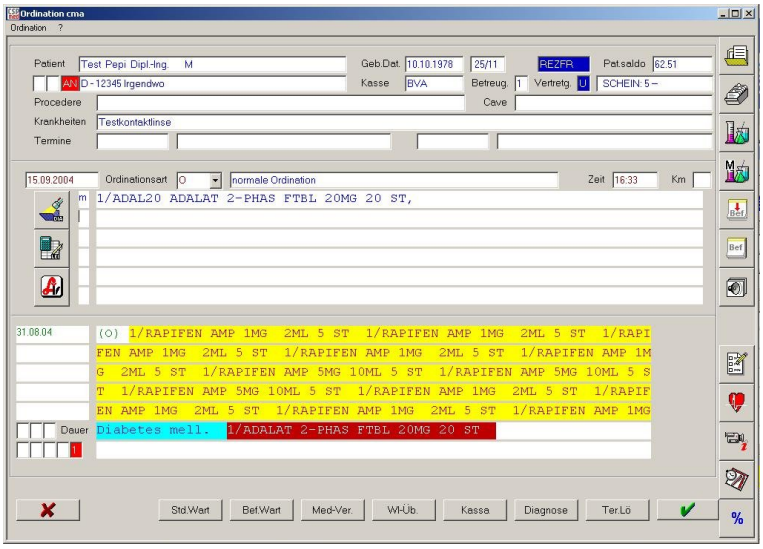

Abbildung 2: Der Standardbildschirm einer offenen Ordination mit einem eingegebenen Medikament

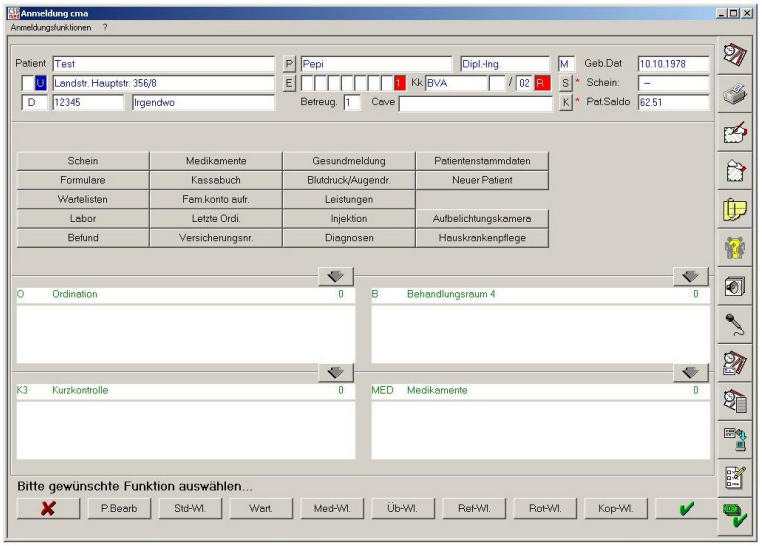

Abbildung 3: Startschirm der Anmeldung nach der Patientenidentifikation

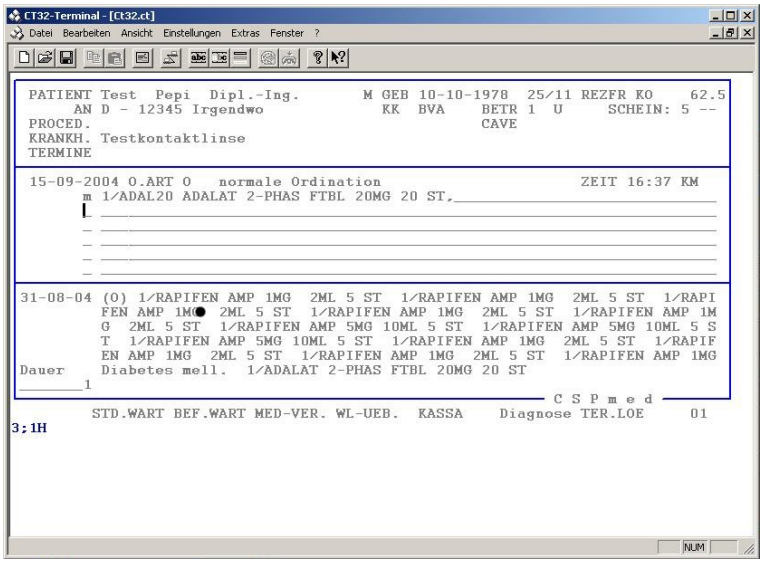

Abbildung 4: Die gleiche Ordination dargestellt am Server-Terminal

4 Zielsetzung

Teil der Anforderungen von Seite der Firma CSP war, den CSPmed Client möglichst plattformunabhängig zu implementieren. Das Ziel lag darin, sich von der Microsoft Plattform mit Betriebssystem Windows und Entwicklungswerkzeug Visual C++ loszulösen und auf Open Source Produkte umzusteigen. Der Vorteil dieser Strategie liegt darin, nicht von zukünftigen, höchstwahrscheinlich inkompatiblen Nachfolgeprodukten obengenannter Microsoft Plattform abhängig zu sein, die eine neuerliche Reimplementierung notwendig machen würden. Weiters ist natürlich das Wegfallen von Lizenzgebühren an Microsoft ein Argument für den Umstieg auf eine freie (Open Source) Plattform. Es sollte aber trotzdem weiterhin möglich sein, den CSPmed Client in einer Version lauffähig auf Microsoft Windows anzubieten, da die bestehende Infrastruktur bei den Kunden weiterhin nutzbar bleiben muss. Es wurde also dahingehend geplant, während der Entwicklung und Testphase des neuen plattformunabhängigen Clients weiterhin die stabile und erprobte Windows-Version einzusetzen und dann langsam, je nach Entwicklungsstand des neuen Clients, sukzessive die Kundensysteme umzustellen.
Das Ziel dieser Arbeit ist ein CSPmed Client, der unter GNU/Linux läuft und die in 4.1 angeführte Funktionalität zur Verfügung stellt. Dabei soll die Implementierung so erfolgen, dass man ohne großen Aufwand, also idealerweise durch einfaches Neukompilieren des Quellcodes, einen CSPmed Client für andere Betriebssysteme, insbesonders für Microsoft Windows, erzeugen kann. Es soll aber ausdrücklich erwähnt werden, dass die Erstellung der Windows-Version *nicht* Ziel dieser Arbeit ist, obwohl bei der Entwicklung darauf natürlich Rücksicht genommen wird.

Bewertung	Beschreibung
1	Kernfunktionalität, für eine lauffähige Portierung unverzichtbar
2	wichtig, um mit der Applikation produktiv arbeiten zu können
3	Zusatzfunktionalität für Spezialaufgaben
4	Zugaben, die keine Auswirkung auf die Funktionalität haben

Tabelle 1: Bewertungsschema

Folgende Anforderungen werden an den neuen CSPmed Client gestellt:

- lauffähig auf der GNU/Linux und Windows Plattform
- (weitestgehend) plattformunabhängige Implementierung
- zukunftssichere, einheitliche Codebasis
- bestmögliche Performance auch auf älteren Kundensystemen
- keine Änderungen an der Serverkonfiguration und -software erforderlich
- möglichst gleiches Benutzerinterface wie beim bisherigen CSPmed Client
- CSPmed Client ist Firmeneigentum und soll als Closed Source Software verkauft werden

Nachdem in 2 und 3 die Ausgangssituation dargelegt wurde und aufgrund obiger Auflistung die Ziele definiert sind ist es nun an der Zeit einen Weg zu finden wie man diese beiden Punkte verbinden kann. Um dies möglichst konsistent und nachvollziehbar zu gestalten wird eine Klasseneinteilung vorgenommen:

4.1 Zu implementierende Funktionalität

Die folgende Analyse der bestehenden Applikation umfasst die einzelnen Module, ihre Funktion und eine Bewertung die besagt, inwiefern diese Funktion substantiell für die Portierung der Applikation ist. Dabei wird das in Tabelle 1 dargestellte Schema zugrundegelegt.

In dieser Arbeit sollten die Stufen 1 und teilweise Stufe 2 implementiert werden, Stufe 3 würde aller Voraussicht nach den Rahmen überschreiten. Wichtig ist, ein System zu erstellen, das die in 4 angeführten Punkte erfüllt und eine Basis für weitergehende Entwicklungen darstellt. In Stufe 4 sind all jene Aufgaben und Funktionen zusammengefasst, deren Implementierung vielleicht zeitaufwändig, aber nicht unbedingt notwendig ist. Tabelle 2 setzt die identifizierte Hauptfunktionalität in Beziehung zum Bewertungsschema wobei die ganz rechte Spalte dokumentiert ob die jeweilige Funktion in dieser Arbeit implementiert wurde und lauffähig ist.

Funktion	Beschreibung	Bewertung	impl.
RPC-Kommunikation		1	ja
Anmeldung, Ordination	typische Arbeitsabläufe	2	nein
Stammdatenprogramme		1	ja
Login-Dialog	alle Optionen einstellbar	3	nein
	Grundfunktionalität	1	ja
Anzeige von Bedienelementen	Schaltflächen, Textboxen, Menüs usw.	1	ja
	farbliche Hervorhebung von Elementen	2	ja
	Splashscreen, "Eye-Candy"	4	nein
	Sidebar, Bitmapbuttons	2	ja
Funktionalität der Bedienelemente	Rückmeldung zum Server	1	ja
Ein-/Ausgabefunktionen	Druckeransteuerung	3	nein
	Ansteuerung medizinischer Geräte	3	nein
	Befundübertragung	3	nein
	Ansteuerung von Barcodescannern	3	nein
Zugriffsfunktionen	konfigurierbare Menüs	4	nein
	Aufruf einer externen Textverarbeitung	2	nein
	CSPmed Grafikprogramm	3	nein
	Terminsystem	3	nein
	Spracherkennung	3	nein

Tabelle 2: Bewertung der Komponenten

Diese Einteilung wurde so vorgenommen, um möglichst schnell zu einem in der Praxis einsatzfähigen System zu kommen. Die Grundfunktionalität muss auf jeden Fall vorhanden sein, ein Programm, das nicht mit dem Server kommunzieren kann und keine Interaktionsmöglichkeiten bietet, wird sicherlich sehr schwer zu verkaufen sein. Deshalb sind diese grundlegenden Eigenschaften als erstes zu implementieren. Weiters sollte die Übersichtlichkeit der Oberfläche sichergestellt werden, sodass es auch für ungeübte Benutzer einfach möglich ist, den CSPmed Client zu bedienen. Hierzu sind bestimmte Eigenschaften zur Strukturierung der Ausgaben vorhanden, wie zum Beispiel unter Anderem das farbige Hinterlegen oder Sperren von Eingabefeldern beziehungsweise blinkende Darstellung von Text. Diese Eigenschaften sind nicht besonders aufwändig zu implementieren, erhöhen die Benutzbarkeit aber ungemein und sollten deswegen auch in der Portierung auf GNU/Linux enthalten sein.
Manche Funktionalität des CSPmed Systems ist jedoch nur unter sehr hohem Zeitaufwand zu implementieren. Zum Beispiel sind auch externe Applikationen, wie das Grafiksystem oder die Terminreservierung im Angebot, die separat portiert werden müssten. Daher werden diese Applikationen vom Umfang dieser Arbeit ausgenommen, da viele Kunden schon mit dem Basissystem ihr Auslangen finden. Für viele kleine Arztpraxen ist der volle Funktionsumfang ohnehin nicht notwendig, für sie rechnet sich das grundlegende System ohne der Ansteuerung von medizinischen Geräten oder Spracherkennung. Letztgenannte Kunden sind es auch, für die vorerst der Einsatz des CSPmed GNU/Linux Clients aufgrund des Kostenvorteils am Interessantesten erscheinen wird, da die Lizenzgebühren für Microsoft Windows entfallen.
Im CSPmed Client ist auch Funktionalität vorhanden, die nur dazu dient, das Erscheinungsbild zu verschönern, ohne einen wirklichen Nutzen hinzuzufügen. Hierzu zählen vor allem hübsche Logos, sich ändernde Mauszeiger oder konfigurierbare Menüs. Dies ist zweifelsohne für eine zeitgemäße Applikation unerlässlich und die Implementierung nicht besonders schwierig, das Hauptaugenmerk dieser Arbeit liegt jedoch darin, einen robusten zuverlässigen Client zu erstellen, „Spielereien" können später noch hinzugefügt werden.

5 Implementierungsmöglichkeiten

Wie obenstehend bereits erwähnt, wurde der CSPmed Client durchgehend auf der Microsoft Windows Plattform entwickelt, eventuelle Portierungsvorhaben wurden bei der Implementierung nicht berücksichtigt. Grob gesehen kann man die Clientapplikation in zwei Teile trennen. Den ersten Teil stellt jene Funktionalität dar, die sich um den Aufbau und die Initialisierung der Netzwerkkommunikation mit der Serverapplikation kümmert. Dies erfolgt mittels der in 3.1.2 bereits angesprochenen RPC Funktionen. Der zweite und überaus umfangreichere Teil ist die Umsetzung der RPC Aufrufe in Methoden, die Benutzerinterface-Funktionalität zur Verfügung stellen. Wird also zum Beispiel vom Server das RPC Kommando zum

Anzeigen einer Messagebox gesendet, muss dies der Client korrekt interpretieren und im Falle des bisher verwendeten CSPmed Clients unter Microsoft Windows über MFC Aufrufe die Messagebox darstellen sowie mit dem vom Server übermittelten Text füllen.
Da die MFC Klassen nicht für GNU/Linux existieren und Microsoft in naher Zukunft unserer Einschätzug nach auch keine Anstrengungen unternehmen wird, dies zu tun, ist eine Reimplementierung des CSPmed Clients unumgänglich. Auch unter der Annahme, Microsoft würde seine Strategie gegenüber der Open Source Gemeinschaft ändern und eine Verwendung der MFC Klassen unter GNU/Linux ermöglichen, wäre hierbei wieder die Abhängigkeit von einer kommerziellen Firma gegeben, mit den bereits erwähnten Nachteilen wie Lizenzzahlungen und eventuelle Inkompatibilitäten bei neuen Produktversionen.
Aufgrund der Funktionsweise des CSPmed Systems wäre ein weiteres mögliches Konzept, die Interaktion mit der Applikation über dynamisch generierte Webseiten durchzuführen. Diese Implementierungsmöglichkeit unterscheidet sich grundsätzlich von den anderen hier vorgestellten Konzepten, die jeweils darauf beruhen eine eigene clientseitige Applikation zu erstellen, die ihre Eingabedaten per RPC über das lokale Netzwerk erhält.
Um die in 4 genannten Punkte zu erfüllen, bieten sich Möglichkeiten der Implementierung an, die jeweils unterschiedliche Vor- und Nachteile aufweisen und im Folgenden behandelt werden.

5.1 Überblick über die Technologiekonzepte

5.1.1 Benutzerschnittstelle

Je nach Art der Implementierung spielen bei der Technologie zur Realisierung einer Benutzerschnittstelle für eine plattformübergreifend einsetzbare Applikation unterschiedliche Faktoren eine Rolle.

Implementierung als eigenständige Applikation

Betrachtet man die aktuell verfügbaren Bibliotheken und Toolkits zur Programmierung pattformunabhängiger Applikationen, bemerkt man einige unterschiedliche Ansätze aber auch Gemeinsamkeiten. Vor allem wenn eine grafische Benutzerschnittstelle implementiert werden soll, die auf möglichst vielen Plattformen mit der gleichen Codebasis einsetzbar ist, sieht man sich mit zwei Alternativen konfrontiert:

1. Native Bibliotheken: Zur Darstellung von Widgets werden Routinen des Betreibssystems beziehungsweise der darunterliegenden grafischen Oberfläche verwendet. Dies hat zur Folge, dass das „Look and Feel” des jeweiligen Systems übernommen wird und der Benutzer im Idealfall keinen Unterschied feststellen kann, ob er eine native Applikation benutzt oder eine plattformunabhängige.

2. Bibliotheken, die die Darstellung der Widgets selbst übernehmen. Applikationen, die mit Hilfe dieser Bibliotheken erstellt werden sehen auf jeder unterstützter Plattform gleich aus, da das Erscheinungsbild einheitlich festgelegt ist.

Der Vorteil von Punkt 1 ist der „elegantere" Ansatz, der Benutzer braucht sich nicht auf ein anderes Erscheinungsbild der Applikation, als er sonst von seiner Umgebung gewohnt ist, einstellen. Geht man von der Annahme aus, dass die Darstellungsroutinen der jeweiligen GUI des verwendeten Betriebssystems bereits relativ ausgereift und performant sind, ist zu erwarten, dass sich die Ausführungsgeschwindigkeit einer Applikation, die dies Routinen benutzt höher ist als die einer Applikation, die ihre eigenen Routinen verwendet.
Punkt 2 bringt vor allem den Vorteil, dass sich die implementierte Applikation auf allen Betriebssystemen identisch verhält, da die Darstellung nicht einer darunterliegenden Schicht überlassen wird. Dies ist vor allem bei der Programmierung von Events von Vorteil, da in diesem Bereich die unterschiedlichen Plattformen teilweise verschiedene Konzepte benutzen und somit für jede Plattform umfangreiche Tests der GUI vorgenommen werden müssen.

Implementierung als Webapplikation

Entscheidet man sich dafür, eine Benutzerschnittstelle als Webseite zu realisieren die in jedem Webbrowser angezeigt werden soll, kann man zwei grundsätzliche Strategien verfolgen:

1. Verwendung offizieller Standards: Organisationen, wie zum Beispiel das *World Wide Web Consortium (W3C)* entwickeln standardisierte Formate zum Datenaustausch, die Applikationen unterschiedlicher Hersteller implementieren um Informationen einheitlich und herstellerunabhängig anzuzeigen.

2. Per Plug-in nachrüstbare Lösungen: Um die Darstellungsfähigkeiten von Webbrowsern zu erweitern existieren zahlreiche separat zu installierende Applikationen von verschiedenen Herstellern. Diese erlauben es zwar oft modernere und funktionellere Benutzerschnittstellen zu implementieren, die entsprechenden Plug-ins müssen dann allerdings für alle Browser auf allen Plattformen die man einsetzen möchte existieren.

Der große Vorteil von Plug-ins ist, nicht auf langwierige Standardisierungsverfahren angewiesen zu sein und in vielen Fällen große gestalterische Freiheit zu haben. Allerdings setzt man damit wie schon erwähnt auf proprietäre Technik, bei deren Verfügbarkeit man auf die Marktstrategie des Herstellers angewiesen ist. Weiters sollte man bedenken, dass es für die Nutzung eines Dienstes von erheblichem Nachteil sein kann, wenn die potenziellen Benutzer zuvor umständlich das Plug-in herunterladen und installieren müssen und die ursprünglich angeforderten Seiten nicht sofort verfügbar sind.

Setzt man auf etablierte Standards, ist die Wahrscheinlichkeit seine Benutzer durch unvorhergesehene Downloads zu frustrieren geringer, da davon ausgegangen werden kann, das die verwendete Software am anderen Ende mit den empfangenen Daten umzugehen weiß. Hat man hingegen besondere Anforderungen, wie zum Beispiel pixelgenaues Layout der Benutzerschnittstelle oder interaktive dreidimensionale Darstellungen, kann es sein dass man diese nicht oder nur über umständliche Workarounds realisieren kann.

5.1.2 Programmiersprache

Ein weiteres wichtiges Entscheidungskriterium für oder gegen ein Cross-Platform Toolkit ist die unterstützte Programmiersprache. In den meisten Fällen ist ein Toolkit nur für eine Programmiersprache ausgelegt, manchmal werden jedoch auch mehrere unterstützt. Hierbei sollte man in Betracht ziehen, dass die Programmierung in einer interpretierten Sprache wie zum Beispiel Phython und mehr oder weniger auch Java einen Verlust an Performance nach sich zieht, verglichen mit Applikationen die in einer Sprache implementiert sind, die direkt ausführbare Binärdateien erzeugt. Besonders bei der Entwicklung von Benutzerschnittstellen ist es wichtig, auf angemessene Ausführungsgeschwindigkeit auch auf älterer Hardware zu achten, da Wartezeiten bei häufig wiederkehrenden Aktionen, wie etwa das Öffnen eines Menüs, von den Benutzern als sehr lästig empfunden werden.
Bei der Auswahl einer geeigneten Sprache für die Implementierung einer Benutzeroberfläche als Webseite ist es ohnehin gängige Praxis fast ausschließlich interpretierte Sprachen zu verwenden, falls man Java Servlets und Java Server Pages nicht zu der Gruppe interpretierter Sprachen zählt. Auch im Bereich der Webprogrammierung zeigen sich natürlich wieder Performanceunterschiede zwischen den Sprachtypen.
Sieht man sich mit dem Problem konfrontiert, eine Technologie zur Implementierung eines Projektes zu wählen, trifft man oft auf relativ neue Programmiersprachen wie Java oder C# um nur zwei Beispiele zu nennen. Diese Sprachen sind zwar oft sehr vielversprechend, was die Effektivität und die universelle Einsetzbarkeit des generierten Codes betrifft, jedoch weiß man nicht, wie sie sich am Markt behaupten und ob der anfängliche Hype über einen längeren Zeitraum anhält und sich soetwas wie ein allgemein akzeptierter Standard entwickelt. Ist dies nicht der Fall, setzt man unter Umständen auf eine Technologie, die bereits, überspitzt formuliert, nach zwei Jahren von niemandem mehr benutzt wird und man ist mit Problemen wie fehlenden Support (vom Hersteller und/oder der Benutzergemeinschaft), veralteten Tools und Erweiterungen konfrontiert.
Deshalb wäre es günstig, bei der Neu- oder Reimplementierung eines Projektes Technologien zu verwenden, die möglichst weit verbreitet sind und schon einige Zeit existieren. Dies bringt zwar den Nachteil mit sich, gewünschte Funktionalität unter Umständen recht aufwändig selbst implementieren zu müssen (zum Beispiel Plattformunabhängigkeit, Netzwerkfunktionalität usw.) beziehungsweise sich nach bereits implementierten Lösungen umzusehen. Andererseits minimiert man auch

die Wahrscheinlichkeit, aufgrund von noch nicht bekannten Bugs und Problemen, das Projekt grundsätzlich zu gefährden und kann auf eine breite Codebasis und Erfahrung bei Schwierigkeiten zurückgreifen.

5.1.3 Lizenzrechtliches

Gerade bei der Entwicklung von kommerziellen Applikationen ist die Lizenz unter der ein Toolkit beziehungsweise eine Programmiersprache veröffentlicht ist, von größter Wichtigkeit. Es muss sichergestellt sein, dass die erstellten Programme kommerziell vertrieben werden können, ohne den Quellcode veröffentlichen zu müssen.
Die *GPL* (*Gnu Public License*), unter der manche Toolkits veröffentlicht sind, macht dies zum Beispiel nicht möglich, da man den erzeugten Quellcode wieder öffentlich zugänglich machen müsste.
Bei der *LGPL* (*Library General Public License*, später zu *Less Restrictive General Public License* umbenannt) fallen Arbeiten, die dazu ausgelegt sind gegen eine Biblithek unter der LGPL dynamisch gelinkt zu werden, nicht automatisch wieder unter die LGPL. Verteilt man allerdings ausführbare Dateien einer Applikation, die gegen eine Bibliothek unter der LGPL statisch gelinkt sind, enthalten diese teilweise Code der Bibliothek und die Applikation muss wieder unter die LGPL und somit der gesamte Quellcode zur Verfügung gestellt werden.
Bei beiden genannten Lizenzen ist der Autor der Applikation verpflichtet, einen Hinweis auf die verwendete Lizenz an prominenter Stelle innerhalb der Applikation anzubringen.
Eine relativ neue Open Source Lizenz ist die *MPL* (*Mozilla Public Licence*), eine sehr liberale und für die professionelle Weiterverwertung geeignete Lizenz. Anders als bei der LGPL muss sich der Autor nicht mehr um den verwendeten Open Source Quellcode kümmern. Er ist nur mehr dazu verpflichtet, die Copyright Notiz innerhalb des Quellcodes beizubehalten und etwaige Modifikation am Open Source Code wieder zu veröffentlichen. Entsprechende Veränderungen sind natürlich als solche zu Kennzeichnen.
Weiters existiert die *AL (Apache License)* in der aktuellen Version 2.0. Sie besagt, dass man Software die unter dieser Lizenz veröffentlicht wurde in jedem Umfeld verwenden, verändern und weitergeben darf. Bei Weitergabe von Software unter dieser Lizenz muss eine Kopie der Lizenz dem Programmpaket beiliegen, der Quellcode muss nicht beigelegt werden. Auch Änderungen am Quellcode müssen nicht an die Apache Software Foundation zurückgeschickt werden. Falls eigene Software erstellt wird, die Apache Software verwendet, muss erstere nicht unter der Apache Lizenz veröffentlicht werden.
Leider gibt es zur Zeit über 50 unterschiedliche Open Source Lizenzen, alle diese hier anzuführen oder gar zu beschreiben wäre weder sinnvoll noch möglich, da laufend neue Lizenzen hinzukommen, die allerdings oft nur geringfügige Abwandlungen von bestehenden Lizenzen sind. Die obige Beschreibung soll also nur mit der Problemstellung vertraut machen und ist nicht auch nur im Entferntesten

	AL	GPL	LGPL	MPL
Mischung mit kommerziellem Code	Ja	Nein	Ja	Ja
Autor muss im Programm genannt werden	Ja	Ja	Ja	Nein
Autor muss in der Dokumentation/Quellcode genannt werden	Ja	Ja	Ja	Ja
Lizenz muss beigegeben werden	Ja	Ja	Ja	Nein
Bei Mischung muss Originalcode beigegeben/bereitgesellt werden	Nein	Ja	Ja	Nein

Tabelle 3: Übersicht über einige Open Source Lizenzen

vollständig. Viele kommerzielle Softwarehersteller werden durch die Vielzahl der existierenden Open Source Lizenzen daran gehindert, ihre Open Source Projekte schnell in die Realität umzusetzen, da zuerst geprüft werden muss, ob nicht eventuell Lizenzrechte verletzt werden.
Nähere Informationen zu den oben genannten und anderen im Open Source Bereich verwendeten Lizenzen sind unter [7] einsehbar. Viele Toolkits kommen mit einer eigenen, proprietären Lizenz, die allerdings oft auf der GPL oder LGPL basiert, um auch das Entwickeln von kommerzieller Software zu ermöglichen.

5.1.4 Funktionsumfang

Implementierung als eigenständige Applikation

Ein wichtiges Unterscheidungskriterium der diversen existierenden Toolkits zur plattformunabhängigen Programmierung ist natürlich ihr Funktionsumfang. Manche Toolkits verfolgen einen sehr minimalistischen Ansatz, der sich darauf konzentriert, die Größe der ausführbaren Dateien möglichst gering und damit die Ausführungsgeschwindigkeiten hoch zu halten. Dies ist vor allem dann interessant, wenn man die Bibliotheken statisch linkt und die Größe der Applikation dennoch klein halten kann. Andererseits stellt es meistens auch kein besonders großes Problem dar, das Toolkit als dynamische linkbare Bibliothek auf dem jeweiligen System zu installieren. Benötigt man zusätzliche Funktionalität, die vom Toolkit nicht angeboten wird, muss man dafür auf andere Quellen zurückgreifen und deren plattformunabhängigkeit separat prüfen.
Im Gegensatz dazu existieren einige Toolkits, die relativ umfangreich sind und abgesehen von den Klassen zur Programmierung einer Benutzerschnittstelle auch verschiedenste Klassen für Datenstrukturen (Listen, Strings, Hashtabellen usw.) mitbringen, als auch solche für die Unterstützung der Netzwerkfunktionalität oder

Multithreading. Dies ist auf jeden Fall, falls die zur Verfügung gestellte Funktionalität für den jeweiligen Verwendungszweck ausreicht, die bequemere und konsistentere Variante. Alle Klassen des Toolkits sind bereits für die unterstützten Plattformen getestet und fügen sich konsistent in die Klassenstruktur ein, wodurch die Verständlichkeit des Codes gefördert wird.

Implementierung als Webapplikation

Hierbei gelten ähnliche Kriterien wie bei der Implementierung als eigenständige Applikation, wobei festgehalten werden muss, dass die heutzutage verfügbaren serverseitig verwendeten Sprachen hinsichtlich ihrer gebotenen Funktionalität locker die Anforderungen der CSPmed Applikation erfüllen.

5.1.5 Eventprogrammierung

Implementierung als eigenständige Applikation

Unter den diversen Toolkits gibt es auch verschiedenste Mechanismen, um Reaktionen auf Events zu implementieren. Als Event wird zum Beispiel der Klick auf einen Button oder das Verschieben eines Fensters verstanden. Events können sowohl vom Benutzer der Applikation abgesetzt werden, als auch vom Betriebssystem, das zum Beispiel das Neuzeichnen des Fensters mit Hilfe einer Nachricht die es an die entsprechende Applikation sendet, veranlasst. Bei der Entscheidung für oder gegen ein Toolkit sollte man vor allem berücksichtigen ob:

- Events zur Laufzeit verbunden und wieder getrennt werden können
- Elemente der Benutzerschnittstelle Events sowohl senden als auch empfangen, oder diese nur senden können
- sich die Eventbehandlung des Toolkits in objektorientierte Sprachen integrieren lässt
- die Eventbehandlung typensicher ist
- ein spezieller Präprozessor verwendet werden muss

5.1.5.1 Funktionszeiger Das Codebeispiel in Algorithmus 4 zeigt die Verwendung von Call-back Funktionen in dem Cross-Platform Toolkit FLTK ([8]) und soll zur Erklärung des Konzepts dienen.
In dem Beispiel wird ein Menü einer Applikation definiert, wobei der Eintrag „`&New`" von besonderem Interesse ist, da er einen Zeiger auf eine Funktion beinhält. Wenn die Applikation also eine Nachricht vom Betriebssystem erhält, dass der Menüpunkt „New" ausgewählt wurde, dann wird innerhalb der Applikation aufgrund obigen Codes die Funktion `new_cb` aufgerufen.

Algorithm 4 Konzept einer Call-back Funktion

```
Fl_Menu_Item menuitems[] =
{ { "&File", 0,0,0, FL_SUBMENU},
{ "&New", FL_ALT + 'n', (Fl_Callback *) new_cb },
...
...

void new_cb(void)
{
    if (changed)
        if ( !check_save() ) return;
    filename[0] = '\0';
    input->value("");
    set_changed(0);
}
```

Algorithm 5 Beispiel einer Eventtabelle in wxWidgets

```
BEGIN_EVENT_TABLE (BasicFrame, wxFrame)
    EVT_MENU ( BASIC_EXIT, BasicFrame::OnExit)
    EVT_MENU ( BASIC_ABOUT, BasicFrame::OnAbout)
    EVT_MENU ( BASIC_OPEN, BasicFrame::OnOpenFile)
END_EVENT_TABLE()
```

Vorteil des Konzepts der Call-back Funktionen ist, dass es relativ einfach zu verstehen und zu implementieren ist, jedoch ist es nicht objektorientiert und somit schwierig in Sprachen wie zum Beispiel C++ oder Java zu integrieren. Weiters ist die Typensicherheit nicht gewährleistet, das heißt, dass man nicht sicher sein kann, dass der aufzurufende Funktion auch die korrekten Paramter übergeben werden. Es besteht auch eine sehr starke Bindung zwischen der aufrufenden und der aufgerufenen Funktion, da erstere in jedem Fall den Funktionszeiger zur Call-back Funktion immer kennen muss.

5.1.5.2 Eventtabellen Bei diesem Ansatz wird mit Hilfe von Makros eine Tabelle definiert, die die Verbindung zwischen Event und Methode herstellt (siehe Algorithmus 5).
Die erste Zeile des Codeausschnittes der als Algorithmus 5 dargestellt ist, markiert den Anfang der Tabelle, die letzte das Ende. Die drei Statements dazwischen verbinden eine Konstante mit einer Klassenmethode. Falls also von dem Menü des Fensters Basicframe zum Beispiel der Event `BASIC_OPEN` gesendet wird, wird die entsprechende Methode aufgerufen. Die Konstanten (`BASIC_OPEN`, `BASIC_ABOUT`, `BASIC_EXIT`) können dabei aus einem gewissen freien Bereich gewählt werden und werden beim Erzeugen des entsprechneden Menüpunktes angegeben. Die Eventtabellen werden gemäß der Fensterhierarchie durchgegangen

Algorithm 6 Definition einer Massagemap mittels FOX-Toolkit

```
FXDEFMAP(FXGLViewer) FXGLViewerMap[]={
    FXMAPFUNC(SEL_PAINT,0,FXGLViewer::onPaint),
    ....
    FXMAPFUNCS(SEL_UPDATE,MINKEY,MAXKEY,
FXGLViewer::onUpdAll),
};
FXIMPLEMENT(FXGLViewer,FXGLCanvas,FXGLViewerMap,
    ARRAYNUMBER(FXGLViewerMap))
```

und diejenige Klasse, deren Eintrag in der Eventtabelle „passt", bearbeitet den Event.
Man kann in einem Programm natürlich mehrere Eventtabellen definieren, sinnvollerweise wird man das zum Beispiel für jede Klasse machen, die ein Menü darstellt. Events werden allerdings nicht nur in Verbindung mit Menüs verwendet, sondern auch wenn Benutzerinteraktionen (zum Beispiel ein Tastendruck oder ein Mausklick) stattfinden, Fenster gezeichnet oder geöffnet werden und vieles mehr. Eventtabellen werden in verschiedenen Toolkits verwendet und treten dort eventuell unter einer anderen Bezeichnung auf, wie zum Beispiel beim FOX-Toolkit, wo sie als *Message Map* bezeichnet wird.
Die Syntax der Message Maps weicht von jender der Eventtabellen in wxWidgets leicht ab, beschreibt jedoch das gleiche Konzept. Auch bei dieser Technik wird eine Nachricht mit angegebener MessageID an ein angegebenes Ziel gesendet. Die MessageID wird benötigt um den Absender feststellen zu können, da ein Ziel Nachrichten von mehreren Widgets erhalten kann. Weiters wird auch noch der Nachrichtentyp mitgeschickt, da ein Widgets auch meherere unterschiedliche Nachrichtenarten verschicken kann. Jedes Objekt kann prinzipiell in der Lage sein, Nachrichten entgegenzunehmen. Die Nachrichten an ein Objekt werden dann mit Hilfe einer Message Map an die entsprechenden Memberfunktionen „gemapped". Zur Illustration ist ein kurzer Codeauszug in Algorithmus 6 angegeben.
Das Makro `FXMAPFUNC` benötigt als Argumente den Nachrichtentyp, die MessageID und die Funktion die der Nachricht zugewiesen wird. `FXMAPFUNCS` kann einen ganzen Bereich an Nachrichten zuweisen. Auch bei dieser Implementerung werden die Message Maps gemäß der Klassenhierarchie in Richtung Basisklasse durchgegangen und die Memberfunktion der ersten „passenden" Klasse aufgerufen. Das letzte Makro im Algorithmus, `FXIMPLEMENT`, muss in der Implementierungsdatei angegeben sein und bekommt vier Parameter übergeben: Den Namen der Klasse und der unmittelbaren Basisklasse sowie einen Zeiger auf die Message Map und die Anzahl der Einträge in der Message Map.

5.1.5.3 Signal/Slot Mechanismus Dies ist ebenfalls eine Technik zur Kommunikation zwischen Objekten. Falls ein Ereignis eintritt, also wenn zum Beispiel ein Textfeld seinen Inhalt ändert, wird von diesem Objekt ein *Signal* ausgesendet. Die-

Algorithm 7 Beispiel einer Message Map

```
BEGIN_MESSAGE_MAP(CMainWindow, CFrameWnd)
    ON_WM_PAINT()
END_MESSAGE_MAP()
```

se Signale sind zum Teil vom Toolkit vordefiniert, aber es können auch eigene erstellt werden. Ein *Slot* ist eine Funktion, die als Antwort auf ein angegebenes Signal aufgerufen wird. Auch von diesen Slots sind einige durch das Toolkit vordefiniert, wie bei den Signalen kann sich der Programmierer jedoch auch auf seine Bedürfnisse abgestimmte Slots erstellen. Der Signal/Slot Mechanismus ist typensicher, da die Signatur des Signals zur Signatur des Slots passen muss. Der Slot kann dabei jedoch eine geringere Anzahl an Parametern aufweisen, in diesem Fall werden die zusätzlichen Parameter ignoriert. Aufgrund der kompatiblen Signaturen kann der Compiler den Programmierer bei Fehlern unterstützen. Die Kopplung von Signalen an Slots ist sehr leicht, da ein Objekt, das ein Signal aussendet nicht weiß, ob und welcher Slot welchen Objektes es empfangen wird. Durch diese Art der Kapselung ist es auch möglich, das Objekt als Softwarekomponente wiederzuverwenden.
Technisch gesehen sind Slots einfach Memberfunktionen eines Objekts. Dieses Objekt hat keine Information darüber, ob und welche Signale zu seinen Memberfunktionen verbunden sind. Diese Eigenschaft hilft wiederum, eine Informationskapselung durchzuführen und das Objekt in eine Softwarekomponente einzubetten. Zu einem Slot können mehrere Signale verbunden werden, und umgekehrt. Ein Signal kann auch direkt mit einem anderen verbunden werden, was bewirkt dass das zweite Signal ausgesendet wird, sobald das erste auftritt.

5.1.5.4 Nachrichtenverarbeitung mit fixem Routing Eine Technik, die sehr ähnlich den oben erwähnten Eventtabellen ist, kommt bei den Microsoft Foundation Classes zum Einsatz, was für eine Portierung von MFC-Applikationen natürlich sehr angenehm ist. Microsoft nennt die Technik zur Behandlung von Events, die das Betriebssystem an die Applikation sendet, ebenfalls *Message Maps* (siehe dazu Algorithmus 7).
Das oben angeführte Beispiel zeigt eine eigens erstellte Message Map der Klasse `CMainWindow`, welche von `CFrameWnd` abgeleitet ist. Message Maps können vererbt werden und werden ebenfalls wie Eventtabellen in Richtung der Basisklassen abgearbeitet. Im Unterschied zu den Eventtabellen ist es bei manchen Messages, wie bei jener in obigem Beispiel, nicht möglich, Parameter anzugeben. In obigem Fall wird daher bei Eintreffen der Message `WM_PAINT` die Standardmethode `OnPaint()` aufgerufen, die der Programmierer entsprechend überschreiben muss. Die Argumente der Behandlungsmethoden sind dabei wieder unterschiedlich und müssen in der Dokumentation nachgeschlagen werden. Die Methode `OnButtonLDown()` zum Beispiel (aufgerufen bei Mausklick mit der linken Taste) erwartet als Parameter ein Flag-Objekt, das den Zustand der Maustasten re-

präsentiert und ein Point-Objekt, das die Koordinaten des Mauszeigers beinhaltet. Ein Nachteil dieser Methode ist, dass GUI-Elemente Messages nur senden und nicht empfangen können. Ebenso müssen die Messages aufgrund des fixen Routings global einheitlich sein.

5.1.5.5 Objektorientierte Methode Auf diese Art wird das Eventhandling in Java gelöst. Damit ein Objekt einen Event empfangen kann, muss es ein entsprechendes Interface implementieren und beim „Verursacher" des Events als Eventlistener angemeldet werden. Konkret sieht die Deklaration einer Klasse, die auf Events reagieren kann folgendermaßen aus:

```
public class MyClass implements ActionListener {
```

Eine andere Codezeile registriert eine Instanz obiger Eventhandler-Klasse als „Listener" von einer oder mehreren Komponenten (zum Beispiel Fenster oder Buttons)

```
someComponent.addActionListener(instanceOfMyClass);
```

Die Eventhandler-Klasse implementiert die Methoden des `ActionListener`-Interfaces, zum Beispiel folgendermaßen:

```
public void actionPerformed(ActionEvent e)
{ //Code der beim Erhalt des Events ausgeführt wird...}
```

Neben ActionListenern, die zum Beispiel auf Buttonklicks, Tastendrücke und Menüauswahlen reagieren, existieren auch `WindowListener`, die auf Ereignisse reagieren, welche von Fenstern ausgesendet werden (etwa wenn der Benutzer ein Fenster schließt) oder MouseListener, die dazu verwendet werden können um den Status der Maustasten abzufragen. Dieses Konzept der Eventbehandlung ist sehr logisch, einfach zu erlernen und sehr gut in die objektorientierte Programmierweise integriert.

Implementierung als Webapplikation

In diesem Fall ist es notwendig eine Skriptsprache einzusetzen, die clientseitig die Benutzereingaben überprüft und darauf basierend entsprechenden Code ausführt. Hinsichtlich der Unterscheidungsmerkmale können wieder die Kriterien die bereits in 5.1.2 beschrieben wurden angewendet werden.

5.1.5.6 Eignung der Konzepte für die Implementierung Mit Ausnahme des Konzepts der Funktionszeiger stellen alle oben angeführten Möglichkeiten der Eventbehandlung zeitgemäße Alternativen dar und eignen sich daher für die Verwendung bei der Implementierung des CSPmedclients. Funktionszeiger sollten aufgrund ihrer Verletzung des objektorientierten Konzepts nicht verwendet werden, da sie sich sehr schlecht eignen, übersichtlichen und stabilen Code zu schreiben.

5.2 Diskussion der einzelnen Konzepte

5.2.1 WINE

Wine ([1]) ist eine Open Source Implementierung der Microsoft Windows API, die auf das X-Window System und Unix aufsetzt und für Linux, Solaris und FreeBSD ausschließlich für x86 Prozessoren verfügbar ist. Dabei steht Wine für „Wine Is Not an Emulator", was verdeutlichen soll, dass es sich hierbei um eine Art Windows-Kompatibilitätslayer handelt. Das Projekt hat das Ziel, Applikationen die für Microsoft Windows implementiert wurden, auf einem Unix-System auszuführen, was mittels *Wine Program Loader* und *Winelib* erfolgen kann. In beiden Fällen benötigt man keine Microsoft Windows Lizenz, Wine selbst besteht zur Gänze aus (Microsoft-) freiem Code.

Kompatibilität und Performance

Wine unterstützt sowohl Win32 (Windows 9x, NT, XP), als auch Win16 (Windows 3.x) und DOS Binärdateien, es kann also 32bit und 16bit Code ausgeführt werden. Falls spezifische Funktionalität fehlt, oder deren Implementierung im Wine Projekt noch nicht so weit fortgeschritten ist, kann man auch original Microsoft DLL Dateien mit Wine benutzen. Dies wirft allerdings wieder lizenzrechtliche Fragen auf, da man, falls man auch nur eine DLL des Microsoft Windows Betriebssystems verwendet, man eine Windows-Lizenz erwerben muss, womit die Microsoft-Unabhängigkeit wieder nicht gegeben wäre. Beim derzeitigen Entwicklungsstand von Wine ist es jedoch leider oft nötig, die original DLL Dateien von Microsoft zu verwenden um viele Windows Applikationen fehlerfrei ausführen zu können.
Laut Wine Website ([1]) ist die Performance fast gleich mit native Unix Applikationen. Die Windows-Kompatibiliätsschicht nimmt zwar etwas Performance weg, doch falls die Windows Applikation effizient implementiert ist, kann sie ohne Weiteres schneller als eine wenig effiziente Unix Applikation unter Wine laufen.
Ein Nachteil von Wine ist, dass es von der Funktionalität her den aktuellen Microsoft Windows Versionen zwangsläufig hinterherhinken muss. Neue DLLs und Funktionalität wird aufgrund der Architektur von Wine zwar relativ schnell hinzugefügt, ebenso schnell können diese durch neue Versionen von Microsoft jedoch wieder verändert werden.
Zum Zeitpunkt der Verfassung dieses Dokuments existierten nur 9 Applikationen, die ohne Probleme mit Wine zusammenarbeiten, sowohl bei der Installation als auch bei der täglichen Verwendung. Dabei handelte es sich um FTP-Clients, Mailprogramme und Dateiutitlities, also Applikationen, die unter GNU/Linux zuhauf als native Versionen exisiteren.

Wine Program Loader

Der Wine Program Loader soll Binärkompatibilität zu bestehenden Microsoft Windows Applikationen ermöglichen, da man diese zum Ausführen unter Unix bezie-

hungsweise GNU/Linux nicht neu kompilieren muss. Dazu führt man einfach Wine mit der Microsoft Windows Applikation (Normalerweise eine .exe Datei) als Kommandozeilenparameter aus.

Winelib

Winelib ist ein Entwicklungstoolkit das es erlaubt, Microsoft Windows Applikationen unter Unix beziehungsweise GNU/Linux zu kompilieren. Es handelt sich hierbei um eine Bibliothek, mit der man bestehenden Microsoft Windows Code übersetzen kann, und eine ausführbare Unix Datei erhält. Allerdings wird 16bit Quellcode sowie COM und structured exception handling nicht unterstützt.
Der Vorteil, den man erhält, wenn man eine Applikation mit Hilfe von Winelib kompiliert, ist dass man Unix API-Calls direkt aus dem Windows Code absetzen kann, was eine bessere Integration in die Unix beziehungsweise GNU/Linux Umgebung gewährleistet.
Möchte man MFC-Applikationen mit Winelib kompilieren, muss man zuerst die MFC selbst kompilieren. Leider ergeben sich durch diesen Umstand rechtliche Probleme:

- Die MFC sind unter einer sehr restriktiven Lizenz veröffentlicht, und diese Restriktionen ändern sich unter Umständen von Version zu Version aber auch zwischen Service Packs.
- Um den MFC Code zu bekommen muss man eine Lizenz von Visual Studio kaufen. Der MFC Code ist dann allerdings über die Winelib Bibliothek auch im fertig kompilierten Produkt enthalten, sodass man vermutlich für jeden Arbeitsplatz, der die portierte Applikation verwenden soll, eine Visual Studio Lizenz benötigt.
- Unter Umständen ist es von Microsoft nicht erlaubt, die MFC unter einem anderen Betriebssystem zu kompilieren. Hier ein Auszug aus der MFC-Lizenz von Microsoft Visual Studio 6.0:

  > „1.1 General License Grant. Microsoft grants to you as an individual, a personal, nonexclusive license to make and use copies of the SOFTWARE PRODUCT for the sole purposes of designing, developing, and testing your software product(s) that are designed to operate in conjunction with any Microsoft operating system product. [...]"

  Es scheint also so, dass es nicht erlaubt ist, MFC für Winelib zu kompilieren, wenn man an diese Lizenz gebunden ist. Ein Auszug aus der Microsoft Visual Studio 6.0 Service Pack 3 Lizenz lautet folgendermaßen:

  > „1.1 General License Grant. Microsoft grants to you as an individual, a personal, nonexclusive license to make and use copies of

> the SOFTWARE PRODUCT for the purpose of designing, developing, and testing your software product(s). [...]"

Unter obiger Lizenz scheint es also legal zu sein MFC für die Winelib zu kompilieren.

- Man muss prüfen, ob man das Recht besitzt, eine MFC Bibliothek zu publizieren. Hierzu gibt es wiederum eigene Absätze in der Microsoft Visual Studio MFC Lizenz.

Verwendbarkeit für den CSPmed Client

Startet man den CSPmed Client mit Hilfe von Wine unter GNU/Linux, bemerkt man dass die Darstellung einiger Objekte fehlerhaft ist. Weiters fällt die überaus schlechte Performance auf, falls man zum Beispiel Menüs öffnet oder einfach nur Textfelder editiert. Alles scheint mit einiger Verzögerung zu geschehen, was flüssiges Arbeiten in der Geschwindigkeit, die man von dem CSPmed Client unter Windows gewohnt ist, unmöglich macht. Die ursprüngliche Vermutung über die Ursache dieses Verhaltens waren die zahlreichen RPC-Calls, die die Windows Kompatibilitätsschicht passieren müssen. Verwendet man jedoch eine Microsoft Windows Applikation, die nicht auf Netzwerkfunktionalität zurückgreift, stellt man ebenso eine, im Vergleich zu nativen GNU/Linux Applikationen, deutlich geringere Performance der grafischen Oberfläche fest. Abgesehen von diesen Problemen der Darstellung, wird eine Verwendung von Wine durch die in 5.2.1 angesprochenen lizenzrechtlichen Probleme verhindert. Ziel der Umstellung des CSPmed Clients war eine Loslösung von proprietärer Software, die bei Verwendung der MFC wieder gegeben wäre und zwar verbunden mit dem Kauf von Lizenzen. Selbst wenn man Winelib nicht benutzen würde, wäre man weiterhin an die Verwendung von Microsoft Visual C++ gebunden, und könnte den CSPmed Client nur für jene Plattformen anbieten, für die auch Wine verfügbar ist.
Ein weiteres Faktum, das man nicht außer acht lassen sollte ist, dass sich Wine aufgrund seines Anspruchs kompatibel zu Microsoft Windows zu sein, ständig relativ schnell weiterentwickeln muss. Deshalb kann man damit rechnen, dass bedingt durch die Vielzahl an notwendigen Codeänderungen auch sehr viele Fehler enthalten sind. Man hat dann das Problem, dass Fehler auf drei verschiedenen Ebenen auftreten können:

1. CSPmed Server

   Dies sind Fehler in der „eigentlichen" Applikation, die in den meisten Fällen von den Kunden festgestellt werden.

2. CSPmed Client

   Gegenwärtig sehr selten auftretend, da der Client unter Microsoft Windows ziemlich stabil und ausgereift ist. Durch Modifikationen, die bedingt durch

den Einsatz von Wine notwendig sind, ist zu erwarten, dass in der ersten Erprobungsphase Fehler in dieser Ebene gehäuft auftreten werden

3. Wine

   Fehler in dieser Ebene sind schwerer zu finden und zu beheben und können in zukünftigen Versionen von Wine unterschiedliche Auswirkungen haben.

Insgesamt handelt man sich mit Verwendung von Wine also eine zusätzliche Ebene an Komplexität ein, die ebenfalls fehleranfällig ist und noch dazu in ständiger Entwicklung unterliegt. Fehler der Ebenen 1 und 2 konnten zwar auch bisher bei Entwicklung unter Microsoft Windows (Client) und GNU/Linux beziehungsweise SCO (Server) auftreten, diese könnten aber durch die dritte Ebene schwieriger zuzuordnen sein.
Von Vorteil könnte womöglich eine Codebasis sein, die unter Windows und GNU/Linux vollkommen identisch ist. Aufgrund der Fehlerhaftigkeit von Wine und der Tatsache, dass viele Funktionen noch nicht unterstützt werden, ist anzunehmen, dass man unterschiedliche Versionen des CSPmed Clients für den Einsatz unter Windows und GNU/Linux pflegen muss. Man denke zum Beispiel an eventuell notwendige workarounds um Fehler in der grafischen Darstellung von Dialogelementen in Wine zu umgehen. Diese wären unter Windows nicht notwendig beziehungsweise sogar der Funktionalität und/oder der Performance hinderlich.

Zusammenfassung

Vorteile:

- schnelle Erfolge ohne Rekompilierung des Codes
- einheitliche Codebasis realisierbar

Nachteile:

- Lizenzrechtliche Probleme vor Allem in Verbindung mit MFC
- schlechte out-of-the-box Performance der grafischen Benutzerschnittstelle
- zusätzliche Ebene an Komplexität

Aus den in obenstehenden und den in 5.2.1 genannten Nachteilen habe ich mich gegen eine Verwendung von Wine in dieser Arbeit entschieden.

5.2.2 Java

Java ist ein von Sun ([11]) entwickelte Programmiersprache, die seit 1994 verfügbar ist. Sie wurde als Ersatz für C++ im Hinblick auf leichtere Portierbarkeit, höhere Sicherheit, Threadunstertützung und vieles mehr entworfen. Die Sprache ist mitsamt ihrem API sehr vielfältig und versetzt Entwickler in die Lage

- Software auf einer Plattform zu schreiben und auf einer anderen auszuführen
- Applikationen zu schreiben, die in einem Webbrowser laufen
- Serverseitige Applikationen für Onlineforen, Umfragen, eBusiness und vieles mehr zu entwickeln.
- Applikationen für Mobiltelefone und andere Consumergeräte zu entwickeln

Java selbst dürfte bereits allgemein bekannt sein, deshalb soll hier nur ein kurzer Überblick über seine Technologie gegeben werden. Eine Applikation, die in Java geschrieben wurde muss bevor sie ausgeführt werden kann, kompiliert werden, genauso wie eine Applikation in vielen anderen Sprache auch. Der Java Compiler erzeugt allerdings keinen plattformspezifischen Code, sondern generiert einen sogenannten Bytecode, der nur von einer Java Virtual Machine (JVM) interpretiert werden kann. Diese JVM existiert auf jeder unterstützten Pattform und muss genau für diese Plattform implementiert sein, da sie den nativen Code erzeugt. Aufgrund der Tatsache, dass die JVM den Bytecode interpretiert, sind Programme die mit Hilfe von Java implementiert wurden, aus Prinzip langsamer als native Programme, obwohl dieser Unterschied bei den meisten Applikationen auf zeitgemäßer Hardware so gut wie nicht mehr mehr wahrnehmbar ist. Technologien wie optimierte Compiler sowie „just-in-time" Bytecodecompiler bringen die Performance von Java Code in den Bereich nativer C oder C++ Programme. Ein zentraler Bestandteil von Java ist die Java API, die umfangreiche Funktionalität zur Verfügung stellt um damit sehr vielseitige Applikationen zu implementieren. Die Möglichkeiten der Java API umfassen unter Anderem:

- Grundlegendes: Objekte, Strings, Threads, Eingabe/Ausgabe, Datenstrukturen, Systemeinstellungen, Datum/Uhrzeit und vieles mehr
- Netzwerkfunktionalität: URLs, TCP, UDP, IP-Adressen
- Internationalsierung: Hilfe bei der lokalen Anpassung für die weltweite Benutzung der Applikation
- Sicherheit: elektronische Signaturen, public und private Key Unterstützung, Zugriffskontrolle und Zertifikate
- Softwarekomponenten: JavaBeans Komponenten können in existierenden Komponentenarchitekturen eingebunden werden
- Objektserialisierung: erlaubt Kommunikation mittels Remote Method Invocation (RMI), also dem Aufruf von Methoden über das Netzwerk
- Java Database Connectivity (JDBC): Ermöglicht einen einheitlichen Zugriff auf viele verfügbare Datenbanken

Da bei der Implementierung des CSPmedclients der Fokus auf der automatisierten Erstellung einer Benutzerschnittstelle sowie der Kommunikation über das Netzwerk liegt, soll im Folgenden diese Funktionalität von Java beleuchtet werden.

Verwendung von bestehendem Code in Java

Um die JVM in native Applikationen, das heißt in Applikationen, die unter einer anderen Programmiersprache für eine spezifische Plattform erstellt wurden, einzubetten, wurde das *Java Native Interface* (*JNI*) entwickelt. Ziel von JNI ist, Binärkompatibilität von nativen Methoden und allen JVM Implementierungen auf einer Plattform zu ermöglichen. JNI ist Teil des *Java 2 Platform Standard Edition Development Kit* (*JDK*) und erlaubt den Zugriff auf native Methoden aus Javaprogrammen heraus, um zum Beispiel plattformspezifische Funktionalität zu implementieren oder bestehende Bibliotheken beziehungsweise Schnittstellen zu proprietären (*legacy*-) Applikationen verwenden zu können.
Es ist nicht nur möglich, aus Java auf native Methoden zuzugreifen, die native Methode kann auch Java-Objekte genauso verwenden wie in Java implementierter Code dies tut, also diese erzeugen, verändern und auf bereits bestehende zugreifen. JNI fungiert dabei als Schicht zwischen Java und der nativen Sprache, es sind prinzipiell zwei verschiedene Szenarien denkbar:

Einbindung von nativem Code in Javaprogramme: Hierbei ist es ist notwendig, den nativen Code als shared Library zu kompilieren und die Signatur von bereits bestehendem einzubindenden Code anzupassen. Letzteres ist notwendig, da das Programm `javah` mit der Javaklasse, die den Aufruf der nativen Methode beinhaltet, als Parameter ausgeführt werden muss. Das Dienstprogramm `javah` erstellt eine C-Headerdatei mit der Signatur der nativen Methode, die auch in der zugehörigen C-Implementierungsdatei enthalten ist. Bestehender Code muss, wie erwähnt, dann natürlich an die Signatur in der Headerdatei angepasst werden, da der Methodenname vom JNI verändert wird sowie zwei zusätzliche Parameter übergeben werden. Diese Parameter ermöglichen zum Einen, auf Parameter und Objekte zuzugreifen, die von Java an die native Methode übergeben werden und zum Anderen eine Referenz auf das Objekt aus dem die native Methode aufgerufen wurde, also quasi ein Zeiger auf `this`. Primitive Java-Datentypen wie zum Beispiel `boolean`, `int` oder `double` können im nativen Code einfach als `jboolean`, `jint` oder `jdouble` angesprochen werden. Für den Zugriff auf Objekte wie zum Beispiel `string` stellt JNI Interfacefunktionen zur Verfügung, die diese Objekte zum Beispiel unter C verwendbar machen, also etwa eine Konvertierungsfunktion von der standardmäßigen Unicode Repräsentation von Strings in Java auf das in C gebräuchliche UTF-8 Format. Die Interfacefunktionen können mit Hilfe des ersten Parameters angesprochen werden, der jedem Aufruf einer nativen Methode übergeben wird und sozusagen eine Funktionstabelle darstellt. Über sie ist es möglich, auf jegliche Art von Javaobjekten und -funktionen zuzugreifen und Informationen zwischen Java und der nativen Seite in beide Richtungen zu übertragen.

Einbindung der JVM in native Programme: Dies ist gewissermaßen die umgekehrte Situation zu obigem Punkt da man davon ausgeht, bereits funktionsfähi-

gen C-Code zu besitzen und in diesem Code ein Javaprogramm einbinden möchte. Die JVM wird von Sun als shared library ausgeliefert, was es ermöglicht, dies durch Linken zu einer nativen Applikation einfach bewerkstelligen zu können. Die im JNI enthaltene C-Headerdatei `jni.h` und die shared library `libjava.so` ermöglichen es, die aufzurufende JVM zu initialisieren und zu starten, der aktuelle native Thread klinkt sich in die JVM ein und läuft dort wie eine native Methode (siehe oben). Bei der Erzeugung der JVM erhält man wieder einen Zeiger auf eine Tabelle mit Konvertierungsfunktionen, die man dann genauso wie oben beschrieben benutzen kann, um auf die Java-Funktionalität zugreifen zu können.

Implementierung einer grafischen Oberfläche

Um eine grafische Benutzeroberfläche mit Hilfe von Java zu implementieren, kann man entweder die veralteten *AWT* Klassen oder die *Java Foundation Classes* (*JFC*) verwenden. Die JFC bauen zum Teil auch auf AWT auf, bieten aber ein moderneres Aussehen als diese. Die Komponenten von JFC umfassen weiters:

- Java2D: Eine 2D Grafik-API basierend auf Technologie von IBM/Taligent.
- Accessibility: Eine API, die Unterstützung für Benutzer mit Behinderungen bereitstellt, wie zum Beispiel Vergrößerung eines Bildschirmbereichs oder alternative Eingabemethoden.
- Drag&Drop Unterstützung
- Swing: Mit diesem Namen bezeichnet man die Gesamtheit an Komponenten, die bei den JFC für die Darstellung von Formularbasierten Applikationen zuständig sind. Swing stellt eine Menge von ausgereiften Widgets zur Verfügung, die man für die Gestaltung von grafischen Benutzeroberflächen (GUIs) benötigt. Swing ist als eine Erweiterung von AWT um neue Komponenten wie etwa Baumansichten oder Karteireiter anzusehen. Viele der Komponenten in Swing sind sogenannte *leightweight components*, also Komponenten die zu ihrer Darstellung nicht auf Methoden des darunterliegenden Betriebssystems angewiesen sind.

Entscheidet sich man für die Entwicklung einer GUI in Java ist es sinnvoll Swing einzusetzen, da es im Gegensatz zur ausschließlichen Verwendung von AWT viele Vorteile bietet. So kann man etwa das Look&Feel, also das Aussehen der Oberfläche, dynamisch Verändern und so Motif, GTK oder Microsoft Windows Oberflächen nachahmen. Weiters ist es mit Swing möglich, Benutzerschnittstellen zu entwickeln, die auf allen Plattformen großteils gleich aussehen und sich der Programmierer nicht mehr um Auflösung, Schriftgröße und Schriftart oder Ähnliches auf der Zielplattform kümmern muss.

Eventsystem

Implementiert man in Java eine Applikation mit grafischer Benutzeroberfläche und benutzt dabei die AWT oder Swing Klassen, dann kümmert sich die JVM um die Ausführung des Eventloops, der Programmierer kann und braucht diesen nicht explizit implementieren oder aufrufen. Man programmiert also wie gewohnt seine Applikation samt `main()` Methode, instanziiert die gewünschten GUI-Objekte und zeigt diese an, danach verlässt man die `main()` Methode einfach. Das Programm wird nun aber nicht beendet sondern es wird in einem separaten Thread ein Eventloop ausgeführt der überprüft, welche Aktionen (Mausklicks, Tasteneingaben und Ähnliches) der Benutzer durchführt und diese dann an die Behandlungsroutinen der laufenden Applikation weiterleitet. Für den Programmierer gibt es also keine Möglichkeit, wie etwa bei den MFC oder GTK+ Toolkits, einen Eventloop explizit aufzurufen. Diese Eigenschaft von Java ist der Grund, warum man bei einer eventuellen Implementierung des CSPmedclients in dieser Sprache die Kommunikationsalgorithmen des Clients mit dem Server grundlegend überarbeiten müsste. Der Aufruf von Methoden der libcsprpc via *JNI* (*Java Native Interface*) würde zwar grundsätzlich möglich sein, ist jedoch mit dem Eventkonzept von Java nicht verträglich. Man müsste vielmehr einen Mechanismus implementieren, der im Falle eines RPC-Calls einen Event mit der angeforderten Aktion (`Ptform`, `Rdform` usw.) sendet, auf den man im Programm reagiert. Der `Rdform` Aufruf würde meiner Meinung nach der problematischste aller Aufrufe sein, da er erst einen Rückgabewert liefern darf, wenn der Benutzer eine Aktion in der GUI gesetzt hat. Dies könnte man eventuell lösen, indem man das anzuzeigende Formular modal öffnet und den Rückgabewert auswertet, wie zum Beispiel beim Aufruf von `JFileChooser.showDialog()`.

Versionen

Das Einsatzgebiet von Java ist sehr breit gefächert, es ist möglich stand-alone Applikationen zu implementieren, Applikationen, die für clientseitigen Einsatz gedacht sind (*Applets*) sowie Applikationen die auf einem Server laufen und dessen Fähigkeiten erweitern (*Servlets*). Darüberhinaus existieren mächtige Lösungen für die kommerzielle Softwareentwicklung wie zum Beispiel *JavaBeans* als Möglichkeit zur Softwarewiederverwendung. Gleichzeitig ermöglicht die Java-Technologie aber auch Applikationen die auf Mobiltelefonen lauffähig sind sowie auf Geräten im embedded Bereich, die im Allgemeinen über stark begrenzte Speicherkapazität und Rechenleistung verfügen. Um all diese Einsatzgebiete unter einen Hut zu bringen, existieren verschiedene Editionen der Java2-Plattform:

***Standard Edition* (*J2SE*):** Diese Edition bietet eine Umgebung zur Entwicklung von Applikationen die auf Desktops und Servern laufen sollen sowie in embedded Umgebungen. Sie lässt sich wiederum in vier Gruppen unterteilen:

- Core Java: Diese APIs bilden die Grundlage der Java 2 Plattform. Sie umfassen sicherheitsrelevante Funktionalität (Authentifizierungsmechanismen, sichere Kommunikation über Sockets), Anbindung an Datenbanken (*Java Database Connectivity*, *JDBC*), Javadoc als Dokumentationstool und andere Technolgien wie Internationalisierung, entfernte Methodenaufrufe (*Remote Method Invocation*, *RMI*) und *Java Naming and Directory Interface* (*JNDI*).
- Desktop Java: In Verbindung mit Core Java können diese APIs verwendet werden, um Applikationen und Applets mit Benutzerschnittstellen zu implementieren, die sicher und portabel sind. Die Java Foundation Classes sind Teil dieser Gruppe, genauso wie das Komponentenmodell JavaBeans und eine Sound API.
- Embedded: Die Java 2 Standard Edition besitzt alle Fähigkeiten die man benötigt, um mit der entwickelten Software auch Embedded Systems abdecken zu können. Dazu gehören hohe Hardwareunabhängigkeit, Möglichkeit zur Vorentwicklung auf herkömmlichen PCs, schnelle UI-Entwicklung, Wiederverwendbarkeit von Komponenten und vieles mehr.
- Real-Time System: Das *Java Real-Time System* (*Java RTS*) ist eine Erweiterung der Java APIs und der JVM um echtzeitfähige Applikationen entwickeln zu können. Echtzeitfähig bedeutet, dass der Entwickler Kontrolle über das Zeitverhalten seiner Applikation hat und garantiert werden kann, dass ein Algorithmus in einem genau spezifizierten Zeitintervall ausgeführt wird und zu einem Ergebnis kommt. Es ist also die Fähigkeit vorhersehbar und verlässlich auf Ereignisse aus der „Außenwelt" zu reagieren.

***Enterprise Edition* (*J2EE*):** Diese Edition baut auf der J2SE auf und soll die Entwicklung von Geschäftsapplikationen vereinfachen. Durch die Verwendung von standardisierten modularen Komponenten ist es möglich, viele Details des Programmverhaltens automatisch ohne viel Programmieraufwand einzubinden. Zusätzlich zu den Möglichkeiten der J2SE APIs verfügt die J2EE unter Anderem über eine API zum Erstellen von Servlets, *Java Server Pages* (zur Erstellung von dynamischen Webinhalten) und XML Funktionalität. Die J2EE ist primär darauf ausgerichtet, ein übergeordneter Standard für Geschäftsanwendungen zu sein, der viele bisher übliche Technologien unterstützt und vereint.

***Micro Edition* (*J2ME*):** Mit Hilfe der J2ME wird es möglich, Anwendungen für Consumergeräte wie PDAs, Mobiltelefone, Settop-Boxen oder andere Geräte mit stark begrenzten Speicherkapazitäten zu realisieren. Da diese Geräte oft sehr unterschiedliche Funktionen beinhalten, ist die J2ME kein starres Gebilde von APIs sondern kann von den Geräteherstellern relativ flexibel unterstützt werden. Die Konfiguration der J2ME setzt sich aus aus Konfigurationen, Profilen und optionalen Paketen zusammen. Eine Konfiguration besteht aus einer JVM und einer minimalen Menge an APIs, die unterstützt werden müssen und die Basisfunktionalität

zur Verfügung stellen. Um die Programmausführung auf einem solchen System zu ermöglichen, müssen zusätzlich noch Profile, das sind APIs, die weiterführende Funktionalität bieten, mit der Konfiguration kombiniert werden. Diese Profile bieten Zugriff auf gerätespezifische Funktionalität und engen das Feld der durch die Konfiguration unterstützten Systeme weiter ein. Die optionalen Pakete unterstützen sehr spezifische Funktionalität, wie zum Beispiel Bluetooth, Multimedia oder Webservices und variieren natürlich je nach Endgeräteart.

***Java Card*:** Mittels der Java Card Technologie kann man Applets auf sogenannten Smart-Cards speichern und ausführen und ist kompatibel zu den existierenden Standards für diese Karten. SIM Karten in Mobiltelefonen können zum Beispiel mit dieser Technologie betrieben werden um etwa zusätzliche Services wie remote banking zu ermöglichen. Der Schwerpunkt liegt also hauptsächlich auf sicherer Authentifizierung und transaktionsbasierten Services. Es existieren eigene Spezifikationen für die unterstützten Instruktionen der Java Card Virtual Machine sowie der unterstützten Untermenge der Java Sprache und Dateiformate die die entsprechenden Geräte unterstützen müssen, um Java Card kompatibel zu sein.

Lizenzrechtliches

Das Java Delvelopment Kit (JDK) und Java Runtime Environment (JRE), die beide Bestandteile der J2SE sind, stehen unter Suns *Binary Code License* (*BCL*). Sie sind für den Einsatz auf Desktopcomputern kostenlos direkt von Sun erhältlich, beim Einsatz auf Embedded Devices und anderen Geräten benötigt man allerdings unter Umständen eine Lizenz von Sun. Ansonsten ist es möglich sowohl das JDK als auch das JRE ohne entstehende Kosten zu verbreiten und Software zu schreiben, die darauf zurückgreift.
Der komplette Quellcode des JDK ist unter der *Sun Community Source License* (*SCSL*) verfügbar, zusätzlich wird der Quellcode der momentan aktuellen Version des JDK (5.0 zum aktuellen Zeitpunkt) unter der *Java Research License* (*JRL*) freigegeben, die verglichen zur SCSL einfacher zu verstehen sein soll und sich primär an akademische Einrichtung wendet.

Zusammenfassung

Vorteile:

- Weit verbreitet, daher guter Support bei Problemen. Existiert seit mehr als zehn Jahren und ist mittlerweile sehr ausgereift.
- Umfangreiche APIs für alle erdenklichen Einsatzgebiete
- Wurde von Anbeginn der Entwicklung auf Plattformunabhängigkeit ausgelegt
- Logisches, durchdachtes Konzept, dadurch einfach zu erlernen

Nachteile:

- Im Vergleich zu C und C++ eine verhältnismäßig neue Technologie
- Schwierigere Integration in bestehende Umgebungen die nicht Java als Codebasis haben, da Java eine komplett neue Sprache mitsamt Bibliotheken und keine API zu einer bestehenden Sprache darstellt.

Verwendbarkeit für den CSPmedclient

Aufgrund der vielen Vorteile die Java auf den ersten Blick bietet, wäre eine Neuimplementierung des CSPmedclients in dieser Sprache natürlich eine Lösung, die alle Anforderungen erfüllen könnte. Man muss hierbei aber Bedenken, dass Java einige konzeptionelle Unterschiede zu vielen anderen in diesem Vergleich angeführten Alternativen aufweist. Dies bewirkt, dass der CSPmedclient eigentlich neu implementiert werden muss, die Weiterverwendung von Code oder Klassenhierarchien aus dem bestehenden CSPmedclient-Code ist nur in seltenen Fällen möglich. Vor allem die Kommunkation eines in Java implementierten CSPmedclients mit der Serverapplikation stellt sich als schwierig dar, es gibt jedoch zwei prinzipielle Lösungsmöglichkeiten.
Erstens könnte man die bestehende libcsprpc Bibliothek weiterverwenden und so erweitern, dass als shared library kompiliert und zu dem CSPmedclient gelinkt werden kann um eine Einbindung unter Benutzung des JNI zu ermöglichen. Der Aufwand und die Realisierbarkeit dieses Ansatzes ist nicht leicht abzuschätzen, man sollte sich aber auch die Frage stellen, ob dies ein wünschenswerter Weg ist. Java stellt ausgereifte Methoden zur Netzwerkkommunikation und auch zu RPC zur Verfügung, daher wäre die zweite Möglichkeit, alternativ zum libcsprpc-Aufruf per JNI, die libcsprpc Bibliothek nicht mehr zu Verwenden und stattdessen zu Versuchen, eine Möglichkeit zu finden, wie das WMS auf andere Weise (unter Benutzung von RPC) mit dem Java-CSPmedclient kommunzieren kann.
Betrachtet man Java genauer, fallen bald auch die Unterschiede zu den anderen Toolkits im Hinblick auf das Eventsystem auf. Dadurch, dass man gezwungen ist, genau einen Eventloop zu verwenden (den, den die JVM ausführt), ist es notwendig, ein neues Konzept zur Abbildung der WMS-Aufrufe auf das Java-Eventsystem zu erstellen. Dies ist auf jeden Fall weitaus aufwändiger, als das bestehende Konzept, das der unter Benutzung der MFC implementierte CSPmedclient verwendet, auf die verwendete API zu adaptieren.
Aufgrund dieser doch sehr umfangreichen Neuentwicklungen, die man hinsichtlich Kommunikation mit der Serverapplikation durchführen müsste, muss abgewogen werden, ob dieser Aufwand gerechtfertigt ist. Andere Toolkits, die zum Beispiel auf C++ basieren, können hinsichtlich der Darstellung einer Applikation auf unterschiedlichen Betriebssystemen Ähnliches leisten wie Java. Natürlich wäre eine Implementierung in Java zum gegenwärtigen Zeitpunkt die modernste und auch „sauberste" Lösung, das Konzept des WMS und des CSPmedclients könnten allerdings die vielen Vorteile von Java nicht ausnutzen. Vor allem existieren in Java

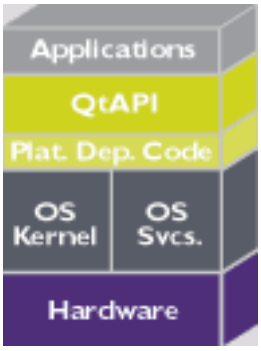

Abbildung 5: Qt-Architektur

andere Möglichkeiten bei der plattformunabhängigen Darstellung von Formularen, die in 10.3.2 beschrieben sind. Um diese verwenden zu können, wären weitere umfangreiche Änderungen am WMS notwendig, genauso wie bei der Verwendung eines anderen Toolkits, das diese Möglichkeiten zur Formulardarstellung bietet.
Die Hauptaufgabe des CSPmedclients ist die ansprechende Aufbereitung der Oberfläche des Serverprogramms und zur Zeit existieren keine Absichten, diese Funktionalität maßgeblich zu erweitern. Der Einsatz von Java für den CSPmedclient ist zwar hinsichtlich der von Java gebotenen Flexibilität und Leistungsfähigkeit sehr verlockend, doch der dafür zu betreibende Aufwand ist als weitaus höher einzuschätzen als bei der Verwendung eines plattformübergreifenden C++ Toolkits. Weiters kann der Großteil des Funktionsumfanges von Java durch den CSPmedclient nicht ausgenutzt werden, weshalb meiner Meinung nach der Aufwand einer Neuimplementierung in Java im Verhältnis zum tatsächlichen Nutzen viel zu hoch ist und deshalb einer anderen Option der Vorzug gegeben wird.

5.2.3 Qt

Qt ist ein plattformübergreifendes Toolkit, das es ermöglicht mit C++ unter anderem Applikationen mit grafischer Benutzeroberfläche zu realisieren. Es wird von der norwegischen Firma Trolltech [3] entwickelt und in etlichen Open-Source Projekten erfolgreich eingesetzt. Das Toolkit wurde mit dem Ziel entwickelt, plattformunabhängige Client und Server Applikationen einfach und intuitiv zu erstellen. Dies wird, ähnlich wie bei andern Toolkits, dadurch erreicht, dass zwischen Applikation und Betriebssystem eine neue Schicht eingeführt wird, die Qt-API. Diese besteht aus genau dokumentierten und standardisierten Methoden, die auf die jeweilige Plattform, auf der die Applikation ausgeführt wird, umgesetzt werden (siehe Abbildung 5).
Mit Qt erstellte Applikationen laufen nativ mit dem gleichen Sourcecode auf den folgenden Plattformen:

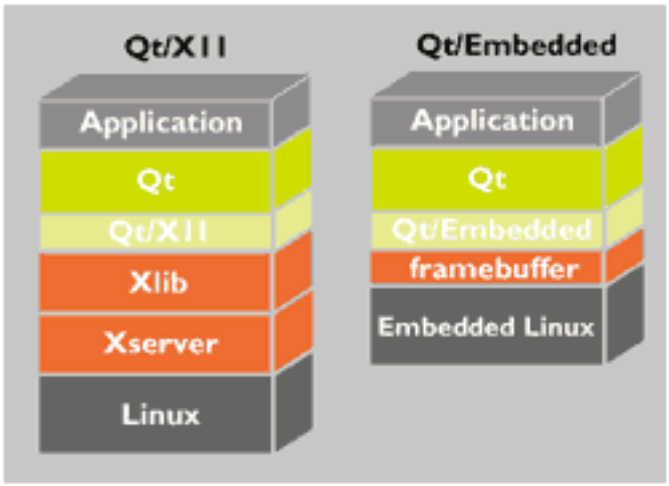

Abbildung 6: Architektur des Qt/Embedded Systems im Vergleich zur „herkömmlichen" Qt/X11 Variante

Qt/Windows: Microsoft Windows XP, 2000, NT 4, Me/98/95. Zur Darstellung von grafischen Objekten werden keine zusätzlichen Layer benötigt, da zum Beispiel unter Microsoft Windows die grundlegende Windows API zur Darstellung verwendet wird.

Qt/X11: Linux, Solaris, HP-UX, AIX und andere Unix-Varianten. Auch unter Unix wird kein zusätzlicher grafischer Layer benötigt, zur Ausgabe von Grafikobjekten wird direkt die grundlegende Xlib Bibliothek verwendet.

Qt/Mac: Mac OS X. Jede mit Qt erstellte Applikation läuft ohne Sourcecodeänderungen auch unter Mac OS X, inclusive dem entsprechenden Aqua „Look-and-Feel".

Qt/Embedded: Diese Variante ist dafür gedacht, die gleichen Applikationen, die unter einer anderen Plattform entwickelt wurden, auf Geräten wie zum Beispiel Mobiltelefonen, Set-Top-Boxen oder elektronischen Kiosksystemen auszuführen. Um dies zu erreichen, muss man besonders auf den Speicherbedarf des Toolkits achten, der nicht zu groß sein darf, da die Endgeräte der angesprochenen Bereiche oft über eine sehr beschränkte Speicherausstattung verfügen. Aus diesem Grund hat man sich entschieden, das X-Server/Xlib System zu vermeiden und stattdessen die Grafikausgabe direkt über den Framebuffer durchzuführen. Abbildung 6 soll dies verdeutlichen.
Das Ziel des Qt-Toolkits besteht darin, das darunterliegende Fenster- und Betriebssystem zu abstrahieren und dem Programmierer ein logisches, objektorientiertes Interface zu diesem zu bieten. Die Qt-API ist auf allen unterstützten Plattformen

einheitlich, was plattformunabhängiges Entwickeln sehr erleichtert, da die Programme nativ unter den verschiedenen Plattformen von ein- und derselben Codebasis laufen. In der Qt-API sind umfangreiche Methoden enthalten, die über das bloße Entwickeln von Benutzerschnittstellen weit hinausgehen. So sind etwa primitive Datentypen definiert (boolsche Typen, verkettete Listen usw.), und auch der Zugriff auf Datenbanken von Oracle oder zum Beispiel auf den Microsoft SQL Server wird über Qt-eigene Funktionaliät ermöglicht. Weiters wird auch 2D und 3D Grafik unterstützt, letztere mit Hilfe der OpenGL Bibliothek.

Kommunikation zwischen den Objekten

Eine Besonderheit ist die Verwendung von „signals" und „slots" (siehe 5.1.5.3), die auf einfache Art und Weise die Kommunikation zwischen einzelnen Objekten ermöglicht. Um die Lauffähigkeit der mit der hauseigenen Klassenbibliothek Qt erstellten Programme auf mehreren Plattformen zu gewährleisten, definiert Qt an vielen Stellen Makros, die es in C++ normalerweise nicht gibt. Bevor der C++-Präprozessor diese expandiert oder entfernt, wird mit Hilfe des sogenannten "Meta Object Compiler" (MOC) ergänzender C++-Quellcode erstellt. Dieser muss ebenfalls übersetzt und dann mit dem gesamten Programm gelinkt werden. Der „Meta Object Compiler" liest alle Headerdateien der Applikation und erzeugt den notwendigen Code, um zum Beispiel den Signal-Slot Mechanismus zu unterstützen. Der Aufruf des MOC erfolgt automatisch über das in der Qt-API enthaltene Programm „qmake" und der Entwickler bekommt normalerweise den automatisch generierten Code weder zu Gesicht, noch muss er ihn editieren. Zusätzlich zum Signal-Slot Mechanismus unterstützt der MOC auch den Übersetzungsmechanismus von Qt. Qt stellt auch Tools zur Verfügung, die ein einfaches und schnelles Übersetzen der Applikation in andere Sprachen erlauben. Der Programmierer markiert im Quellcode einfach den Text der übersetzt werden soll und ein Tool extrahiert diesen Text aus dem Quellcode. Mit Hilfe einer grafischen Applikation werden dann die zur Übersetzung vorgemerkten Texte samt Kontextinformation aufbereitet und falls die Übersetzung abgeschlossen ist, ein sogenanntes *translation file* erzeugt, das in der Applikation verwendet wird.

Lizenzrechtliches

Um die Qt-API benutzen zu können hat man die Wahl zwischen vier Lizenzen:

- Kommerzielle Lizenz
- Open Source Lizenz
- Ausbildungslizenz
- Akademische Lizenz

Die kommerzielle Lizenz muss erworben werden, wenn man die Absicht hat, kommerzielle Software oder Software dessen Quellcode man nicht mit veröffentlichen möchte, mit Hilfe der Qt-API zu erstellen. Weiters benötigt man diese Lizenz, wenn man den Qt-Support in Anspruch nehmen möchte. Die Lizenz wird pro Entwickler vergeben und darf nur einmal in sechs Monaten den Entwickler in derselben Organisation wechseln. Es gibt zwei Varianten, die Professional Edition und die Enterprise Edition. In der Enterprise Edition sind zusätzlich die Qt-Solutions enthalten, das sind weitere Tools und Komponenten, sowie Datenbank- und Netzwerkmodule und einiges mehr (siehe Trolltech Webpage [3]).
Von der Open Source Lizenz existieren zwei Ausprägungen, die GPL und die QPL Lizenz. Beide Lizenzen können zur Entwicklung von Applikationen verwendet werden, die unter Linux, Unix und MacOS X lauffähig sind. Die Open Source Lizenzen können frei kopiert und in Umlauf gebracht werden, jedoch wieder nur unter der gleichen Lizenz in der sie erhalten wurden. Dies stellt einen signifikanten Unterschied zur kommerziellen Lizenz dar, die man überhaupt nicht verbreiten darf. Die Open Source Lizenzen werden ohne Garantie und Support vergeben, bei Benutzung einer solchen Lizenz ist man dazu verpflichet, auch den Quellcode der Applikation unter der GPL beziehungsweise QPL zur Verfügung zu stellen. Umgekehrt muss man eine kommerzielle Lizenz kaufen, wenn man einen kommerziellen Vorteil aus einer mit Hilfe von Qt entwickelten Applikation ziehen möchte, deren Quellcode man nicht unter einer Open Source Lizenz freigibt.
Die Ausbildungslizenz umfasst 100 Qt/Windows Lizenzen mit derselben Funktionalität wie die kommerzielle Lizenz (außer den Zugriff auf Qt-Solutions). Lizenznehmer muss eine Schule, Universität oder andere akademische Institution sein, man darf mit dieser Lizenz keine kommerzielle Software entwickeln und auch technischer Support und Qt-Quellcode wird nicht zur Verfügung gestellt.
Die akademische Lizenz wird für alle unterstützten Plattformen angeboten und ist eine verbilligte Version der kommerziellen Lizenz. Auch mit dieser Lizenz darf man keine Software entwickeln, die kommerzielle Interessen verfolgt.

Zusammenfassung

Die Vorteile der Qt-API sind wie folgt:

- Produktivität: Die Entwickler müssen sich nur mit einer API vertraut machen und nicht auf Eigenheiten des entsprechenden Zielsystems eingehen.
- Kosten/Nutzen: Es muss nur eine Codebasis gepflegt werden, was Kosten die für Wartung senkt und die Stabilität der Applikation erhöht.
- Funktionsumfang: Die Qt-API bezieht sich, im Gegensatz zu Microsofts MFC, nicht nur auf die Benutzerschnittstellenentwicklung, sondern auch auf zum Beispiel einheitliches Filehandling, Netzwerkzugriff, Prozesshandling, Threads, Datenbankzugriff und einiges mehr.

- Look & Feel: Die Applikationen sehen auf allen unterstützten Plattformen so aus, als wären sie nativ für dies Plattform programmiert. Die Qt-API nimmt nämlich intern eine Umsetzung der API-Aufrufe auf das native UI-System vor.
- Performance: Es wird kein Runtimeenvironment, Virtuelle Maschine oder Emulationsschicht benötigt. Deshalb laufen Qt-Applikationen auch genauso schnell wie native Applikationen. Ebenso braucht man keinen X-Server um mit Qt erstellte Applikationen unter Microsoft Windows ausführen zu können.
- Qualität: Die API ist für viele verschiedene Betriebssysteme und Compiler gut getestet. Sie wird bereits von vielen kommerziellen aber auch non-profit Organisationen eingesetzt und ist daher verhältnismäßig ausgereift und robust.
- Portierungsfreundlich: Eine relativ einfache Portierung von MFC Applikationen auf die Qt-API ist möglich, da Qt und MFC ähnliche Features aufweisen und beide C++ Toolkits sind. Dies begünstigt die Wiederverwendung von Codeteilen im Vergleich zu einer Portierung auf Java oder .NET beziehungsweise C#.
- Support: Dadurch dass Qt von einer kommerziellen Organisation entwickelt und vertrieben wird, steht kompetenter und schneller Support zur Verfügung. Aufgrund der mittlerweile weiten Verbereitung und Verwendung des Toolkits ist es aber auch einfach auf Websites und in Foren Lösungen zu seinen Problemen zu finden.

Nachteile:

- Da Qt von einer kommerziellen Organisation vertrieben wird, ist auch der Quellcode dieser Bibliothek geschützt und es fallen Kosten zur kommerziellen Nutzung an. Details dazu siehe weiter unten unter „Lizensierung”.
- Signal-Slot Mechanismus ist im Prinzip eine Erweiterung der C++ Sprache, es wird ein eigener Präprozessor (MOC) benötigt

Verwendbarkeit für CSPmedclient

Die technischen Merkmale der Qt-API sind hervorragend und erfüllen alle Anforderungen, die an den plattformunabhängigen CSPmedclient gestellt werden. Auch die gute Integration in bestehende Entwicklungsumgebungen und die Möglichkeit einer schrittweisen Migration von MFC zu Qt ist ein großer Vorteil. Für den kommerziellen CSPmedclient ist es natürlich notwendig, mindestens eine kommerzielle Lizenz der Qt-Enterprise Edition zu erwerben. Dies ist schon alleine aus dem

Grund notwendig, da die Open Source Lizenz nicht für Microsoft Windows angeboten wird. Die Enterprise Edition der Qt-API schlägt mit 2290 Euro zu Buche, zusätzlich zu den zu leistenden Programmieraufwand für die Portierung. Im Hinblick darauf, dass es freie Open Source Toolkits mit ähnlichem Leistungsumfang gibt, mit denen es möglich ist, ohne Lizenzzahlungen kommerzielle Software zu entwickeln, habe ich mich gegen den Einsatz von Qt für den CSPmedclient entschieden.
Natürlich ist es zu kurzsichtig, alleine nach den Kosten für die Lizenzen zu entscheiden, da es möglich wäre, mit der Qt-API aufgrund ihrer Ausgereiftheit und dem guten Support insgesamt schneller und mit weniger Aufwand die Portierung durchführen zu können. Diese beiden Eigenschaften sind allerdings vor dem konkreten Einsatz des APIs und des Beginns des Projektes schwer abzuschätzen, man kann sich höchstens an Erfahrungsberichten anderer Projekte orientieren. Faktum ist, dass für das primäre Ziel, der Portierung des CSPmedclients auf GNU/Linux mit Hilfe des Qt-APIs, ziemlich sicher genausoviel Aufwand betrieben werden muss wie mit seinem schärfsten Konkurrenten in diesem Vergleich, wxWidgets (das unter 5.2.7 vorgestellt wird).
Weiters ist zu bedenken, dass die Tatsache dass Qt von einer kommerziellen Organisation gepflegt wird, keine Garantie für dessen Zukunftssicherheit darstellt. Dies ist im Übrigen auch nicht der Fall wenn ein Softwareprojekt auf Open Source Basis entwickelt wird, doch kann man in diesem Fall davon ausgehen, dass allgemein verständliche Dokumentation vorliegt sodass man notwendige Änderungen weiterhin vornehmen kann, sollte das Projekt aufgegeben werden. Mit dem Kauf einer kommerziellen Lizenz begibt man sich wiederum in die Abhängigkeit eines Herstellers, von der man ja eigentlich durch die Portierung auf ein anderes Toolkit beziehungsweise Betriebssystem wegkommen wollte, insofern habe ich mich dafür entschieden, Qt für die Realisierung des neuen CSPmedclients nicht zu verwenden, ungeachtet der verlockenden technischen Daten und Möglichkeiten.

5.2.4 GTK+

GTK+ ist ein Mutliplattform-Toolkit zur Erstellung grafischer Benutzeroberflächen. Das Toolkit wurde von Grund auf für die Unterstützung möglichst vieler Programmiersprachen ausgelegt, man kann GTK+ unter Anderem mit folgenden Programmiersprachen benutzen:

- C/C++/C#
- Perl
- Python
- Java
- PHP

- u.v.m

GTK+ ist offiziell verfügbar unter vielen UNIX-Derivaten, darunter natürlich auch GNU/Linux für das es ursprünglich entwickelt wurde. Unabhängig davon existieren auch Portierungen auf Microsoft Windows und MacOS X, die jedoch weniger ausgereift und stabil sind. Das Toolkit wurde eigentlich als Widgetbibliothek für GIMP (GNU Image Manipulation Program) entwickelt, wurde aber immer mächtiger und wird heute von einer Vielzahl von Applikationen eingetzt, die bekannteste davon ist vielleicht der GNOME Desktop. GTK+ enthält sowohl primitive Widgets wie Textfelder, Combo- und Dropdownboxen als auch komplexe wie zum Beispiel Dialoge zur Datei- und Farbauswahl. Das GTK+ Toolkit basiert auf den folgenden Bibliotheken:

- GLib ist eine low-level Bibliothek, die die Basis von GTK+ und GNOME bildet. Sie stellt primitive und komplexe Datentypen zur Verfügung wie zum Beispiel bool'sche Typen oder verkettete Listen. Weiters bietet sie Schnittstellen zu runtime-Funktionalität wie Eventloops, Threads, dynamisches Laden und ein Objektsystem.
- Pango ist eine Bibliothek zum Layout und Rendering von Text, mit Schwerpunkt auf Internationalisierung. Sie bildet den Kern der Text- und Schriftverwaltung von GTK+-2.0
- Die ATK Bibliothek unterstützt erweiterte Zugriffsmethoden. Falls eine Applikation auf ATK-Schnittstellen zugreift, kann sie auch zum Beispiel mit Hilfe alternativer Eingabegeräte benutzt werden oder eine automatische Vergrößerung von Bildschirminhalten unterstützen. Diese Methoden sind vor allem für Benutzer mit Behinderungen von Bedeutung.

Callbackmechanismus

GTK+ ist ein eventgesteuertes Toolkit, was bedeutet, dass, falls die Applikation auf Benutzereingaben reagieren soll, die Methode `gtk_main()` aufgerufen wird. In dieser Methode wird eine Schleife durchlaufen die die entsprechende Callbackfunktion aufruft, falls eine solche für die aktuelle Benutzereingabe definiert ist.

Lizenzrechtliches

GTK+ ist als Teil des GNU Projektes freie Software und unter den Bedingungen der GNU LGPL verfügbar. Dies erlaubt auch die Verwendung zur Erstellung proprietärer, kommerzieller Software mit Hilfe dieses Toolkits.

Zusammenfassung

Vorteile:

- Weite Verbreitung, daher auf fast allen Systemen bereits installiert.
- Vielzahl an unterstützten Programmiersprachen.
- Umfangreiche Funktionen, nicht nur für die GUI Programmierung.

Nachteile:

- Da das GTK+ Toolkit eigene Widgets besitzt, sehen die damit erzeugten Applikationen anders aus als „native" Versionen zum Beispiel unter Microsoft Windows oder MacOS X. In der Geschwindigkeit der Anzeige besteht demzufolge auch ein geringer Nachteil gegenüber Applikationen, die etwa direkt die Win32 API verwenden.
- Portierung auf Microsoft Windows noch nicht sehr ausgereift.

Verwendbarkeit für den CSPmedclient

Unter GNU/Linux spricht Vieles für den Einsatz der GTK+ Bibliothek für den CSPmedclient. Vor allem die LGPL Lizenz unter der GTK+ veröffentlicht ist und die Tatsache dass es sehr weit verbreitet und oft verwendet ist, sind die wichtigsten Argumente für die Entscheidung für dieses Toolkit. Für die Entwicklung von Applikationen unter Microsoft Windows scheint GTK+ allerdings nicht so gut geeignet, da man hier den Nachteil hat, dass das Aussehen nicht dem von nativen Windowsapplikationen entspricht. Außerdem ist die GTK+ Version für Microsoft Windows zum gegenwärtigen Zeitpunkt noch sehr instabil und vom Performancestandpunkt bei weitem nicht so schnell wie ein natives Windows-GUI. Für den CSPmedclient sollte also idealerweise GTK+ unter GNU/Linux eingesetzt werden und die native Win32 API unter Microsoft Windows.

5.2.5 FLTK

FLTK (ausgessprochen „fulltick") ist ein plattformübergreifendes C++ Toolkit, das für folgende Plattformen zur Verfügung steht:

- UNIX/Linux und ähnliche Systeme für die ein XServer verfügbar ist
- Microsoft Windows in den Versionen 95, 98, ME, NT 4.0, 2000 und XP
- MacOS X

Es ist besonders darauf ausgelegt, kompakt zu sein um sich besser dazu zu eignen, statisch zu Applikationen gelinkt zu werden. FLTK wurde ursprünglich dazu entwickelt, kompatibel zur Forms Library auf SGI Maschinen zu sein, bei der alle Struktur- und Funktionsbezeichnungen mit `fl_` beginnen. Diese Konvention wurde auch bei der C++ Bibliothek beibehalten und bildete daher auch den Namen derselbigen. Aufgrund praktischer Überlegungen wurde die Bibliothek später von

FL in FLTK umbenannt und eine Bedeutung (*Fast Light Tool Kit*) für dieses Akronym erfunden. Die Entwicklung von FLTK begann vor jener von GTK+ und Qt, das Projekt wurde aber erst 1998 öffentlich verfügbar.
Zur Ausgabe der Widgets unter den verschiedenen Plattformen wird die jeweils native Methode verwendet, also Xlib unter X11, WIN32 unter Microsoft Windows und Carbon unter MacOS. FLTK ähnelt in dieser Hinsicht also vielen anderen Toolkits, die ebenfalls die nativen Möglichkeiten zum Zeichnen von Widgets der jeweiligen Plattform nutzen. Es fungiert daher als Schicht zwischen Applikation und Betriebssystem und lediglich ungefähr 10% des Quellcodes von FLTK sind auf den einzelnen Plattformen unterschiedlich. Bemerkenswert ist jedoch, dass nicht die Standardsteuerelemente für die Dialoge verwendet werden, sondern das Toolkit seine eigenen Buttons, Textfelder usw. mit dem damit einhergehenden einheitlichen Aussehen auf allen Plattformen verwendet.

Eventbehandlung

Hinsichtlich der Eventbehandlung findet man in FLTK große Ähnlichkeiten zu anderen Toolkits. Auch hier muss am Ende der `main()` Methode „manuell" ein Eventloop gestartet werden. Konkret bewerkstelligt man die durch Aufruf der Methode `Fl::run()` die das Gleiche bewirkt wie die in 5.2.4 beschriebene Methode `gtk_main()` des GTK+ Toolkits. Um festzulegen, welche Aktionen bei welchem Event auszuführen sind, bedient sich FLTK eines relativ einfachen Callbackmechanismus. Hat man zum Beispiel ein Objekt `button` der Klasse `Fl_Button` definiert, kann man mittels

```
button->callback(xyz_callback, &xyz_data);
```

die Methode festlegen, die nach einem Event angesprungen werden soll. Der Event der als Auslöser dienen soll wiederum wird folgendermaßen festgelegt:

```
button->when(FL_WHEN_CHANGED);
```

Die Callbackfunktion muss eine statische Klassenmethode sein und hat die folgende Signatur:

```
void xyz_callback(Fl_widget *w, void *data)
{
...
}
```

Als Parameter `w` wird dabei ein Zeiger auf das Objekt übergeben, das der „Verursacher" des Events ist. Man bemerkt, dass dieser Ansatz nicht die Möglichkeiten einer objektorientierten Sprache ausnutzt, die Übergabe des Zeigers auf ein `Fl_widget` Objekt würde entfallen, wenn man die Callbackmethode als Mitglied des Objekts `button` implementieren könnte. Vermutlich liegt der Grund in dieser Lösung einfach darin, dass die XForms Bibliothek, zu der FLTK ursprünglich kompatibel sein wollte, nicht objektorientiert war und man beim Umstieg auf eine objektorientierte Sprache einfach das Callbackkonzept beibehielt. Die Nachteile dieser Lösung liegen in der Übergabe der Zeiger, die nicht typsicher ist und dadurch

eine immanente Fehlerquelle darstellt. Weiters wird die funktionale Kapselung von Methoden und Daten, die ja ein Vorteil der objektorientierten Programmierung ist, durch diesen Mechanismus wieder ausgehebelt.
In der Version 2.0, die eine größtenteilige Neuimplementierung von FLTK ist, wurde der Mechanismus des Eventhandlings, überarbeitet, es existiert jetzt die Funktion `handle()` in der `Widget` Klasse, die aufgerufen wird, falls ein Event an das Objekt gesendet wird. Die Version 2.0 befindet sich allerdings noch in Entwicklung, aktuell stabil und unterstützt sind die Versionen 1.1.x, die das oben erwähnte Callbackkonzept verwenden.

Erstellung einer Benutzerschnittstelle

Da das FLTK ziemlich kompakt ist, existieren auch wenig Möglichkeiten zum Gestalten von Benutzerschnittstellen. Techniken wie Sizer oder Layoutmanager, die ein überlappungsfreies Darstellen von Dialogelementen ermöglichen, existieren nicht, die Positionen und Größen der Elemente werden in Koordinaten und Pixel angegeben. Es kann allerdings relativ detailliert festgelegt werden, wie sich die Elemente bei Größenänderungen des umgebenden Fensters verhalten sollen.
Die Benutzerschnittstellen können entweder direkt im Code erzeugt werden oder grafisch mit Hilfe des Programms *FLUID*, das Header- und Implementierungsdateien automatisiert erstellen kann, die man dann um die gewünschte Funktionalität ergänzt.

Lizenzrechtliches

FLTK basiert auf der GNU Library General Public License (LGPL), jedoch mit einigen Ausnahmen:

- Änderungen des FLTK Konfigurationsskripts, der Konfigurationsheaderdatei und der Makefiles mit dem Zweck, eine spezielle Plattform zu unterstützen, stellen keine Modifikation oder abgeleitete Arbeit im Sinne der LGPL dar. Derart veränderte FLTK Bibliotheken müssen dem FLTK Projekt jedoch bekanntgegeben werden.
- Widgets, die von FLTK klassenhierarchisch abgeleitet werden, stellen keine Modifikation oder abgeleitete Arbeit im Sinne der LGPL dar.
- Statisches Linken von Applikationen und Widegets gegen die FLTK Bibliothek stellt eine Modifikation oder abgeleitete Arbeit im Sinne der LGPL dar und zwingt den Autor weder den Quellcode der Applikation oder des Widgets offenzulegen, noch FLTK als shared library zu verwenden oder die Applikation beziehungsweise das Widget gegen eine vom Benutzer zur Verfügung gestellte Version von FLTK zu linken. Falls die Applikation oder das Widget gegen eine modifizierte Version der FLTK gelinkt wird, müssen diese Modifikationen unter den Bedingungen der LGPL, §§1, 2 und 4 bereitgestellt werden.

- Es ist weder notwendig, eine Kopie der Lizenz von FLTK mit Applikationen die gegen sie gelinkt sind, zur Verfügung zu stellen, noch ist es notwendig die FLTK Lizenz in den Applikationen oder deren Dokumentation zu erwähnen. Es muss jedoch die Tatsache, dass FLTK verwendet wird, in der Programmdokumentation angegeben sein.

Abgeleitete Arbeiten von unter der LGPL stehender Software müssen ihrerseits wiederum unter der LGPL veröffentlicht, sowie die vorgenommenen Änderungen deutlich gekennzeichnet werden. Weiters muss die abgeleitete Arbeit selbst wieder eine Bibliothek sein und noch einige zusätzliche Kriterien erfüllen, die in [26], §2 dokumentiert sind.
Lizenzrechtlich ist es also absolut kein Problem, proprietäre Software unter Verwendung von FLTK zu entwickeln, deren Quellcode nicht zugänglich sein soll.

Zusammenfassung

Vorteile:

- Schlanke Bibliothek, produziert kompakte ausführbare Dateien
- OpenGL Unterstützung
- Generische Widgets vereinfachen koordinatenbasierte Platzierung

Nachteile:

- Verhältnismäßig geringe Verbreitung
- Klassen decken ausschließlich den GUI-Bereich ab
- Generische Widgets verhindern optische Konsistenz mit nativen Applikationen des verwendeten Betriebssystems
- Callbackkonzept
- Brauchbare Dokumentation schwer zu finden
- Themeunterstützung erst in Version 2.0

Verwendbarkeit für den CSPmedclient

Hinsichtlich der für das GUI-Design bereitgestellten Funktionalität sollte eine Verwendung zur Implementierung des CSPmedclients mit FLTK prinzipiell möglich sein. Dabei ist die Verwendung von generischen Widgets, das heißt von Widgets, die von der FLTK Bibliothek erzeugt werden und auf allen Plattformen identisch aussehen, sowohl als Vor- als auch als Nachteil zu werten. Der Vorteil liegt darin, dass man die Widgets pixelgenau platzieren kann und man sich nicht darum kümmern muss, wie sie auf den anderen Plattformen dargestellt werden. Andererseits

fügt sich eine solche Applikation nicht harmonisch in das Aussehen der jeweils verwendeten Oberfläche ein, sie wirkt durch das unterschiedliche Design wie ein Fremdkörper. Dieser Nachteil ist aber bezogen auf den CSPmedclient als eher untergeordnet einzuschätzen, da erstens die Oberfläche des existierenden CSPmedclients durch ihre Herkunft vom Terminal schon nicht mehr zeitgemäß ist und es zweitens ein zweifellos höherer Aufwand ist, eine einwandfreie Positionierung von nativen Widgets sicherzustellen.
Nichtsdestoweniger hat FLTK einige Eigenschaften, die gegen eine Verwendung zur Implementierung des CSPmedclients sprechen. Das größte Manko ist seine geringe Verbreitung und damit die Schwierigkeit, gute Dokumentation und Support zu erhalten. Die etwas verstaubt anmutende Eventbehandlung mittels Callbackfunktionen in den 1.1.x Versionen ist ebenfalls eine gravierende Schwachstelle und die Verwendung der 2.0 Version ist nicht empfehlenswert, da sie derzeit noch nicht als stabil gekennzeichnet ist und deshalb Dokumentation noch schwieriger aufzutreiben ist. FLTK bietet im Vergleich zu anderen Toolkits auch hinsichtlich der GUI-Elemente weniger Möglichkeiten zur Gestaltung, was sich eventuell bei einer späteren Erweiterung des CSPmedclients als durchaus problematisch herausstellen könnte. Auf jeden Fall benötigt man für eine Implementierung des CSPmedclients zusätzlich noch eine plattformübergreifende Bibliothek zur Realisierung von Threads, dynamischen Listen und all der Funktionalität, die nicht durch FLTK abgedeckt wird. Daher wäre ein Toolkit vorzuziehen, das alle für den CSPmedclient benötigten Funktionen bereits enthält.
Zusammenfassend eignet sich FLTK wohl eher für kleine Projekte, bei denen es vor allem auf Kompaktheit ankommt und darauf, schnell ein funktionsfähiges Programm mit grafischer Benutzerschnittstelle erzeugen zu können. Bei größeren Projekten wie zum Beispiel den CSPmedclient bin ich der Meinung, dass man auf besser dokumentierte Toolkits zurückgreifen sollte, die auch von kommerziellen Organisationen eingesetzt werden und einen ausgereifteren Eindruck machen.

5.2.6 FOX-Toolkit

Das *FOX*-Toolkit ([9], FOX steht für *Free Objects for X*) wurde mit dem Ziel entwickelt, einfach und effektiv grafische Benutzerschnittstellen implementieren zu können. Ursprünglich war FOX als C++ Toolkit gedacht, doch zum gegenwärtigen Zeitpunkt existieren auch Sprachbindings für Python, Ruby und Eiffel. Folgende Plattformen werden unterstützt:

- Microsoft Windows in den Versionen 9x, ME, NT 4.0, 2000, XP
- Linux und andere Unix-ähnliche beziehnungsweise Unix-basierende Systeme, also auch Free-BSD, SGI IRIX (ab 5.3), SunOS/Solaris, HP-UX (9.x und 10.x), IBM AIX (4.2), MacOS X (unter Verwendung eines X-Servers) und viele mehr.

Anhand dieser Liste erkennt man, dass im Prinzip „nur" zwei konzeptionell unterschiedliche Betriebssysteme unterstützt werden und auch nur zwei Versionen des Toolkits, eine für Microsoft Windows und eine für Linux/Unix, zum Download zur Verfügung stehen. Das hat seinen Grund in der Tatsache, dass das FOX-Toolkit zur Darstellung der Steuerelemente, wie FLTK auch, generische Widgets verwendet. Zu deren Darstellung werden die jeweils zur Darstellung von Grafikprimitiven zuständigen grundlegenden Betriebssystembibliotheken verwendet, die bei Microsoft Windows und den Systemen für die XServer zur Ausgabe eingesetzt werden, natürlich unterschiedlich sind, was in den beiden Versionen des FOX-Toolkits berücksichtigt ist.
Was das Konzept des Toolkits angeht, stellt es gewissermaßen eine Mischung aus FLTK und wxWidgets dar. Genauso wie FLTK wurde es mit Hauptaugenmerk auf die Implementierung einer Benutzerschnittstelle mit Hilfe generischer Widgets entwickelt und beinhaltet daher auch nur sehr wenige über diese Funktionalität hinausgehenden Klassen. Hinsichtlich Eventbehandlung stellt man viele Ähnlichkeiten zu wxWidgets fest und in beiden Toolkits existieren Layoutmanager-Klassen zur plattformunabhängigen und einheitlichen Darstellung von Eingabemasken ohne Verwendung absoluter Elementkoordinaten.

Eventbehandlung

Hinsichtlich Eventbehandlung wird das bereits in 5.1.5 vorgestellte Konzept des Messagehandlings mit spezifiziertem Ziel verwendet. Die Nachrichten werden von einem Sender zu einem Empfänger weitergeleitet, der sie verarbeitet oder seinerseits wieder an ein anderes Objekt weiterleiten kann. Im Gegensatz zu wxWidgets erfolgt diese Weiterleitung nicht automatisch in die nächsthöhere Hierarchiestufe, sondern muss explizit implementiert werden.
Vergleichbar mit GTK+ und FLTK wird am Ende der `main()` Funktion ein Eventloop mittels `application->run()` gestartet, wobei `application` die eigentliche, von `FXApp` abgeleitete Applikation darstellt.

Erstellung einer Benutzerschnittstelle

Wie bei anderen Toolkits auch gibt es bei FOX eine Klassenhierarchie die alle für die Entwicklung einer grafischen Oberfläche benötigten Elemente, also zum Beispiel Fenster, Steuerelemente oder Kommunikationsobjekte umfasst. Auf einem Fenster können Layoutmanager (Näheres dazu auch in 10.3.2) positioniert werden auf denen wiederum die einzelnen Steuerelemente platziert sind. Auf diese Art ist es möglich, Formulare zu gestalten, deren Elemente ihre Größe dynamisch verändern können, etwa dann wenn der Benutzer das umgebende Dialogfenster vergrößert. Natürlich ist es auch weiterhin möglich, die Position der Elemente eines Formulares mit Hilfe von Koordinaten festzulegen.
Bei der Entwicklung einer Benutzerschnittstelle mit dem FOX Toolkit ist man darauf angewiesen, alle Elemente „von Hand" zu setzen. Es existieren zwar Ansätze

von Tools mit deren Hilfe man Oberflächen interaktiv erstellen kann, diese sind jedoch noch nicht besonders ausgereift beziehungsweise generieren Code in der Programmiersprache Ruby.

Lizenzrechtliches

Das FOX-Toolkit ist unter der GNU LGPL veröffentlicht, womit sichergestellt wäre, es auch zur Entwicklung proprietärer Software verwenden zu können. Tatsächlich basieren viele kommerzielle Projekte auf FOX, eine Liste dieser Applikation ist unter [10] verfügbar.

Zusammenfassung

Vorteile:

- relativ kompakt und schnell
- Unterstützung plattformunabhängiger Layouts
- Generische Widgets vereinfachen koordinatenbasierte Platzierung

Nachteile:

- Außer auf der Microsoft Windows Plattform wird ein X-Server zur Darstellung benötigt
- Generische Widgets verhindern optische Konsistenz mit nativen Applikationen des verwendeten Betriebssystems
- wenig verbreitet
- Microsoft Windows-Look

Verwendbarkeit für den CSPmedclient

Das FOX-Toolkit ist FLTK sehr ähnlich, jedoch ist bessere Dokumentation dafür verfügbar und es ist auch weiter verbreitet. Die Tatsache, dass bereits einige kommerzielle Applikationen damit entwickelt wurden, verdeutlichen, dass es durchaus ausgereift und brauchbar für den Produktiveinsatz ist. Die zur Verfügung stehenden Darstellungselemente reichen für eine Implementierung des CSPmedclients durchaus aus, jedoch ist FOX genauso wie FLTK ausschließlich für die GUI-Entwicklung ausgelegt und es fehlen zum Beispiel grundlegende Datenstrukturen wie dynamische Listen oder auch Klassen zur Netzwerkkommunikation, Datenbankzugriff und HTML-Behandlung. Falls eine solche Funktionalität benötigt wird, müsste man auf entsprechende andere frei erhältliche Bibliotheken zurückgreifen.

Unterstützte Plattformen	Bemerkung	Bezeichnung
Microsoft Windows	Win32/Win64/WinCE	wxMSW
Unix/GTK+		wxGTK
Unix/Motif und X11 (Xlib)		wxX11
Apple Macintosh	Version 9.x und X	wxMac
MGL	DOS, Linux und von MGL unterstützte Plattformen	wxMGL
OS/2	derzeit Betastatus	wxOS2
PalmOS	derzeit Alphastatus	wxPalmOS

Tabelle 4: Von wxWidgets unterstützte Plattformen

Ein weiterer gemeinsamer Nachteil von FOX und FLTK ist auch, dass durch die Verwendung von generischen Widgets die Besonderheiten der jeweils verwendeten Plattform nicht berücksichtigt werden und sich eine mit Hilfe von FOX implementierte Applikation deutlich von nativen Applikationen unterscheidet. Dies ist zwar eine Erleichterung für die koordinatenbasierte Platzierung der Steuerelemente, wie dies für den CSPmedclient notwendig ist, könnte jedoch manche Benutzer bei der Bedienung der Applikation sehr irritieren. Da zu erwarten ist, dass auch in der GNU/Linux Welt der Trend in Richtung Vereinheitlichung und Standardisierung der grafischen Oberflächen geht, ist es sicher von Vorteil, diese Standards auch mit dem CSPmedclient zu unterstützen, was das FOX-Toolkit aufgrund seiner generischen Widgets nicht bieten kann.
Bezogen auf die Eventbehandlung fallen sehr starke Ähnlichkeiten zu wxWidgets auf und auch der restliche Funktionsumfang des FOX-Toolkits ist eine Untermenge von wxWidgets. Da wxWidgets jedoch unter Anderem auf die GTK+ und die native WIN32 API zur Darstellung der Steuerelemente zurückgreifen kann und auch funktionsmäßig FOX überlegen ist, fiel die Entscheidung gegen die Verwendung des FOX-Toolkits.

5.2.7 wxWidgets

Das Toolkit wxWidgets besteht aus mehreren Bibliotheken, die C++ Applikationen erlauben unter verschiedenen Betriebssystemen mit minimalen Quellcodeänderungen zu laufen. Für jede Zielplattform (z.B. Windows, GTK+, Motif, Mac, siehe dazu Tabelle 4) existiert eine eigene Bibliothek, die sowohl GUI-Funktionalität als auch einige betriebssystemspezifische Funktionen wie zum Beispiel Dateioperationen oder Threadunterstützung zur Verfügung stellt. wxWidgets ist eine Art Framework, das auch sehr viel eingebaute Funktionalität beinhaltet wie etwa grundlegende Datenstrukturen (Strings, Arrays, verkettete Listen). Ähnlich wie das Qt-Toolkit (siehe 5.2.3) stellt wxWidgets API Methoden zur Verfügung, die die nativen Methoden des jeweils zugrundeliegenden Betriebssystems beziehungsweise dessen nativer GUI-API aufrufen.

Das wxWidgets Toolkit macht es möglich, verhältnismäßig schnell Applikationen die mit Hilfe der MFC geschrieben sind, auf andere Plattformen zu portieren, da die APIs syntaktisch einander sehr ähnlich sind. wxMSW setzt die Aufrufe an die wxWidgets API in Aufrufe der Win32 API um und sollte daher einigermaßen zukunftssicher sein, da nicht damit gerechnet wird, dass die Win32 API bald durch einen inkompatiblen Nachfolger ersetzt wird, schließlich greifen auch die .NET Applikationen darauf zurück. Als Entwicklungsumgebung kann unter Microsoft Windows weiterhin Visual C++ eingesetzt werden, mit dem auch der CSPmedclient unter Verwendung der MFC implementiert wurde. Natürlich können, wie unter GNU/Linux bei den wxGTK beziehungsweise wxX11 Versionen, auch freie Compiler verwendet werden, also zum Beispiel der c++ Compiler der gcc (gnu compiler collection) oder dessen Portierung auf Microsoft Windows, Mingw32. wxWidgets kann auch dabei helfen, Memory-Leaks zu lokalisieren und zu beseitigen, wenn man die Applikation im Debugmodus startet.
Durch das sehr flexible Eventsystem von wxWidgets gestaltet sich eine Portierung des CSPmedclients sehr einfach, da Events sowohl zur Kompilationszeit als auch zur Laufzeit des Programms mit ihren entsprechenden Serviceroutinen verbunden werden können. Unterstützung von Threads existiert ebenfalls, mitsamt den notwendigen Klassen zum Synchronisieren mehrerer Threads. Die Benutzerschnittstellenklassen von wxWidgets sind nicht threadsicher, was bedeutet, dass es nicht möglich ist, ohne Weiteres aus zwei oder mehr parallel laufenden Threads heraus auf Methoden zuzugreifen, die für die Darstellung der Benutzerschnittstelle verantwortlich sind, also zum Beispiel die wxButton Klasse. Eine einfach und saubere Lösung dieses Umstandes ist, einen zentralen Thread zu erzeugen, der die Darstellung und Verwaltung der Benutzerschnittstelle übernimmt und der von den anderen Threads Nachrichten bekommt, falls die Benutzerschnittstelle verändert werden soll. Dieses Konzept hilft auch zugleich, die Übersicht in der Applikation zu bewahren und oft ist es auch gar nicht notwendig, einen separaten Thread zu erzeugen, da man vieles auch mit Messages und dem wxWidgets-eigenen Eventsystem lösen kann, zum Beispiel die Implementierung einer `OnIdle()` Methode, die dann aufgerufen wird, wenn sich die Applikation im Ruhezustand befindet.
Umfangreiche Klassen zur Darstellung von Grafiken, sowohl im 2D-Bereich in Form von Grafikprimitiven, als auch im 3D-Bereich in Form der OpenGL Unterstützung sind ein weiterer Grund, warum dieses Toolkit für die Implementierung des CSPmedclients sehr gut geeignet scheint. In vielen Fällen ist es so, dass die Dokumentation der wxWidgets API besser ist, als jene der nativen APIs, wie zum Beispiel GTK+. Die wxWidgets API ist möglichst einfach gehalten und wird in diese Richtung auch laufend weiter verbessert.
Da hinter der wxWidgets API keinerlei finanzielles Interesse steht, ist sie an manchen stellen qualitätsmäßig nicht so ausgereift, wie das manche kommerzielle API ist. Jedoch existieren auch in kommerziellen Produkten Bugs und Ungereimtheiten, die jedoch, falls ausreichend guter Support zur Verfügung steht, mehr oder weniger schnell ausgemerzt werden können. Der Support in Form von Newsgroups und Mailinglisten ist für die wxWidgets API ziemlich gut und für viele Probleme

existieren bald Bugfixes und Workarounds.

Lizenzrechtliches

Die wxWidgets Lizenz basiert auf der LGPL mit der Ausnahme, dass damit erstellte Applikationen in binärer Form unter den Bedingungen des Benutzers veröffentlicht werden dürfen. Also ist es natürlich auch möglich, kommerzielle Applikationen mit Hilfe von wxWidgets zu erstellen, und unter [6] ist eine Liste von Firmen einsehbar, die wxWidgets bereits erfolgreich in ihren Produkten einsetzen.

Zusammenfassung

Vorteile:

- Einfach zu erlernen
- Ähnlichkeit zu MFC
- Umfangreiche Funktionalität
- weite Verbreitung
- guter Support via Newsgroups und mailing list, oft direkt von den Autoren
- wird auch für kommerzielle Applikationen eingesetzt
- Unterstützung populärer Compiler und Entwicklungsumgebungen
- komplett im Quellcode unter einer LGPL-basierten Lizenz verfügbar
- existiert bereits seit zwölf Jahren
- keine sichtbaren Unterschiede zu nativen Applikationen
- kann auch eigene Widgets verwenden statt den nativen
- Unicode- und Internationalisierungs - Unterstützung

Nachteile:

- Implementierungsunterschiede in der wxWidgets-API auf den einzelnen Plattformen führen zu teilweise unterschiedlichem Verhalten
- Probleme beim Vermischen von wxWidgets und MFC-Code
- manche plattformspezifischen Funktionen werden nicht unterstützt

Tier	Beschreibung
Client	Webbrowser in dem die Programmoberfläche zur Interaktion dargestellt wird
Webserver	Kommuniziert mit dem Applikationsserver und erstellt dynamische Webseiten
Applikationsserver	Hier laufen die Applikation die die Geschäftslogik implementieren
Datenserver	Datenbank mit persistenten Daten zur Verwendung durch den Applikationsserver

Tabelle 5: Architektur des CSPmed Systems mit Webservices

Verwendbarkeit für den CSPmedclient

Das wxWidgets Toolkit ist sowohl funktionsmäßig als auch hinsichtlich der Lizenz unter der es veröffentlicht ist, sehr gut für den Einsatz bei der Entwicklung eines plattformunabhängigen kommerziellen CSPmedclients geeignet. Für die Version des CSPmedclients unter GNU/Linux kommt wxGTK zum Einsatz, da GTK+ auf den meisten Systemen vorhanden sowie sehr ausgereift (siehe 5.2.4) ist. Die Version die unter Microsoft Windows lauffähig ist, wird mit Hilfe von wxMSW erstellt. Wie bereits erwähnt greift diese API auf die native Win32 API zur Darstellung der grafischen Benutzeroberfläche zurück und erzeugt somit Applikationen, die rein optisch nicht von nativen Applikationen unterschieden werden können. Natürlich ist der Quellcode, der auf Funktionen von wxMSW beziehungsweise wxGTK zugreift, der gleiche, das bedeutet, ein- und derselbe Quellcode kompiliert und läuft sowohl unter Microsoft Windows als auch unter GNU/Linux. Es sollte auch ohne großen Aufwand möglich sein, den CSPmedclient auf anderen von wxWidgets unterstützten Plattformen, wie zum Beispiel MacOS X, einzusetzen.

5.2.8 Implementierung als Webapplikation

Dieses Konzept zur Implementierung unterscheidet sich grundsätzlich von den anderen oben vorgestellten Möglichkeiten, da in diesem Fall eine Applikation die auf einem zentralen Server ausgeführt wird die Benutzerschnittstelle erstellt und dies nicht wie in den anderen Fällen durch eine auf dem Clientrechner laufende Applikation erfolgt. Jeder Benutzer, der das CSPmed System verwenden möchte verbindet sich mit einem Webbrowser zum Server, der die von diesem empfangenen Daten aufbereitet und die Benutzerschnittstelle darstellt. Das komplette CSPmed Systems würde dann eine 4-tier Architektur aufweisen, welche in Tabelle 5 dargestellt ist. Zu implementieren wäre demnach eine am Webserver laufende Applikation, die Aufrufe des Applikationsservers entgegennimmt und aufgrund dieser dynamische Webseiten erzeugt. Als Client würde, wie bereits erwähnt, nahezu jeder beliebige Webbrowser ausreichen, die benötigte Rechenleistung würde sich also vom Client zum (Web-) Server verschieben.

Besonders im Bereich der Erzeugung von dynamischen Webinhalten existieren viele verschiedene Standards und Produkte sowohl freier als auch kommerzieller Natur. Die Entscheidung für oder gegen ein bestimmtes Produkt wird unter Anderem von folgenden Kriterien beeinflusst:

- Erwartetes Datenaufkommen
- Skalierbarkeit hinsichtlich der Benutzerzahl
- Verfügbarkeit für verschiedenste Plattformen sowohl client- als auch serverseitig
- Lizenz
- Dokumentation und Verbreitung, d.h. installierte Basis

Die client- und serverseitig notwendigen Technologien für die Realiserung einer derartigen Applikation sollen in der vorliegenden Arbeit nicht im Detail erläutert werden, es treffen unter anderem die bereits in 5.1 dargestellten prinzipiellen Unterscheidungsmerkmale zu. Denkbar wäre etwa die Verwendung von *PHP* in Kombination mit *JavaScript* oder eine Realisierung als Java *Servlet*.

Zusammenfassung

Vorteile

- Schlanke Clients mit geringer Rechenleistung einsetzbar
- Aufbereitung der Oberfläche für verschiedene Endgeräte vom PC bis zum PDA oder Mobiltelefon denkbar
- Verwendung offener Standards, CSP-eigene RPC Implementierung nicht mehr notwendig

Nachteile

- Bestehender Code so gut wie nicht verwendbar, komplette Neuimplementierung notwendig

Verwendbarkeit für den CSPmed Client

Die Realisierung der CSPmed Applikation unter Verwendung dynamisch erzeuger Webseiten zur Darstellung der Benutzeroberfläche wäre zweifellos eine sehr elegante und zeitgemäße Möglichkeit den proprietären CSPmedclient abzulösen. Vor Allem eine Aufbereitung für kompakte Geräte, also PDAs oder Mobiltelefone würde einen gewaltigen Zusatznutzen bringen, da sich der Arzt zum Beispiel auf Hausbesuch online mit dem CSPmed Server seiner Ordination verbinden und

seine gewohnte Applikation verwenden könnte. Die zu übertragene Datenmenge wäre aufgrund des verwendeten (X)HTML Formats außerdem sehr gering.
Entscheidet man sich für diese Lösung, kann man bestehenden Code des CSPmedclients nicht weiterverwenden, da sich sowohl die verwendete Programmiersprache von der bisherigen Implementierung unterscheidet als auch das gesamte Interaktionskonzept. Viele der oben genannten Vorteile kann man auch erhalten wenn man die in 10.3.2 vorgeschlagene Strategie verfolgt und den bisherigen CSPmedclient auf dem Server ausführt und die Benutzerschnittstelle auf dem lokalen Rechner ausgibt. Es ist damit zu rechnen, dass die Umwandlung der Maskendateien in ein ansprechendes Weblayout sehr viel Entwicklungszeit beanspruchen wird, vor allem wenn man daran denkt, dass zum Beispiel auch modale Fenster angezeigt werden müssen.
Prinzipiell ist eine Implementierung wie oben beschrieben möglich, doch in Anbetracht des hohen Implementierungsaufwandes und der Tatsache dass weitaus einfachere Lösungen ähnliche Vorteile bieten können wird diese Implementierungsvariante verworfen.

6 Konkrete Implementierung

Bei der Reimplementierung einer bestehenden Softwareapplikation ist es unerlässlich, zuerst eine möglichst umfangreiche Analyse des Systems durchzuführen. Dies bedeutet vor allem, dass man den konkreten Funktionsumfang der Applikation in Erfahrung bringt und wie dieser implementiert ist. Oft werden Funktionen verwendet, die in dem Toolkit, das für die portierte Applikation verwendet werden soll, nicht existieren beziehungsweise anders implementiert sind. Ein Beispiel hierfür wäre zum Beispiel die Art, ob und wie verschiedene Klassen oder Threads zur Laufzeit Information untereinander austauschen. Dies kann je nach Toolkit unterschiedlich gelöst sein oder es existiert zum Beispiel überhaupt keine Unterstützung für Threads. In letzterem Fall muss man entscheiden, ob man ein anderes Toolkit wählt, die entsprechenden fehlenden Funktionen selbst implementiert oder die Funktionalität anders oder gar nicht portiert.
Weiters ist es wichtig zu wissen, von welchen Techniken die Applikation Gebrauch macht und ob diese Techniken auch für alle Zielplattformen zur Verfügung stehen. Im Falle des CSPmed Clients wird eine eigene RPC-Bibliothek verwendet, die sowohl gegen den CSPmed Client als auch gegen die Programme, die am Server laufen, gelinkt werden muss. Zur Zeit kompiliert und funktioniert diese Bibliothek unter SCO Unix, Next, Linux und Windows.

6.1 Herangehensweisen

Da die Klassen des wxWidgets Toolkits sehr ähnlich zu den Klassen der MFC sind, wäre es denkbar, die MFC-Aufrufe durch die entsprechenden Aufrufe der wxWidgets Bibliothek zu ersetzen. Dies funktioniert zwar für grundlegende Funktionaliät

wie zum Beispiel Zeichenketten oder manche GUI-Elemente sehr gut, doch man stößt schon bei etwas komplexeren Aufgaben auf deutliche Unterschiede zwischen den beiden Bibliotheken. Vor allem die Klassen für das Fenstermanagement oder diejenigen zur Erstellung und Verwaltung von dynamischen Listen weisen in der wxWidgets Bibliothek einen etwas geringeren Funktionsumfang auf und sind auch teilweise anders zu handhaben (etwa in Bezug auf das Hinzufügen von Elementen an einer festen Stelle). Wählt man also den Ansatz der direkten Konvertierung von bestehendem Code und stößt man auf Differenzen zwischen MFC und wxWidgets, ist es meist der Fall, dass nicht nur in der gerade bearbeitete Klasse umfangreiche Änderungen vorgenommen werden müssen, sondern auch in all jenen Klassen, die mit dieser in Verbindung stehen und ihre Methoden aufrufen. Man sieht sich also schnell mit einer Lawine an Änderungen konfrontiert, die man alle durchführen muss, um die Applikation zu kompilieren, da paralleles Verwenden der MFC und wxWidgets Bibliotheken nicht unproblematisch ist. Dadurch wird auch das Testen sehr erschwert, da die Auswirkung von Änderungen oft nicht auf den ersten Blick ersichtlich ist. Weiters braucht man bei diesem Ansatz genaues Wissen darüber, wie eine Applikation arbeitet und wie die zu konvertierenden Codeteile arbeiten, was man zu Beginn der Konvertierungsarbeit oft noch nicht besitzt. Auf der anderen Seite ist es aber auch unbedingt notwendig zu wissen, welche Klassen der wxWidgets Bibliothek sich wie verhalten und ob sie dazu geeignet sind, die MFC-Klassen in der gewünschten Weise zu ersetzen. Auch dies ist wohl nur dann vorhanden, wenn man bereits einige Erfahrung im Umgang mit der wxWidgets Bibliothek sammeln konnte.
Aufgrund der obigen Nachteile ist auch bei einfachen Applikationen meiner Meinung nach dringend davon abzuraten, bei geringer Erfahrung mit den beiden Toolkits „einfach" MFC-Aufrufe durch ihre wxWidgets Äquivalente oder das, was man dafür hält, zu ersetzen. Es ist auf jeden Fall lohnenswerter, die Struktur und Funktionsweise der Applikation genau zu untersuchen und eine reine wxWidgets Neuimplementierung zu beginnen, die man jeweils nach Hinzufügen von neuer Funktionalität testen kann.
Dies war dann auch der Ansatz, den ich bei der Portierung der Software verfolgt habe. Da einige Bibliotheken gegen den CSPmed Client gelinkt sind, wurde es notwendig, auch diese Bibliotheken dahingehend zu untersuchen, welche Funktionalität sie zur Verfügung stellen und ob Anpassung an die neu verwendete Klassenbibliothek beziehungsweise an die neuen unterstützten Plattformen notwendig sind.
Der erste Schritt bei der Erstellung einer plattformunabhängigen Implementierung war, die notwendigen, in 6.2 beschriebenen Entwicklungswerkzeuge und Bibliotheken einzurichten. Dies ist eine triviale Aufgabe da die Standardentwicklungstools in jeder modernen Linux-Distribution enthalten sind. Bei der wxWidgets Bibliothek hat man die Wahl zwischen zwei Varianten: zum Einen existiert eine als *stable* gekennzeichnete Version 2.4.2 und eine *development* Version 2.5.5 (Stand 09.04.2005). Wie der Name schon vermuten lässt, ist die stable Version getestet und größtenteils fehlerfrei, hat jedoch zum Teil weniger Funktionalität als die de-

velopment Version. Da keine detaillierte Auflistung der in der development Version enthaltenen zusätzlichen Funktionalität beziehungsweise ausgebesserten Fehler gegenüber der stable Version zu existieren scheint, fiel die Entscheidung zugunsten der Verwendung der stable Version.
Nun wurde die eigentliche Applikation von Grund auf ausgehend von dem vorhandenen Microsoft Windows Code unter SuSE Linux neu implementiert. Als Entwicklungsplattform wurde Linux gewählt, da dieses System meiner Meinung nach für das Entwickeln am besten geeignet ist, da viele freie Tools zur Verfügung stehen. Würde man die Applikation unter Windows zu reimplementieren beginnen, müsste man sich die freien Tools zuerst für dieses Betriebssystem besorgen um sicherzugehen, dass das System dann auch unter Linux kompiliert. Es existiert zwar eine gut dokumentierte Unterstützung und Einbindung von wxWidgets in das Microsoft Visual Studio, doch die vorhandenen Unterschiede zwischen Windows- und Unixprogrammierung, auf die später noch eingegangen werden wird, lassen eine Implementierung von Windows-Code unter Linux als die größere Schwierigkeit erscheinen. Das fertige System soll ohnehin auf beiden Plattformen aufgrund einer einheitlichen Codebasis laufen, sodass einem die Portierung, also idealerweise die Rekompilierung des Codes auf dem jeweiligen anderen System, nicht erspart bleibt.
Wie schon öfters erwähnt ist das Ziel der Reimplementierung ein Code, der durch bloßes Rekompilieren den CSPmed Client auf unterschiedlichen Plattformen erzeugt. Durch die Verwendung der wxWidgets Bibliothek ist dieses Ziel für die grafischen Elemente der Benutzeroberfläche verhältnismäßig einfach zu erreichen, indem man versucht, die MFC-Aufrufe durch wxWidgets Aufrufe zu erstzen. Man kann dabei allerdings nicht blindlings vorgehen, sondern muss mehrere unterschiedliche Fälle beachten, die im Folgenden beschrieben werden sollen:

Austauschen der Klassennamen: Grundsätzlich ist es oft der Fall, dass ähnlich benannte MFC- und wxWidgets-Klassen großteils dieselbe Funktionalität zur Verfügung stellen. Zum Beispiel ist sowohl in beiden Toolkits eine Klasse zur Behandlung von Zeichenketten enthalten, die in der MFC-Bibliothek *CString*, in wxWidgets *wxString* heißt. Die Klassennamen beginnen in der MFC-Bibliothek grundsätzlich mit C, in der wxWidgets-Bibliothek mit wx. Methoden und Konstruktoren sind bei den Stringklassen großteils identisch, sodass man in diesem Fall einfach nur den Klassennamen im MFC-Code zu ändern braucht um das plattformunabhängige Äquivalent zu verwenden.

Ähnliche Klassen, unterschiedliche Konstruktoren: Obwohl oft in beiden Toolkits bis auf das Präfix gleichbenannte Klassen existieren, kommt es doch oft vor, daß diese in Details unterschiedlich sind, was dem Entwickler auf den ersten Blick allerdings verborgen bleibt. Am Beispiel der *wxRect* Klasse ist es der Konstruktor, der unterschiedlich zur MFC-Klasse *CRect* definiert ist. Statt ein Rechteck wie in den MFC mittels der Koordinaten links/oben/rechts/unten zu definieren, muss

man dies in der wxRect Klasse mittels der Koordinaten links/oben/breite/höhe erledigen. Obwohl dies ein zugegebenermaßen triviales Beispiel darstellt, ist es doch notwendig in den meisten Fällen die Dokumentation der einzelnen Klassen durchzulesen, zumal ein Fehler im oben beschriebenen Fall zu keiner Fehlermeldung, weder zur Kompilier- noch zur Laufzeit führt und so ein ungewünschtes Programmverhalten auftreten kann.

Unterschiedliche Funktionalität: Da die wxWidgets Klassenbibliothek nicht als eins-zu-eins Kopie der MFC entworfen wurde und dies laut seinen Entwicklern auch keinesfalls sein möchte, existieren in der MFC-Klassenbibliothek natürlich zahlreiche Klassen, für die es keine direkte Entsprechung in der wxWidgets Bibliothek gibt, und man daher versuchen muss, das gewünschte Programmverhalten unter Microsoft Windows mit Hilfe der wxWidgets-Aufrufe nachzuahmen.
Dies ist zum Beispiel der Fall, wenn Events behandelt werden müssen. Falls der Benutzer zum Beispiel in ein Texteingabefeld etwas eingibt, erhält dieses Feld vom Toolkit zum Beispiel die Events, dass der Fokus neu gesetzt werden, das Eingabefeld neu gezeichnet werden muss und so weiter. Diese Events sind jedoch bei den diversen Toolkits unterschiedlich (obwohl in Auftreten und Funktion oft identisch) und werden auch unterschiedlich implementiert. Setzt man etwa im MFC Toolkit den Wert eines Eingabefeldes innerhalb des Codes, wird, im Gegensatz dazu wenn der Benutzer „manuell" etwas in das Eingabefeld einträgt, kein Updateevent an das Feld gesendet. Das wxWidgets Toolkit sendet jedoch den Event in beiden Fällen, sodass sich die Applikation anders verhält wenn man das Eventhandling direkt übernimmt. Diese Art von Unterschieden ist großteils dafür verantwortlich, dass eine als trivial angenommene Portierungsaufgabe doch mehr Zeit als ursprünglich dafür veranschlagt in Anspruch nimmt, da man in diesem Fall sowohl das MFC als auch das wxWidgets Toolkit genau kennen muss und auch das zu portierende Programm umfangreich analysiert werden muss, um die alte Funktionalität mit dem neuen Toolkit möglichst sauber nachzuprogrammieren.
Ein anderes Beispiel wäre etwa die Erzeugung und der Zugriff auf dynamische Listen. Sowohl das MFC- als auch das wxWidgets Toolkit bringen Klassen mit, die die Verwaltung mehrerer zusammengehöriger Elemente vom gleichen Typ übernehmen. Diese Objekte werden aber auf unterschiedliche Weise erzeugt und auch der Zugriff darauf gestaltet sich unterschiedlich. Zum Beispiel existiert im MFC Toolkit eine Methode, die ein Objekt an einer beliebigen angegebenen Stelle der Liste einfügt und, falls der Index des neuen Elementes größer als die Anzahl der bereits in der Liste vorhandenen Elemente ist, die Liste entsprechend vergrößert. Eine solche Methode bietet das wxWidgets Toolkit nicht an, sodass man diese Funktionalität mit den angebotenen Methoden selbst nachprogrammieren muss. Ähnlich verhält es sich übrigens mit Threads, deren Erzeugung wiederum in beiden Toolkits unterschiedlich gelöst ist.

Plattformspezifische Besonderheiten: Es existieren allerdings im Code des CSPmed Clients auch von der Benutzeroberfläche unabhängige Codeteile, die nur unter Microsoft Windows ausgeführt werden können und die in der Form nicht durch wxWidget-Aufrufe ersetzt werden können. Diese Codeteile sind zum Beispiel das Erzeugen von asynchronen Sockets zur Verwendung mit der libcsprpc Bibliothek. Leider weicht Microsofts Socket-Implementierung in einigen Punkten von der standard Berkeley Socket-Implementierung ab, sodass es notwendig ist, den Code zur Erzeugung und zum Datenverkehr über die Sockets, für alle unterstützten Plattformen separat zu implementieren. Am Einfachsten und Übersichtlichsten erfolgt dies durch Präprozessoranweisungen im Code, da man mit deren Hilfe zur Kompilationszeit angeben kann, für welche Plattform man die ausführbare Datei erzeugen möchte und nur die für diese Plattform benötigten Codeteile auch übersetzt werden.
Beispielsweise müssen zur Kommunikation mit der Applikation, die am Server läuft, non-blocking Sockets erstellt werden. Kurz zum Hintergrund: Beim Lesen von einem blocking Socket wird die lesende Methode solange blockiert, bis Daten vorhanden sind. Dies ist bei einem non-blocking Socket nicht der Fall, hier wird ein Fehlercode von der lesenden Methode zurückgeliefert und das Programm weiter ausgeführt. Das Erstellen dieser Sockets geschieht unter Microsoft Windows mit Hilfe der Winsock API und der Anweisung

Algorithm 8 Erstellen eines non-blocking Sockets unter Windows

```
SOCKET hSock;
unsigned long ulFlag=1;
...
ioctlsocket(hSock,FIONBIO,&ulFlag);
```

Im Gegensatz dazu der entsprechende Code für die Verwendung unter GNU/Linux:

Algorithm 9 Erstellen eines non-blocking Sockets unter GNU/Linux

```
int hSock;
...
fcntl(hSock, F_SETFL, O_NONBLOCK);
```

6.2 Entwicklungswerkzeuge

Bei der Erstellung der Portierung des CSPmedclients wurde ausschließlich Open Source Software verwendet. Der Quellcode wurde mittels des C++ Compilers der *GCC* (*Gnu Compiler Collection*) übersetzt und unter Verwendung von *DDD* (*Data Display Debugger*), der wiederum auf *GDB* (*The GNU Project Debugger*) beruht, debugged. Implementiert wurde auf einem Rechner mit SuSE Linux unter *KDE* mit dem Editor *Kate* (*KDE Advanced Text Editor*). Der Login Dialog des CSPmedclients wurde unter Verwendung von *wxGlade* erzeugt, einem Tool mit dem man Benutzerschnittstellen auf grafischem Weg per „point and click” erstellen

kann. Zu guter Letzt wurde auch die vorliegende Arbeit mit dem freien Satzsystem LaTeXgesetzt.

6.3 Struktur und Konzepte

Wie bei allen Applikationen, die mit Hilfe des wxWidgets Toolkits erstellt wurden, beginnt die Ausführung des Programms in der `OnInit()` Methode der von `wxApp` abgeleiteten Klasse, also in unserem Fall `UIMain::OnInit()`. Diese Methode erzeugt das Hauptfenster (Mainframe) und ruft `UIManage::EstablishConnection()` auf. Letztere Methode initialisiert einige globale Variablen der libcsprpc Bibliothek (Modul CSPm_001.c). Weiters wird `UIManage::Connect()` aufgerufen und hier erfolgt auch der erste Netzwerkzugriff mit dem Verbindungsaufbau zum Server. Microsoft Windows- und Unix (GNU/Linux) Socketcode sind mittels `#ifdef` Präprozessoranweisungen getrennt. Die Anmeldung am Server erfolgt über ein eigenes Protokoll (IP-Adresse, Benutzername und Passwort werden vom Client geschickt), danach wird der nonblocking Mode des Clientsockets aktiviert und diverse Initialisierungsroutinen der libcsprpc Bibliothek (welche bereits in 3.1.3beschrieben wurde) ausgeführt. Danach wird die Methode `StartSelectThread()` aufgerufen, die einen non-blocking Socket erzeugt, von dem synchronisert gelesen werden kann. Dies erfolgt mittels eines Threads und einer globalen Datenstruktur (`hP007->iMessageResult` beziehungsweise `hP007->pstSelect`), die von `SelectThread::Entry()` geschrieben und von `CSP007_GetEvent()` in der Datei `CSPm_007wx.c` gelesen wird. Dies wird durch eine Semaphore abgesichert, es ist also ständig sichergestellt, dass wenn `CSP007_GetEvent()` aufgerufen wird, die aktuellen Daten des Sockets in der Struktur `hP007` vorzufinden sind.
Nach erfolgreichem `UIManage::Connect()` wird `UIManage::ConnectAndWait()` aufgerufen. Diese Methode ruft in einer Schleife `CSP005_Receive()` auf, eine Methode der libcsprpc die veranlasst, dass bei einem Aufruf der Methode `wwmsrpc_call_1()` im WMS-Toolkit die gleichnamige Methode des CSPmedclients per RPC aufgerufen wird. Die Schleife in der Methode `UIManage::ConnectAndWait()` ist auch verantworlich dafür, dass der CSPmedclient erst beendet wird, wenn `CSP005_Receive()` einen entsprechenden Rückgabewert liefert, was zum Beispiel der Fall ist wenn der Benutzer auf den Schließen-Button des Hauptfensters klickt. Zur näheren Erläuterung erfolgt im folgenden Abschnitt ein kurzer Einschub zu den grundlegenden Mechanismen der Benutzerinteraktion mit den verwendeten Toolkits, also MFC und wxWidgets.

6.3.1 Eventgesteuerte Interaktion vs. Callbackfunktionen

Die meisten Toolkits zur Programmierung von grafischen Benutzeroberflächen wie zum Beispiel die Microsoft Foundation Classes (MFC) oder eben auch wxWidgets arbeiten nach einem eventbasierten System der Benutzerinteraktion. Die Aus-

führung einer Applikation erfolgt in einer vom Programmierer zu überschreibenden Methode der Applikationsbasisklasse des entsprechenden Toolkits. Dies wäre zum Beispiel die `wxApp::OnInit()` Methode des wxWidgets Toolkits oder die `CWinApp::InitInstance()` Methode der MFC. Implementiert man eine grafische Oberfläche, erzeugt man das Hauptfenster und die Objekte, die man für die Applikation benötigt in dieser Initialisierungsmethode beziehungsweise in den von ihr aufgerufenen Methoden und Konstruktoren. Nach dem Erstellen und Anzeigen wird die Initialisierungsmethode im Allgemeinen wieder beendet.
Das eigentliche Warten auf Benutzereingaben wird durch das Toolkit in den entsprechenden anderen Methoden der Applikationsbasisklasse durchgeführt. Möchte man nun Benutzereingaben, wie zum Beispiel einen Mausklick oder einen Tastendruck in einem Textfeld auswerten, steht dazu ein Eventsystem zur Verfügung. Falls zum Beispiel ein Button angeklickt wird, kann man in der entsprechenden Klasse, deren Mitglied er ist, festlegen, welche Methode bei diesem Klickevent aufgerufen werden soll. In dieser, nun bei jedem Klick auf den Button aufgerufenen Methode, implementiert man nun die Funktionalität, die stattfinden soll, also man erzeugt beispielsweise ein neues Fenster mit einem Hinweis oder schreibt einen Text in bereits angezeigte Formularelemente. Nach diesen Aktionen geht die Programmkontrolle wieder an das Toolkit beziehungsweise das darunterliegende Betriebssystem zurück.
Das WMS-Toolkit, das von den Programmen, die am Server ausgeführt werden, verwendet wird, arbeitet nach einem anderen Prinzip. Soll der Benutzer Eingaben in ein Formular vornehmen können, wird am Serverprogramm die Methode `Rdform()` mit dem auszufüllenden Formular als Parameter aufgerufen. Die `Rdform()` Methode liefert nun erst dann einen Rückgabewert wenn der Benutzer die Abschlusstaste drückt oder abbricht. Somit wird das Serverprogramm auch erst nach vollzogener Benutzereingabe weiter ausgeführt. Möchte man innerhalb dieser Methode auf eine Benutzereingabe reagieren, muss man dies mit Hilfe von Callbackfunktionen machen. Die Zuordnung einer Callbackfunktion zu einer konkreten Benutzereingabe muss dabei vor der Ausführung der Methode `Rdform()` erfolgen, der darin enthaltene Eventloop der auf Benutzereingaben „pollt", wird kurzzeitig unterbrochen und in die Callbackfunktion „zurückgesprungen". Ist der Code in der Callbackfunktion abgearbeitet, wird der Eventloop weiter fortgesetzt. Dieses Verhalten muss nun auch auf den CSPmedclient abgebildet werden. Falls das Serverprogramm die `Rdform()` Methode des WMS aufruft, schickt dieses das Kommando `Rdbuff` per RPC-Aufruf von `wwmsrpc_call_1()` an den CSPmedclient und nimmt als Rückgabewert der `Rdform()` Methode den Rückgabewert der `wwmsrpc_call_1()` Methode. In anderen Worten bedeutet das, dass `wwmsrpc_call_1()` erst einen Wert zurückliefern darf, wenn der Benutzer die Eingabe in das Formular abgeschlossen hat.

6.3.2 Auswirkungen auf den CSPmedclient

Das Konzept der Callbackfunktionen im WMS ist allerdings unvereinbar mit dem Eventsystem des wxWidgets Toolkits, bei dem ja prinzipiell jederzeit eine Benutzereingabe erfolgen kann und nicht nur wenn man dediziert eine Methode aufruft. Ein Lösungsversuch war, die Aufrufe von `CSP005_Receive()` in der Schleife der Methode `UIManage::Connect()` in einen eigenen Thread zu verpacken, der parallel zum User Interface Thread läuft. Allerdings würden dann die Aufrufe von `wwmsrpc_call_1()` sowie die weitere Behandlung der Calls und die Darstellung des UI ebenfalls in diesem Thread erfolgen. Das wxWidgets Toolkit ist allerdings, wie viele andere Toolkits auch, nicht thread-safe ausgelegt was bedeutet, dass nur genau ein Thread UI-Operationen durchführen darf. Also muss der Thread, der die `wwmsrpc_call_1()` Methode aufruft, irgendwann zur Bearbeitung eine Message mit dem zu bearbeitenden Call und dessen Parameter an den UI-Thread schicken, der dann die weitere Bearbeitung übernimmt. Dabei muss man bedenken, dass der Thread, der die `CSP005_Receive()` Aufrufe absetzt, solange warten muss, bis der Benutzer den Einlesevorgang der Maske abgeschlossen hat. Es ist also eine Synchronisierung, zum Beispiel mittels Semaphoren, notwendig.
All dies wäre noch mit vertretbarem Aufwand implementierbar, jedoch gibt es eine weitere Schwierigkeit: Während der Benutzer zum Beispiel Felder des Formulars ausfüllt, muss oft ein Feedback vom CSPmedclient zum Server erfolgen, etwa dann wenn eine Funktionstaste gedrückt wurde und ein Callback ausgelöst werden soll. Dann führt der CSPmedclient nämlich einen sogenannten reverse-Callback durch, der Methoden der libcsprpc Bibliothek aufruft. Diese wartet jedoch jetzt schon auf das Ende der `Rdform()` Methode, ist also durch die Semaphore blockiert und kann den Callback nicht ausführen. Das heißt, man muss dieses Problem anders lösen. Dadurch, dass der CSPmedclient, die libcsprpc Bibliothek und das WMS so eng verzahnt sind, ist es ohne Eingriffe in alle drei Komponenten so gut wie nicht möglich, das WMS auf das eventbasierte Benutzerinteraktionskonzept der wxWidgets anzupassen.
Um die Problematik und deren Lösung noch einmal klar herauszustreichen, sei angenommen, der CSPmedclient und das WMS würden auf dem gleichen Rechner laufen, ohne zwischenliegende libcsprpc Bibliothek. Dabei sei weiters folgendes einfache Szenario angenommen: Auf dem Bildschirm wird ein Formular angezeigt, auf dem sich ein Button befindet. Der Benutzer klickt auf diesen Button und ein neues, anderes Formular erscheint am Bildschirm. Der Programmablauf mit Hilfe des oben angesprochenen Lösungsversuches mittels Threads würde folgendermaßen aussehen:
Das WMS schickt ein `Rdform()` Kommando an den CSPmedclient und erwartet einen Rückgabewert für dieses Kommando. Um diesen zu erhalten muss der CSPmedclient eine Benutzerinteraktion durchführen lassen, was aufgrund des Aufbaus des wxWidgets Toolkits (und in weiterer Folge aufgrund des benutzen Eventsystems) nur dann passieren kann, wenn der Programmfluss die `wxApp::OnInit()`

Methode verlässt. Um dieses Verlassen also bewerkstelligen zu können, muss man einen Thread abspalten um das WMS warten zu lassen und den CSPmedclient davon unabhängig weiter ausführen zu können. Für einfache use-cases würde dies auch funktionieren, nicht jedoch im Falle des in unserem Szenario vorhandenen Buttonklicks **während** der Benutzerinteraktion. Dann nämlich würde ein Methodenaufruf auf das WMS notwendig sein, dessen Thread in dem es läuft wartet jedoch zu diesem Zeitpunkt noch auf das Ergebnis des `Rdform()` Kommandos. Das erwartete zweite `Rdform()` Kommando zum Einlesen des aufgrund des Buttondrucks angezeigten Formulars würde also nie zum CSPmedclient gesendet werden, da schon der Reversecallback aufgrund des Blockierens des WMS-Threads nie ausgewertet wird!
Die Lösung die zufriedenstellen für alle Fälle funktioniert ist Folgende: Zu Beginn des Szenarios schickt das WMS das `Rdform()` Kommando an den CSPmedclient. Letzter startet jetzt jedoch einen eigenen Messageloop, in dem das angezeigte Formular behandelt wird und eine Benutzereingabe möglich ist. Somit steht erst nach dem Ende dieses Messageloops der Rückgabewert des `Rdform()` Kommandos zur Verfügung, man erspart sich, einen eigenen Thread abzuspalten und dessen Synchronisierung. Der Klick auf den Button im aktuell einzulesenden Formular ist auch kein Problem, da nun einfach der Reversecallback und die entsprechende Behandlungsroutine im laufenden Serverprogramm ausgeführt werden kann, die im oben genannten Szenario wieder ein `Rdform()` Kommando an den CSPmedclient absetzt. Die `wwmsrpc_call_1()` Methode des CSPmedclients wird also in diesem Fall rekursiv aufgerufen und die entsprechenden Rückgabewerte immer richtig abgeliefert. Mit Hilfe des eigenen Messageloops nach Erhalt des `Rdform` Kommandos kann man also das Verhalten des CSPmedclients an jenes des WMS angleichen.
Wird nun ein `Rdform` Kommando an den CSPmedclient gesendet, startet dieser in der Methode `UIForm::Run()` einen eigenen Eventloop für das aktuell angezeigte Formular. Innerhalb dieses Loops ist es möglich Eingaben vorzunehmen und auch Events zu verarbeiten. Erst wenn das `UIForm` Objekt den Event `UM_FORM_END` oder `UM_FORM_QUIT` erhält, wird der Messageloop beendet und der Returnwert der `Rdform()` Methode an das WMS per libcsprpc weitergegeben. Die Events zum Beenden des Messageloops werden zum Beispiel dann gesendet, wenn der Benutzer die Abbruchtaste während der Formulareingabe drückt oder das entsprechende Fenster schließt.
Die oben beschriebenen (zum Teil vergeblichen) Versuche zur Abstimmung der verschiedenen Benutzereingabemethoden im CSPmedclient (wxWidgets) und im WMS beanspruchten die meiste Zeit bei der Portierung der Applikation. Da die `wxEventLoop` Klasse der wxWidgets nicht dokumentiert ist, musste man sich die Idee zur Verwendung dieser Klasse aus den modalen Dialogen des wxWidgets Quellcodes holen. Auf jeden Fall scheint die gefundene Lösung sowohl in allen use-cases zu funktionieren, als auch plattformunabhängig zu sein, da sie unter Microsoft Windows und GNU/Linux einwandfrei läuft.

6.3.3 Erzeugung und Anzeige von GUI-Elementen im CSPmedclient

Einer der ersten WMS-Calls den eine Applikation ausführt ist `WMSC030_Init`. Dadurch wird die Methode `UIManage::Init()` aufgerufen, die das „nullte" `UIWin` Objekt anlegt und in das Fensterarray von `UIManage` einträgt. `UIManage` verwaltet in einer dynamischen Liste alle Fenster. Nach dem Eintragen wird die erste Form erzeugt und in das Formulararray von `UIManage` eingetragen (ab dem Index WMS030_HANDLE_INTERNAL_FORMS). Sowohl das `UIWin` Objekt als auch das `UIForm` Objekt bekommen als ID den Index, an dem sie im entsprechenden Array eingetragen sind. Das erste Formular ist die „Controlform", die auf dem 0-ten Fenster sitzt. Danach werden die Buttons sowie das Messagecontrol und das Responsecontrol erzeugt. Das eben erzeugte 0-te Fenster wird dann gleich mittels `UIWin::Create()` an der Standardposition angezeigt. Das Anzeigen und Setzen der Größe erfolgt beim Standardfenster mit fixen Werten, deshalb wird nicht explizit ein `WMSC030_Openwin` und `WMSC030_Ldwin` benötigt. Für alle anderen Fenster mit höherem Index, die in der Applikation benötigt werden, muss man diese beiden Calls explizit aufrufen.
`WMSC030_Openwin` führt die Methode `UIWin::Create()` aus, die wiederum das Fenster erzeugt, falls dies noch nicht geschehen ist, oder es einfach nur mehr anzeigt, falls es bereits erzeugt wurde. Das Anzeigen des Hauptfensters erfolgt übrigens auch erst hier, direkt nachdem das erste benutzerdefinierte Fenster erzeugt wurde.
`WMSC030_Ldwin` setzt die Fensterposition aufgrund des Beschreibungsstrings sowohl in Zeilen- und Spalten- als auch in Pixelkoordinaten. Zum Umrechnen der Koordinaten von Zeilen- und Spalten- in Pixelkoordinaten wird die Methode `UIManage::RecalcRect()` verwendet. Die eigentliche Darstellung eines Formulars und die Erzeugung seiner Controls erfolgt nach dem Call `WMSC030_Ptform`. Dabei werden die `::Create()` Methoden der jeweiligen UI-Controls aufgerufen. Die UI-Controls sind Objekte, die bei jedem Erzeugen eines `UICtl` Objektes angelegt wurden (z.B. `UIButton` oder `UIEdit`) und die jeweilige wx-Widgets Klasse (z.B. `wxButton` beziehungsweise `wxTextCtrl`) und `UICtrl` als Superklasse haben.
Eine vollständige Beschreibung aller Vorgänge und Calls erscheint hier wenig sinnvoll denn von der grundlegenden Vorgangsweise her funktioniert alles ziemlich einheitlich: Die Fenster und Masken werden jeweils mit einer ID vom WMS angesprochen und auf diese kann man die entsprechenden Methoden zur Bearbeitung der Calls (Öffnen, Controls setzen usw.) anwenden. Einzig die Erzeugung und Verwaltung der Controls ist etwas gewöhnunsbedürftig hinsichtlich der Klassenhierarchie gelöst, hier könnte man vielleicht Verbesserungen einbringen und auch einige Membervariablen eliminieren um mehr Klarheit zu schaffen (siehe dazu auch Abschnitt 8 und 10).

6.4 Details zur Kompilierung

Der CSPmedclient Quellcode ist darauf ausgelegt, auf verschiedenen Plattformen kompiliert zu werden. Dies erfolgt unter Verwendung der wxWidgets Bibliothek (siehe 5.2.7), die es gestattet, ein und denselben Quellcode für mehrere Plattformen zu kompilieren. Der aktuell vorliegende Quellcode ist unter Microsoft Windows und GNU/Linux getestet und lauffähig, eventuell auch auf anderen UNIX-ähnlichen Betriebssystemen die nicht getestet wurden. Bei Änderungen am Quellcode ist immer darauf zu achten, die plattformunabhängigen Methoden der wxWidgets Bibliothek zu verwenden, was in den meisten Fällen auch so implementiert ist. Lediglich die Erzeugung von Sockets, die unter Microsoft Windows und GNU/Linux ziemlich unterschiedlich funktioniert, wurde mit Hilfe von Präprozessoranweisungen gelöst. Deshalb ist es unbedingt notwendig die Präprozessorvariable `WIN32` bei der Kompilation des CSPmedclients unter Microsoft Windows zu definieren. Folgende Präprozessorvariablen werden aktuell berücksichtigt:

- `WIN32`: Wichtig für den Socket-relevanten Code, Behandlung der Fehlervariable (`errno`) und plattformabhängige Typdefinitionen
- `DEBUG`: Generelle Debugmeldungen
- `RPC_DEBUG`: RPC-relevante Debuginformationen
- `RPC_CALL_DEBUG`: falls diese Variable definiert ist, wird jeder behandelte RPC-Call auf `stdout` beziehungsweise jeder noch nicht implementierte Call in einer Messagebox ausgegeben

6.4.1 Kompilation zur Erzeugung einer unter GNU/Linux ausführbaren Datei

Für die Kompilation unter GNU/Linux stehen insgesamt 3 Makefiles zur Verfügung. Bevor der eigentliche CSPmedclient kompiliert werden kann, müssen zuerst die libcsprpc Bibliotheken (die RPC-Implementierung) und die Bibliotheken für das CT-Protokoll erstellt werden. Für diesen Zweck existiert im Verzeichnis `./libcsprpc` ein Makefile, das die erstellten `.o` und `.a` Files im Verzeichnis `./libcsprpc/lnxobj` abstellt. Weiters existiert im Verzeichnis `./libwwmssvc` der vom `rpcgen` Generator erzeugte Quellcode zum Austausch der `wwmscmd` Datenstruktur über das Netzwerk. Diese Datenstruktur beinhaltet sämtliche Informationen, die der CSPmedclient benötigt, um die Elemente zur Benutzerinteraktion darzustellen und wird mit Hilfe der libcsprpc Bibliothek über das Netzwerk versandt (näheres dazu ist in 3.1.3 erläutert). Die serverseitigen Programme sind ebenfalls unter Anderem gegen diese Bibliotheken gelinkt, um eine Kommunikation herstellen zu können. Sind die oben angegebenen Bibliotheken kompiliert, kann man mit Hilfe des Makefiles im Verzeichnis des CSPmedclients

diesen kompilieren. Ein einfaches `make` ohne Angabe irgendwelcher Targets kompiliert und linkt die verfügbaren Codedateien und Bibliotheken. Folgende Targets existieren im Makefile des CSPmedclients:

- `all`: Standardtarget, kompiliert die Quellcodedateien des CSPmedclients und linkt gegen die (existierenden) libcsprpc und libwwmssvc Bibliotheken. Achtung: Die letzteren zwei Bibliotheken müssen zuvor manuell (siehe oben) mit Hilfe der Makefiles in den entsprechenden Verzeichnissen erstellt werden.
- `clean`: Löscht alle Objektdateien sowie die ausführbare `CSPlin` Datei. Weiters werden auch alle Objekt- und Bibliotheksdateien aus dem `./libcsprpc` und dem `./libwwmssvc` Verzeichnis gelöscht. Dieses Target lässt das Verzeichnis `./VC++` mit den Objekt- und ausführbaren Dateien der Microsoft Windows-Version des CSPmedclients unberührt.
- `cacheclean`: Löscht alle Dateien aus dem `./System` Verzeichnis. Hier werden Formulardaten gespeichert, um sie nicht ständig neu vom Server abholen zu müssen falls sie nicht verändert wurden (also im Prinzip ein Cache).
- `rebuild`: Erstellt den gesamten CSPmedclient inklusive libcsprpc und libwwmssvc Bibliotheken neu.

Falls der Kompilationsvorgang erfolgreich war, findet man im Verzeichnis des CSPmedclients die ausführbare Datei `CSPlin` vor.

6.4.2 Kompilation zur Erzeugung einer unter Microsoft Windows ausführbaren Datei

Die Kompilation des CSPmedclients ist unter Microsoft Windows mit Hilfe Visual C++ möglich, im Verzeichnis `./VC++` existieren entsprechende `.dsw` Dateien. Ähnlich der Kompilation unter GNU/Linux muss zuerst die Bibliothek für die RPC-Kommunikation kompiliert werden. Dies geschieht mittels der Visual C++ Projektdatei `csprpc.dsw`. Danach erfolgt die Kompilierung des eigentlichen CSPmedclients mittels der Projektdatei `CSPwin.dsw`. Beide Projektdateien sind derzeit nur für den Debug-Build eingerichtet.

6.4.3 Ergänzende Anmerkungen

- Eventuell muss in den Makefiles beziehungsweise in den `.dsw` Dateien der Ort der wxMSW Bibliothek und deren Includedateien angepasst werden, da dieser momentan „hardcoded” angegeben ist.
- Die ausführbaren Dateien heißen auf beiden Plattformen unterschiedlich. Das GNU/Linux Makefile erzeugt die ausführbare Datei `CSPlin`, Microsoft Visual C++ erzeugt ein `CSPwin.exe`. Das Format und der Dateiname der Bibliotheken sind natürlich ebenfalls unterschiedlich.

- Alle benötigten Dateien für eine Kompilation unter GNU/Linux sind bereits im CSPmedclient Verzeichnis und dessen Unterverzeichnissen enthalten, das heißt, in keinem der Makefiles wird mehr auf Verzeichnisse unterhalb von `/udsk_ent` verwiesen. Die Visual C++ Projektdateien dagegen setzen die Verzeichnisse `r:\libgnurpc\win32incl` und `W:\CSP\CPP\COMMON` voraus.

6.5 Screenshots

Die folgenden Abbildungen zeigen beispielhaft Zustände der Benutzerschnittstelle, wie sie vom CSPmedclient dargestellt wird. Dabei ist der MFC-basierte CSPmedclient immer auf der linken Seitenhälfte abgebildet und sein GNU/Linux GTK+ basiertes Pendant auf der rechten Seitenhälfte.

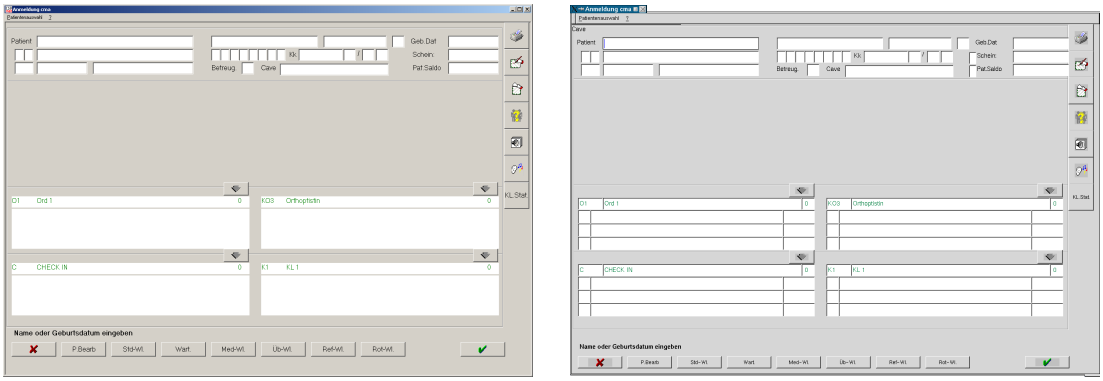

Abbildung 7: Einstiegsbildschirm der Anmeldungsapplikation

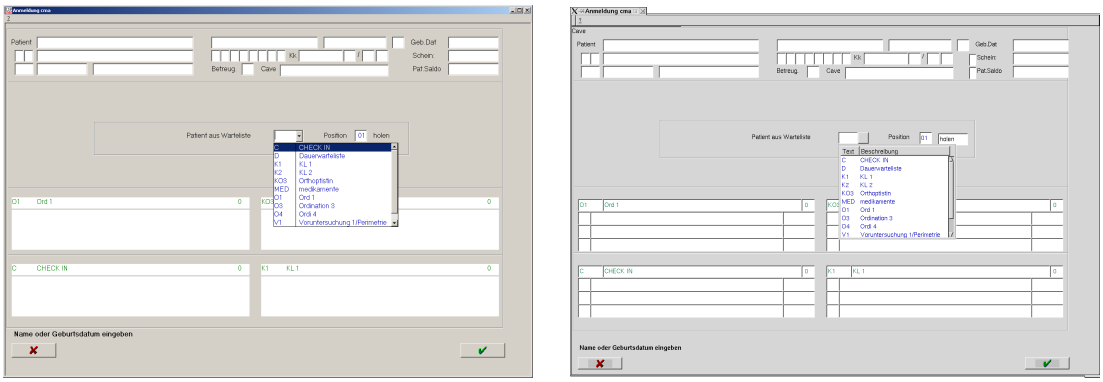

Abbildung 8: Auswahl einer Warteliste

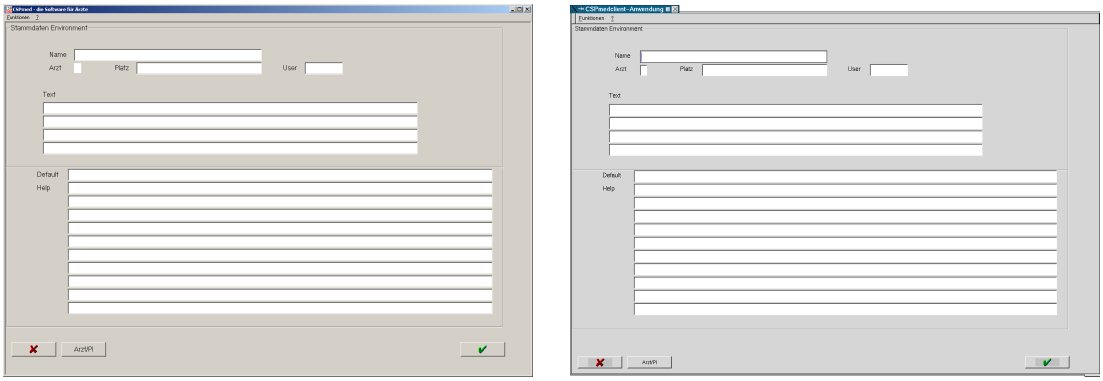

Abbildung 9: Stammdatenprogramm zur Parametrierung von CSPmed, Hauptschirm

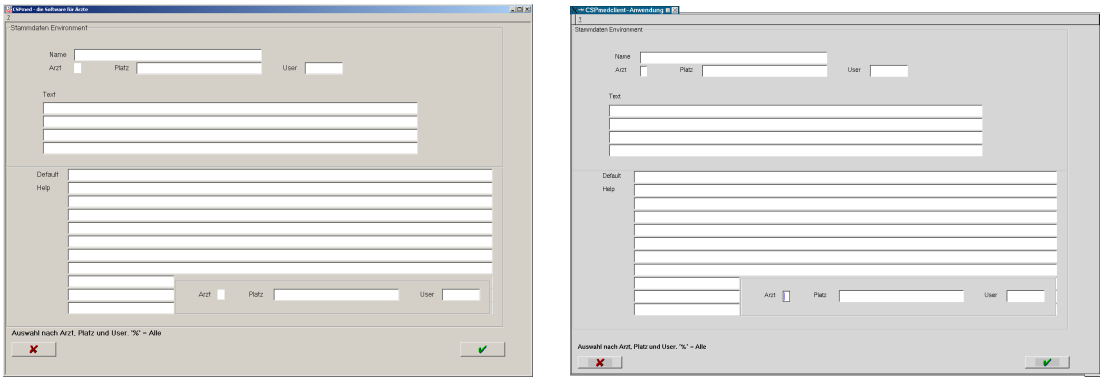

Abbildung 10: Stammdatenprogramm zur Parametrierung von CSPmed, Variablenauswahl

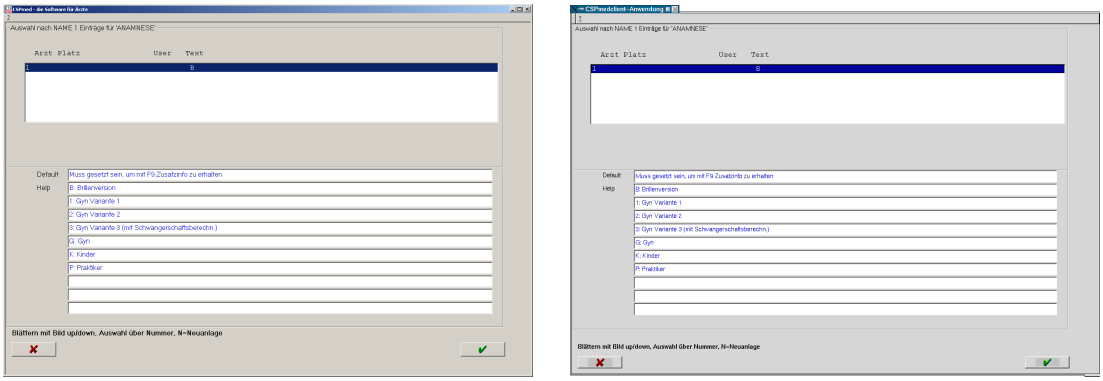

Abbildung 11: Stammdatenprogramm zur Parametrierung von CSPmed, Anzeige der Variablenwerte

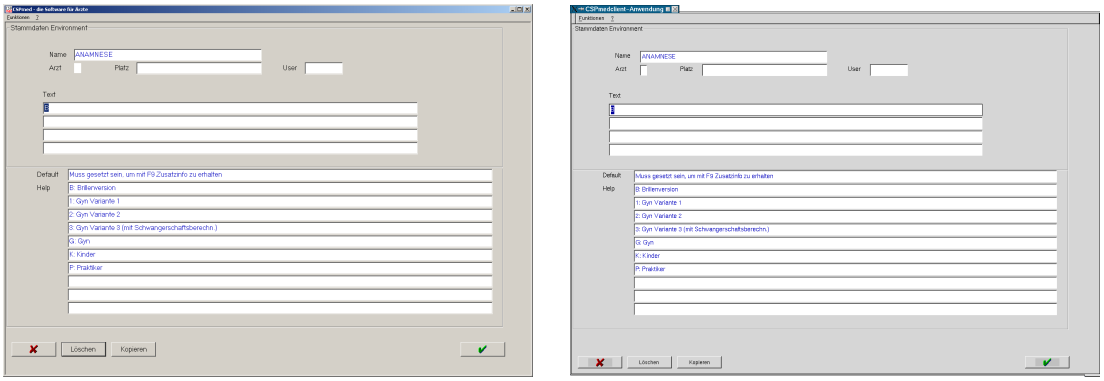

Abbildung 12: Stammdatenprogramm zur Parametrierung von CSPmed, Ändern der Variablenwerte

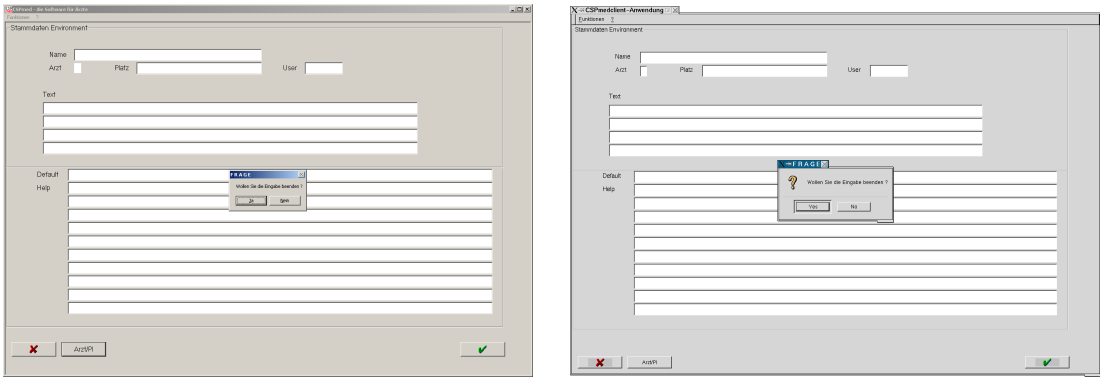

Abbildung 13: Stammdatenprogramm zur Parametrierung von CSPmed, Beenden

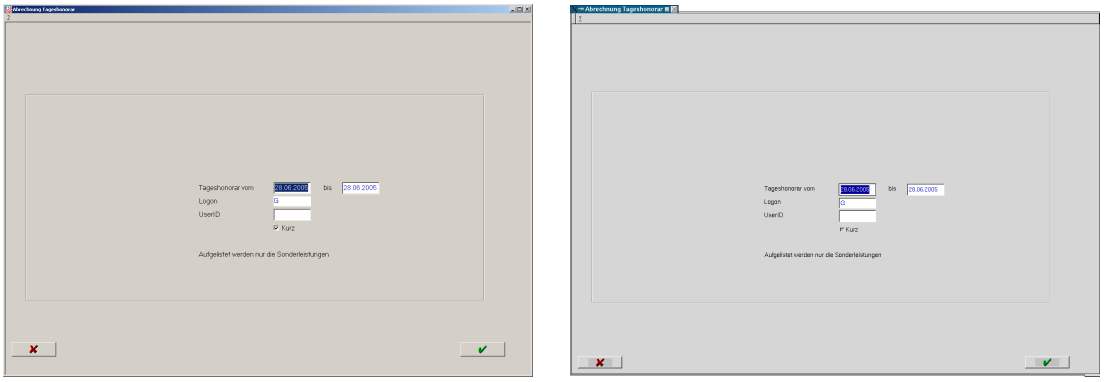

Abbildung 14: Abrechnungsprogramm Tageshonorar

7 Teststrategien

7.1 Die Problematik

Um Fehler in einem Softwareprojekt zu finden, muss der Programmcode getestet werden, wofür es eine ganze Reihe von Möglichkeiten gibt. Primär werden zur Suche von Fehlern im Programmcode Werkzeuge, wie beispielsweise Debugger, genutzt. Aber auch Debugausgaben auf der Konsole können die ensprechenden Fehler ans Tageslicht bringen und somit ihren Zweck erfüllen. Mit Debuggern gestaltet sich die Suche recht mühsam, da man sich meist nur sehr langsam und Schritt für Schritt an den Fehler herantasten kann. Debugausgaben machen den Code unübersichtlich und führen letztlich dazu, dass mit zunehmender Komplexität des Programmcodes auch die zur Fehlersuche eingebauten Kontrollausgaben nicht mehr zu überblicken sind. Das einmalige Testen von Software ist aber bei Weitem nicht befriedigend, der Entwicklungsprozess ist mit der Fertigstellung von Programmteilen längst nicht beendet. So führen Änderungen am Programmcode, sei es nun zu Erweiterungszwecken oder im Zuge einer Optimierungsmaßnahme, immer wieder dazu, dass Funktionen oder Methoden, die zuvor ausgetestet wurden und fehlerfrei funktionierten, plötzlich nicht mehr korrekt arbeiten. Auch kommt es vor, dass sich Änderungen, die bei der Ausmerzung von Programmfehlern getätigt werden, auf andere Programmteile auswirken und somit an anderen Stellen im Code zu Fehlfunktionen führen. Durch automatisierte Tests kann man erreichen, dass man genau auf die Stelle aufmerksam gemacht wird, an der nun Fehlfunktionen auftreten, denn erfahrungsgemäß ist das immer dort, wo man als Letztes danach sucht.

7.2 Unit-Tests

Bei der oben erwähnten Problematik setzen nun die sogenannten Unit-Tests ein. Mit ihrer Hilfe lassen sich solche automatisierten Tests implementieren und jederzeit wiederholen. Ein Unit-Test prüft dabei immer nur einen sehr kleinen und autarken Teil des Software-Systems, wie zum Beispiel eine einzelne Funktion oder Klassenmethode. Dabei wird bei jedem Test die zu testende Funktion oder Methode mit Testdaten (Parametern) konfrontiert und deren Reaktion auf diese Testdaten geprüft. Die zu erwartenden Ausgabewerte werden nun mit den von der jeweiligen Funktion oder Methode gelieferten Ergebnisdaten verglichen. Stimmt das erwartete Ergebnis mit dem zurückgelieferten Ergebnis der Funktion oder Methode überein, so gilt der Test als bestanden. Bei der Wahl von zu testenden Methoden sollte man beachten, dass nicht jede Methode eines Tests bedarf. So ist es relativ sinnlos Set- und Get-Methoden zu testen, da diese meist nur eine recht triviale Funktionalität aufweisen. Private Methoden von Klassen können dagegen, aufgrund ihrer Sichtbarkeit, ohnehin nur indirekt getestet werden. Dabei ist festzuhalten, dass beim Testen mittels Unit-Tests die zu testende Klasse als eine Art Black Box angesehen wird. Es interessiert bloß, ob das aufgrund der Testdaten zurückgelieferte Ergeb-

nis korrekt ist oder nicht. Deshalb ist es auch nicht möglich, geschweige denn im Rahmen eines Unit-Tests sinnvoll, private Methoden zu testen. Sie werden als korrekt angenommen, wenn der Unit-Test erfolgreich ist und müssen korrigiert werden, falls der Test versagt. Da private Methoden im Gegensatz zu den öffentichen Schnittstellenmethoden häufigen Änderungen an der Signatur unterworfen sind, wäre es außerdem eine äußerst mühsame Arbeit, ständig auch die Tests entsprechend anzupassen.
Ein Test besteht im Allgemeinen aus einer ganzen Reihe von Testfällen, die nicht nur ein Parameter-Ergebnis-Paar prüfen, sondern gleich mehrere. Welche Szenarien der Entwickler nun als testwürdig erachtet, bleibt ihm überlassen. Sinnvoll ist es im Allgemeinen, Funktionen und Methoden mit Parametern zu testen, die typischerweise bei deren Aufruf auftreten. Auch die Betrachtung von Grenzwerten (extrem große oder kleine Werte) oder besonderen Werten (Null-Zeiger, Objekte in speziellen Zuständen), bei den unterschiedlichen als Parameter genutzten Datentypen, ist sinnvoll. Liefern alle diese Testszenarien erwartungsgemäß die korrekten Werte, so kann der Entwickler davon ausgehen, dass seine Implementierung der Funktion oder Methode korrekt ist. Der Entwickler implementiert meist zu jeder Klasse eine entsprechende Testklasse, welche die jeweiligen Funktionalitäten der zu testenden Methoden verifiziert. Einzelne Testklassen werden zusätzlich zu sogenannten Test-Suiten zusammengefasst, mit denen dann nicht nur einzelne Funktionen oder Klassen automatisch getestet werden, sondern ganze Softwaresysteme auf einmal. Diese Vorgehensweise führt dazu, dass der Entwickler Funktionen und Methoden möglichst einfach hält und Objektstrukturen sauber entwirft, um entsprechend einfache Tests dazu implementieren zu können. Bei der Entwicklung solcher Tests fallen auch unglücklich gewählte Schnittstellen sofort auf, wodurch die Qualität und auch die Wartbarkeit des Codes steigt. Wird der Code zu komplex, so muss er gegebenenfalls vereinfacht (refaktorisiert) werden. Somit ist die Implementierung von Unit-Tests kein notwendiges Übel, sondern sorgt für einen verbesserten und effektiveren Entwicklungsprozess. Man darf dabei allerdings die Gefahr einer unglücklichen Testauswahl nicht außer Acht lassen. Sind die vom Entwickler gewählten Tests unzureichend oder gar falsch, liefern aber ein positives Ergebnis, so führt diese trügerische Sicherheit zu großen Problemen. Der Entwickler vertraut seinem Code so sehr, dass er erst nach langer Suche auf die Idee kommen wird, in seinem positiv getesteten Code nach Fehlern zu suchen.
Neben *JUnit* (für die Java Programmiersprache), dem bekanntesten und wohl auch meistgenutzten Vertreter unter den Frameworks für Unit-Tests, gibt es Implementierungen für die verschiedensten Programmiersprachen. *CppUnit* beispielsweise ist ein solches Framework zur Programmierung von Software-Tests nach dem Prinzip der Unittests für die Programmiersprache C++.

7.3 CppUnit

Bei CppUnit handelt es sich um eine Portierung des Vorbildes JUnit nach C++, die auf einer ganzen Reihe von Unix-Derivaten, sowie unter Windows läuft. Dabei

Algorithm 10 Headerdatei einer Testklasse

```
#ifndef BRUCHTEST_H
#define BRUCHTEST_H
#include <cppunit/TestFixture.h>
#include <cppunit/extensions/HelperMacros.h>
#include "Bruch.h"
using namespace std;
class bruchtest : public CPPUNIT_NS :: TestFixture
{
    CPPUNIT_TEST_SUITE (bruchtest);
    CPPUNIT_TEST (addTest);
    CPPUNIT_TEST (subTest);
    CPPUNIT_TEST (exceptionTest);
    CPPUNIT_TEST (equalTest);
    CPPUNIT_TEST_SUITE_END ();
    public:
        void setUp (void);
        void tearDown (void);
    protected:
        void addTest (void);
        void subTest (void);
        void exceptionTest (void);
        void equalTest (void);
    private:
        Bruch *a, *b, *c, *d, *e, *f, *g, *h;
};
#endif
```

werden von diesem Framework, neben verschiedenen Unix Compilern wie dem GNU C++-Compiler, auch Visual C++ ab Version 6.0 und Borland C++ unterstützt. Dokumentation ist unter [14] einsehbar, hier sollen lediglich Beispiele zur Illustration der Möglichkeiten von CppUnit gegeben werden. Die folgenden angegebenen Codefragmente dienen lediglich zur Veranschaulichung und sollen nur einen Überblick über die Verwendung von CppUnit und die dafür notwendigen Schritte geben, weshalb sie auch nicht im Detail erläutert werden und der Leser auf [14] verwiesen sei. Algorithmus 10 zeigt die Definition der Testklasse zu einer Klasse, die dazu verwendet wird, um mit Brüchen umzugehen. Man sieht, dass die einzelnen Tests als protected Methoden definiert sind und mittels Makro, das vom CppUnit Framework zur Verfügung gestellt wird, zu einer Testsuite hinzugefügt werden. Unter einer Testsuite versteht man eine Sammlung von Tests (Testreihe), die meist mit Instanzen der zu testenden Klasse durchgeführt werden sollen.
Die Methoden `setUp()` und `tearDown()` dienen der Vor- und Nachbereitung der Tests, sie bestimmen die Rahmenbedingungen der Tests. In `setUp()` werden

zum Beispiel die Bruchinstanzen erzeugt, welche in `tearDown()` gelöscht werden. Die Implementierung der Testklasse könnte wie in den Algorithmen 11 und 12 gelöst werden.
Die Tests operieren allesamt auf den in `setUp()` erzeugten Objektinstanzen und geben jeweils die auszuführende Operation und das erwartete Ergebnis an. Um die Tests automatisiert ablaufen zu lassen, benötigt man noch ein Programm (Algorithmus 13), das dies bewerktstelligt und in Algorithmus beispielhaft implementiert ist.
Hierbei werden alle Tests ausgeführt, deren Resultate gesammelt und für jeden einzelnen Test ausgegeben, ob er erfolgreich war oder nicht. Der Vorteil liegt darin, dass das Testen somit in den Compile-Build-Prozess integriert werden kann und jedesmal, wenn der Programmcode neu generiert ist, automatisch getestet wird. Dabei ist es auch möglich, bei fehlgeschlagenen Tests die genaue Zeilennummer des Fehlers anzuzeigen, was eine schnellere Korrektur ermöglicht.

7.4 Anwendung auf den CSPmedclient

Das Testen von Applikationen mit Hilfe von CppUnit erfordert, wie man an obigem Beispiel sehen kann, sehr klar definierte Schnittstellen der Methoden und Klassen, die jeweils möglichst unabhängig getestet werden können. Solche sauber definierten Schnittstellen existieren in der aktuellen Version des CSPmedclients leider nicht, das Design ist daher etwas verbesserungsbedürftig. Es wäre hingegen durchaus möglich und meiner Meinung nach auch empfehlenswert, etwa eine Klasse zur Umrechnung von Koordinaten einzuführen, die die in den Fenster- und Formulardateien angegebenen Koordinaten in ihr jeweiliges Bildschirmäquivalent umrechnet. Diese Klasse könnte man dann sehr gut in das CppUnit Framework integrieren. Es bleibt allerdings fraglich, inwiefern es trotz verbesserter Struktur des CSPmedclients sonst noch möglich ist, CppUnit einzusetzen. Dadurch, dass eigentlich eine Benutzerschnittstelle automatisiert generiert wird, versagt das Konzept spätestens an dem Punkt, an dem Benutzereingaben in die generierte Benutzerschnttstelle notwendig sind und somit automatisches Testen nicht mehr möglich ist. Was man allerdings tun kann ist, möglichst alle Schritte die zwischen Empfang der RPC Kommandostruktur und der Darstellung eines Elements am Bildschirm notwendig sind, in ein geeignetes Klassenschema zu integrieren und darauf zu achten, dass alle Methoden ein testbares Interface mit genau definierter Ein- und Ausgabe erhalten.
Der Vollständigkeit halber sei jeoch erwähnt, dass sehrwohl Testumgebungen existieren, mit deren Hilfe man zum Beispiel Benutzerschnittstellen und Datenbankdesigns automatisiert testen kann. Die Anforderungen an die Testwerkzeuge können je nach Projekt sehr unterschiedlich sein, sodass es auch notwendig sein kann das Testwerkzeug selbst zu implementieren. Hierzu und auch was die allgemeine Problematik des automatisierten Testens von Benutzerschnittstellen anbelangt bietet [16] nützliche Informationen. Ein solches oben angesprochenes Testwerkzeug wird in Fällen eingesetzt in denen die Gesamtheit der von einem Softwaresystem

Algorithm 11 Implementierung einer Testklasse

```
#include "bruchtest.h"
CPPUNIT_TEST_SUITE_REGISTRATION (bruchtest);
void bruchtest :: setUp (void)
{
    // Vorbereitungen treffen, indem Objekte
    initialisiert werden
    a = new Bruch (1, 2);
    b = new Bruch (2, 3);
    c = new Bruch (2, 6);
    d = new Bruch (-5, 2);
    e = new Bruch (5, -2);
    f = new Bruch (-5, -2);
    g = new Bruch (5, 2);
    h = new Bruch ();
}
void bruchtest :: tearDown (void)
{
    // Objekte alle wieder loeschen
    delete a; delete b; delete c;
    delete d; delete e; delete f; delete g; delete h;
}
void bruchtest :: addTest (void)
{
    // Ergebnisse der Subtraktionen pruefen
    CPPUNIT_ASSERT_EQUAL (*a + *b, Bruch (7, 6));
    CPPUNIT_ASSERT_EQUAL (*b + *c, Bruch (1));
    CPPUNIT_ASSERT_EQUAL (*d + *e, Bruch (-5));
    CPPUNIT_ASSERT_EQUAL (*e + *f, Bruch (0));
    CPPUNIT_ASSERT_EQUAL (*h + *c, Bruch (2, 6));
    CPPUNIT_ASSERT_EQUAL (*a + *b + *c + *d + *e
        + *f + *g + *h, Bruch (3, 2));
}
```

Algorithm 12 Implementierung einer Testklasse, Fortsetzung

```
void bruchtest :: subTest (void)
{
    // Ergebnisse der Subtraktionen pruefen
    CPPUNIT_ASSERT_EQUAL (*a - *b, Bruch (-1, 6));
    CPPUNIT_ASSERT_EQUAL (*b - *c, Bruch (1, 3));
    CPPUNIT_ASSERT_EQUAL (*b - *c, Bruch (2, 6));
    CPPUNIT_ASSERT_EQUAL (*d - *e, Bruch (0));
    CPPUNIT_ASSERT_EQUAL (*d - *e - *f - *g - *h,
        Bruch (-5));
}
void bruchtest :: equalTest (void)
{
    // Test erfolgreich, wenn er true liefert
    CPPUNIT_ASSERT (*d == *e);
    CPPUNIT_ASSERT (Bruch (1) == Bruch (2, 2));
    CPPUNIT_ASSERT (Bruch (1) != Bruch (1, 2));
    // Beide Ausdruecke muessen dasselbe Ergebnis
liefern
    CPPUNIT_ASSERT_EQUAL (*f, *g);
    CPPUNIT_ASSERT_EQUAL (*h, Bruch (0));
    CPPUNIT_ASSERT_EQUAL (*h, Bruch (0, 1));
}
```

Algorithm 13 Testprogramm

```
#include <cppunit/CompilerOutputter.h>
#include <cppunit/extensions/TestFactoryRegistry.h>
#include <cppunit/TestResult.h>
#include <cppunit/TestResultCollector.h>
#include <cppunit/TestRunner.h>
int main (int argc, char* argv[])
{
    // Informiert Test-Listener ueber Testresultate
    CPPUNIT_NS :: TestResult testresult;
    // Listener zum Sammeln der
    // Testergebnisse registrieren
    CPPUNIT_NS :: TestResultCollector
collectedresults;
    testresult.addListener (&collectedresults);
    // Test-Suite ueber die Registry
    // im Test-Runner einfuegen
    CPPUNIT_NS :: TestRunner testrunner;
    testrunner.addTest(CPPUNIT_NS ::
TestFactoryRegistry
        :: getRegistry ().makeTest ());
    testrunner.run (testresult);
    // Resultate im Compiler-Format ausgeben
    CPPUNIT_NS :: CompilerOutputter compileroutputter
        (&collectedresults, std::cerr);
compileroutputter.write ();
    // Rueckmeldung, ob Tests erfolgreich waren
    return collectedresults.wasSuccessful () ? 0 : 1;
}
```

implementierten Funktionen ausschließlich über die grafische Benutzerschnittstelle erschlossen werden kann. Im Fall der CSPmed Applikation würde es daher Sinn machen, es zum Testen einer Serverapplikation einzusetzen, die zur Ausgabe den CSPmedclient verwendet. Alleine zum Testen des CSPmedclients wäre ein automatisiertes Werkzeug zum Testen der Benutzschnittstelle fehl am Platze, da dieser ja im Prinzip „nur" je nach Bibliotheksaufruf entsprechende grafische Elemente erzeugt und mit der eigentlichen Programmlogik, also dem was die Applikation für den Benutzer leistet, nichts zu tun hat.
Solange CppUnit aufgrund falscher oder fehlender Klassenstruktur noch nicht eingesetzt werden kann, oder wenn man spezifische Fälle testen möchte, hat man auch die Möglichkeit, spezielle Serverprogramme zu implementieren, die ausschließlich die zu testende Funktionalität (also die veränderten Methoden) benutzen und „manuell" zu überprüfen, ob das Ergebnis zufriedenstellend ist. Eventuell kann man auch eine Applikation implementieren, die „handgemachte" RPC Kommandostrukturen an den CSPmedclient sendet und seine Reaktion darauf mit einer Referenz vergleicht. Dieser Ansatz wäre dann halbautomatisiert, da der CSPmedclient die in den Strukturen enthaltenen Kommandos ausführen und die Benutzerschnittstelle erzeugen würde sowie (in den meisten Fällen) nach einer Eingabe des Benutzers ein Rückgabewert zum Vergleich mit einem Referenzwert zurückgegeben würde. Es wäre auch möglich, diese Art zu testen in CppUnit zu integrieren, das mometan implementierte Interface zwischen empfangener RPC-Kommandostruktur und Rückgabewert an den Server würde sich dazu eignen.

8 Refactoring des bestehenden Systems

8.1 Motivation

Laut [12] kann man den Ausdruck „Portierung" folgendermaßen definieren:

> „Unter einer Portierung versteht man in der Softwarebranche den Vorgang, ein Computerprogramm, das unter einem bestimmten Betriebssystem läuft, auch auf anderen Betriebssystemen lauffähig zu machen."

Um eine Portierung vorzunehmen ist es, wie auch im vorliegenden Fall des CSPmedclients, meistens notwendig, den Quellcode des Programms so anzupassen, sodass die vom jeweiligen Betriebssystem angebotenen Dienste verwendet werden können. Oft ist es dabei notwendig, das Programm so umzuschreiben, dass es auf eine andere, (nur) für das neue Zielbetriebssystem verfügbare Bibliothek zugreift. Damit gehen umfangreiche Quellcodeänderungen einher, die sich in den verwendeten Algorithmen und oft auch in der gesamten Architektur des Programms, das heißt der Klassenhierarchie, wiederspiegeln.
Bei der Portierung des CSPmedclients auf GNU/Linux musste ebenfalls der Quellcode der Anwendung so angepasst werden, dass anstatt der Microsoft Foundation Classes (MFC) als Toolkit zur Generierung und Darstellung der Oberfläche, das

freie Toolkit wxWidgets verwendet wird. Dabei war jedoch nicht von Beginn an klar, ob es überhaupt möglich ist, den CSPmedclient mit Hilfe von wxWidgets zu implementieren, da noch keine Erfahrungswerte mit diesem Toolkit existierten. Aufgrund der Tatsache, dass es in C++ programmiert werden kann und gewisse Ähnlichkeiten zu MFC aufwies, wurden die Algorithmen und die Klassenhierarchie des MFC-basierten CSPmedclients vom Prinzip her beibehalten und ausschließlich Aufrufe des wxWidgets Toolkit verwendet. Diese Strategie erwies sich als sehr gut geeignet, da man die Funktionalität nach und nach implementieren und davon ausgehen konnte dass, falls eine bestimmte einfache Anwendung hinsichtlich GUI-Erstellung und Kommunikation einwandfrei läuft, auch alle weiteren Anwendungen unterstützt werden können. Nachteilig wirkte sich hingegen aus, dass es zu einer Abnahme der Qualität des Codes kam. Einige Algorithmen waren zur Verwendung mit den MFC ausgelegt und mussten für die Verwendung von wxWidgets entsprechend angepasst werden. Die dadurch vorgenommenen Änderungen brachten Schwachstellen im Bereich der Funktionskapselung (Mitgliedsvariablen und -funktionen) zu Tage deren Behebung grundlegende Eingriffe in den Aufbau des CSPmedclients erfordert hätten. Hätte man diese sofort vorgenommen, hätte man natürlich auch die neue Funktionalität die hinzukam an die geänderte Struktur anpassen müssen, die Übernahme von Code aus der MFC-Applikation wäre bald völlig unmöglich und eine nahezu komplette Neuimplementierung notwendig geworden. Stattdessen wurde, falls immer das notwendig erschien, die alten, suboptimalen Algorithmen und Architektur beibehalten, workarounds geschrieben und als verbesserungswert kommentiert. Damit erhielt man sich den Vorteil des parallelen Debuggens von MFC- und wxWidgets Client und konnte die Abläufe innerhalb der Software leichter verstehen und schneller implementieren. Das Resultat ist eine Applikation, die in ihrem Verhalten größtenteils identisch mit ihrem MFC-Vorbild ist, vom internen Aufbau allerdings noch einige Verbesserungen notwendig hat, um leichter wartbar zu sein.
Da für einen Produktiveinsatz des wxWidgets-basierten CSPmedclients ohnehin noch einige Entwicklungsarbeit notwendig ist, wäre es daher eine gute Gelegenheit, das in [18] dargelegte Konzept des Refactorings anzuwenden. Der Zeitpunkt dafür ist günstig, da der aktuelle Entwicklungsstand des CSPmedclients als prinzipiell funktionsfähig angesehen werden kann, was das Testen von Änderungen, die hauptsächlich der perfektionierenden Wartung dienen, wesentlich leichter macht.

8.2 Was ist Refactoring?

Autor Martin Fowler definiert in [18] Refactoring folgendermaßen:

> „Refactoring (noun): a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour."

Refactoring ist ein Prozess, der die interne Struktur einer Applikation verbessert und gleichzeitig ihr Verhalten nach außen hin nicht beeinflusst. Es ist ein Weg, den

Code zu verbessern nachdem er geschrieben wurde, wobei die Wahrscheinlichkeit, neue Fehler einzubringen, relativ klein ist.
Oft werden Softwareentwicklungsmodelle verwendet bei denen am Anfang der Entwicklung zuerst das Design der Software festgelegt und anschließend implementiert wird. Mit der Zeit wird der Code immer wieder modifiziert und die Struktur, die bei der ursprünglichen Planung festgelegt wurde, immer mehr aufgeweicht. Irgendwann ist ein Punkt erreicht, an dem man nicht mehr von Softwareentwicklung sondern von Hacking sprechen muss. Refactoring verfolgt das umgekehrte Prinzip. Ausgehend von einem schlecht designten, vielleicht sogar chaotischem Code, gelangt man durch Überarbeiten zu einem brauchbaren und gut wartbaren Code. Jeder Schritt, den man unternimmt, ist sehr einfach, man verschiebt etwa eine Methode in eine andere Klasse in die sie besser passt, fasst wiederkehrenden Code in Methoden zusammen und so weiter. In Summe gesehen verbessern all diese kleinen Änderungen das Design der Software, es ist gewissermaßen das genaue Gegenteil der herkömmlichen Softwareentwicklung. Man geht beim Refactoring von einem System aus und verbessert dieses, sodass es alle Anforderungen erfüllt. Andere Softwareentwicklungsmodelle versuchen oft, alle Eventualitäten in der Designphase zu berücksichtigen, was zu unnötig komplexer Software führt, da man später zu der Erkenntnis gelangt, dass ein einfacheres Modell bei Weitem ausreichend gewesen wäre. Je mehr man von einem Projekt implementiert hat, desto mehr steigt auch das Wissen darüber und es wird klar, dass manche Designentscheidungen falsch sind. Deshalb liegt die zentrale Aufgabe beim Refactoring darin, neu gewonnene Erkenntnisse sofort beim Implementieren in das Design zurückfließen zu lassen, es ist kein herkömmlicher „top-down" Vorgang sondern „bottom-up". Man braucht beim Refactoring jedoch eine plausible erste Lösung, die man als Ausgangspunkt für weitere Modifikationen betrachtet. Falls sich diese Lösung, und das wird sie mit an Sicherheit grenzender Wahrscheinlichkeit tun, als unzureichend erweist, ist das überhaupt kein Problem, sie wird durch Refactoring einfach verbessert, solange bis sie wieder passt. Danach implementiert man weiter, bis man wieder zu einem Punkt gekommen ist, an dem man das Design in Frage stellt. Dann wird abermals refaktoriert und so weiter.
Wichtig ist, zwischen Implementierung (also Hinzufügen von neuer Funktionalität) und Refactoring zu unterscheiden. Wenn man Funktionalität hinzufügt, ändert man existierenden Code nicht. Wenn man hingegen Refacturing durchführt (also „refaktoriert") implementiert man keine neuen Funktionen sondern verbessert lediglich die Struktur des Codes. Softwareentwicklung unter Verwendung von Refactoring ist ein ständiger Wechsel der Tätigkeiten vom Implementieren zum Refactoring zum Testen und wieder weiter beim Implementieren.
Laut Definition von Refactoring soll das „beobachtbare Verhalten" der Software, die refaktoriert wird, nicht verändert werden, deswegen ist das Testen ein extrem wichtiger Bestandteil dieser Strategie. Ohne Testen würde man oft meinen „Verbesserungen" an der Software zu machen, durch im Zuge des Refactorings eingeschleppte Fehler würde die Software allerdings im kürzester Zeit unbrauchbar gemacht werden. Deshalb ist es wichtig, nach jeder Änderung zu testen, ob sich

am Programmverhalten etwas verändert hat und falls dem so ist, die Änderung rückgängig zu machen. Idealerweise geht man beim Hinzufügen neuer Funktionalität derart vor, dass man zuerst die Tests implementiert und erst danach den Code. Die Tests sollten automatisiert ablaufen und den Programmierer warnen, falls sie fehlgeschlagen sind. Nur auf diese Weise kann man sicherstellen, dass die Entwicklung zügig vorangeht und jeder Entwickler auch die Disziplin aufbringt, seine Tests durchlaufen zu lassen. Testframeworks wie zum Beispiel *JUnit* unter Java oder dessen C++ Gegenstück *CppUnit* erleichtern das Testen sehr und machen es zu einem integralen Bestandteil der Softwareentwicklung.
Wichtige Zeitpunkte zu denen man wieder einen Refactoring-Durchgang starten sollte wären zum Beispiel wenn eine Funktion hinzugefügt, ein Fehler korrigiert werden soll oder während Codereviews. Natürlich gibt es auch Situationen, in denen man Refactoring nicht anwenden sollte, nämlich dann wenn eine komplette Neuimplementierung besser ist. Das ist vor allem dann der Fall, wenn der Code von dem man ausgeht einfach zu unübersichtlich ist oder überhaupt nicht funktioniert. Weiters sollte man kurz vor Deadlines nicht mit dem Refactoring beginnen, da der Produktivitätsgewinn durch das Refactoring erst nach der Deadline zum Tragen kommen würde.
Natürlich muss man, um überhaupt Nutzen aus Refactoring ziehen zu können, ein Gespür dafür haben, welcher Code „schlecht" ist, also zu Problemen führen kann und deshalb in eine bessere Form umgewandelt werden muss. Schlechtes Design macht sich unter Anderem durch die in Tabelle 6 erwähnten Eigenschaften bemerkbar, eine umfangreichere Liste ist in [18], Seite 75ff nachzulesen.
Zu allen Vorgängen des Refactorings existert in [18] eine detaillierte Beschreibung samt Beispielen für deren Anwendung.

8.3 Anwendung auf den CSPmedclient

Das Auseinandersetzen mit dem Konzept des Refactorings ist auf jeden Fall eine lohnenswerte Sache bei der Implementierung des CSPmedclients. Die Tatsache, dass bereits eine funktions- und lauffähige Applikation vorliegt die in einer objektorientierten Programmiersprache geschrieben ist, lädt geradezu ein, Refactoring anzuwenden. Die in [18] zahlreich vorhandenen Hinweise zur Verbesserung des Designs sind eine gute Hilfe für Entwickler, die in der objektorientierten Programmierung weniger Praxiserfahrung haben. Das automatisierte Testen des CSPmedclients ist jedoch nicht in jedem Fall möglich, da eine grafische Benutzerschnittstelle generiert wird, wofür man andere Testmethoden einsetzen muss. Für alle anderen Klassen und Methoden ist es jedoch möglich mit Hilfe von CppUnit Tests zu erstellen, die automatisiert abgearbeitet werden können.

Zuviel Funktionalität in einer Methode	Durch lange Methoden leidet die Übersichtlichkeit, daher Aufspalten in mehrere kurze Methoden.
Copy&Paste von Code	Dies soll auf jeden Fall vermieden werden, da sich Inkonsistenzen bei nachträglichen Änderungen des duplizierten Codes ergeben können. Stattdessen sollte man aus diesem Code Methoden generieren, die dann an den entsprechenden Stellen aufgerufen werden.
Lange Parameterlisten von Methoden	Diese sind oft schwierig zu verstehen und können durch Änderungen inkonsistent werden. Daher ist es oft besser, die benötigten Daten von anderen Objekten anzufordern. Außerdem sind in objektorientierten Sprachen sowieso die meisten Daten, die eine Methode braucht, in dem Objekt vorhanden, in dem sich die Methode befindet.
Datenklumpen	Datenfelder treten oft immer in der gleichen Kombination auf, man kann dies zu Klassen zusammefassen.
Switch Statements	Können eventuell durch Polymorphismus erstetzt werden
Temporäre Felder	Eine Instanzvariable einer Klasse ist nur in manchen Fällen auch tatsächlich mit sinnvollen Werten gefüllt, was verwirrend sein kann. Hier ist es besser, eine eigene Klasse mit solchen Feldern zu erzeugen.
Verkettungen von Aufrufen	Lange Ketten von Aufrufen um Daten von Objekten zu erhalten sollten vermieden werden. Zum Beispiel könnte man Person→GetAbteilung→GetChef durch Person→GetChef ersetzen
Unpassende Intimität	Vielfach werden zu freizügige Kriterien bei den Zugriffsregeln auf Klassen angewendet. Um eine klare Schnittstelle einer Klasse zu definieren, muss der Zugriff sehr restriktiv gestaltet werden damit ungewünschte Nebeneffekte bei der Verwendung der Klasse ausbleiben.
Datenklassen	Klassen, die nur aus Feldern und deren Get- und Setmethoden bestehen, werden oft besser mit den Klassen, die diese Methoden aufrufen zusammengelegt.
Kommentare	Kommentare sind meistens eine hilfreiche Sache, doch sie dürfen nicht dazu verwendet werden, schlechtes Design zu beschönigen. Methoden- und Variablennamen sollten sprechend gewählt werden, der Code sollte selbsterklärend sein. Kommentiert sollte nicht, **was** eine Methode macht, sondern **warum**.

Tabelle 6: Kennzeichen schlechten Softwaredesigns

9 Modernisierung von Legacy Systemen

9.1 Definition

Unter dem Begriff Legacy System versteht man existierende Computersysteme oder Applikationen die historisch gewachsen sind und noch immer benutzt werden. Solche Systeme sind vor allem aber natürlich nicht ausschließlich bei Organisationen im Finanzwesen anzutreffen, da in diesem Bereich die elektronische Datenverarbeitung schon seit geraumer Zeit eingesetzt wird. Legacy Systeme sind aus folgenden Gründen als problematisch einzuschätzen:

Einsatzfähigkeit: Sie benutzen veraltete Hardware für die es schwierig und teuer sein kann Ersatzteile zu erhalten.

Verständnis des Systems: Der Wartung, Erweiterung und Verbesserung sind starke Grenzen gesetzt, da die ursprünglichen Entwickler nicht mehr verfügbar sind und ein umfassendes Verständnis des Systems in der Organisation fehlt. Durch die vielen Jahre in denen Wartungsarbeiten durchgeführt wurden, wird die ursprüngliche Architektur meist aufgeweicht und durch zunehmende Komplexität schwieriger zu verstehen. Dies kann durch schlampige und fehlende Dokumentation noch verschlimmert werden.

Erweiterbarkeit: Die Integration mit neuen Systemen erweist sich als schwierig da in Legacy Systemen oft komplett andersartige Technologien zum Einsatz kommen als solche die heute verfügbar sind. Die Tatsache dass Legacy Systeme oft hochverfügbar sein müssen ist oft ein Grund für fehlende Motivation für Wartungsarbeiten da die Wahrscheinlichkeit von Fehlern aufgrund des fehlenden umfassenden Verständnises des Systems sehr hoch ist.

9.2 Bezug zum CSPmed System

Gemäß der in 9.1 angegebenen Kriterien stellt sich die Frage ob man das CSPmed System als Legacy System bezeichnen kann. Dies ist insofern der Fall als dass einige Teile, wie zum Beispiel die RPC Implementierung oder das WMS existieren deren Entwickler nicht mehr zur Verfügung stehen und deren Dokumentation teilweise sehr schlecht ist, oft erhält man nur durch Kommentarzeilen im Quellcode einen vagen Eindruck davon, wozu der entsprechende Abschnitt dient. Es ist also nicht hauptsächlich die Programmlogik von der Probleme zu erwarten sind sondern eher das Ersetzen von alten proprietären Toolkits und Protokollen die zeitgemäßen Anforderungen nicht mehr entsprechen.
Im Falle des CSPmed Systems ist es notwendig die Software auf der Client- und auf der Serverseite zu unterscheiden. Der serverseitige Teil ist der bei Weitem ältere Teil der die Programmlogik implementiert. Obwohl diese Logik auch teilweise schlecht dokumentiert und unverständlich ist, ist es doch durch laufende kleine

Verbesserungen möglich, die Übersichtlichkeit zum Beispiel durch Kapselung in Module zu erhöhen. Anders gestaltet sich der Wechsel auf eine andere, moderne Möglichkeit, die Benutzerschnittstelle darzustellen. Hierbei ist es vom Aufwand her nicht vertretbar, alle Applikationen zu modifizieren, da dies einer kompletten Reimplementierung gleichkommen würde. Um einer Lösung dieses Problems näherzukommen könnte man dokumentierte Strategien zur Migration von Legacy Systemen heranziehen und evaluieren inwiefern diese für das CSPmed System eingesetzt werden können (siehe 9.3.1).
Der clientseitige Teil ist deutlich einfacher zu handhaben, da die verwendete Technologie, sieht man von der Kommunikation mit dem Server ab, weit verbreitet ist und auch in anderen Projekten eingesetzt wird. Die weitere Existenz beziehungsweise Funktionalität des Clients hängt natürlich davon ab für welche Strategie man sich für den Server entscheidet beziehungsweise was die zukünftigen Anforderungen an das CSPmed System sind.

9.3 Strategien

9.3.1 Modernisierung der Schnittstellen

Hierbei untersucht man die Eingabe und Ausgabe in beziehungsweise von einem Legacy System mit dem Ziel eine Kapselung der Schnittstellen zu erreichen. Diese Strategie zielt auf die sichtbaren Teile des Systems ab, so ersetzt man zum Beispiel textbasierte Eingaben in das System durch eine grafische Benutzeroberfläche oder ein Web Frontend und stellt auch die erhaltenen Ausgaben auf diese Art dar. Dabei wird die interne Funktionsweise des Legacy Systems nicht verändert, was sowohl hinsichtlich Programmierfehlern sehr sicher und vom Aufwand her gering ist. Trotzdem erhält man einen enormen Zusatznutzen was die Integration in ein modernes Geschäftsumfeld angeht.
Natürlich müssen bei dieser Strategie die Schnittstellen bekannt und einheitlich sein. Das Hinzufügen einer zusätzlichen Schicht bei der Ein- und Ausgabe erhöht die Komplexität des Systems abermals.

9.3.2 Adaptive Migration

Bei dieser Methode wird versucht, die Applikation auf eine neue Plattform, Sprache oder Datenbank zu migrieren, mit dem Ziel mit technischen Fortschritten mitzuhalten. Obwohl umfangreiche Änderungen notwendig sind um dies zu erreichen, ist der entstehende Code in den meisten Fällen standardkonformer und vorhersehbarer. Es ist hierbei möglich, die grundlegende Struktur der Applikation beizubehalten und gleichzeitig ihren Lebenszyklus zu verlängern sowie die Betriebskosten zu senken.
Diese Strategie birgt jedoch die Gefahr, eine Anwendung zu beschädigen noch bevor sie repariert ist, da Fehler in den Code gelangen können, den man selbst kaum versteht. Weiters produziert man, wenn man Legacy Code neu schreibt, per

Definition wieder Legacy Code sodass das Problem unter Umständen nur auf einen späteren Zeitpunkt verschoben wird.

9.3.3 Funktionale Transformation

Darunter versteht man die komplette Reimplementierung des Systems von Grund auf. Vorausgesetzt man ist erfolgreich, erhält man am Ende eine saubere und moderne Applikation, jedoch ist dies auch mit einem Maximum an Aufwand und Eingriff in das ursprüngliche System verbunden. Man benötigt dafür ungefähr die zehnfache Zeit wie für die Modernisierung der Schnittstellen.

9.4 Anwendung auf das CSPmed System

Eine große Erleichterung bei der Anpassung des CSPmed Systems an zeitgemäße Technologien ist die Tatsache dass die einzelnen Komponenten über relativ gut dokumentierte Schnittstellen miteinander kommunizieren. Die Programmlogik der Serverapplikationen verwendet zur Darstellung der Benutzerschnittstelle Aufrufe des WMS sodass man an dieser Stelle wie in 9.3.1 skizziert vorgehen kann. Dabei müssen existierende Programmmodule nicht verändert werden, was die Fehleranfälligkeit reduziert.
Bei der Serverapplikation verhält sich die Sache anders, hier hat man es mit einer Applikation zu tun, die speziell für ein Betriebssystem und eine Biblothek zu Darstellung der Benutzerschnittstelle ausgelegt ist. Hierbei kann man zum Beispiel eine formale Transformation wie in 9.3.2 erwähnt durchführen, was bei der konkreten Implementierung unter Verwendung des wxWidget Toolkits und GNU/Linux als Betriebssystem auch geschehen ist.

10 Weitere Entwicklung und Alternativen

10.1 Änderungen an der momentanen Implementierung

- Die Anzahl der Aufrufe von `wxGetApp()`, die eine Referenz auf das Applikationsobjekt der Klasse wxApp zurückgeben, sollte zwecks Verbesserung der Struktur der Applikation verrringert werden.
- Methodenaufrufe der `UICalls` Klasse sollten direkt nach `UIMain::ProcessCall()` verschoben werden um lange Verkettungen von Aufrufen zu vermeiden.
- `UICtlEntry` wird eigentlich nur dazu verwendet, um zu einem Control auch dessen Typ (Checkbox, Textfeld usw.) zu verwalten. Dies ist möglicherweise ist durch Polymorphismus besser lösbar.
- Für `Set-` und `GetName()` Methoden von Controls auf die entsprechenden Methoden von `wxWindow` zurückgreifen.

- Die Konstruktoren, besonders jene der Widgets, sollten überarbeitet werden um einen einheitlichen Zugriff zu gewährleisten und sicherzustellen, dass alle Membervariablen initialisiert und auch benötigt werden.
- Nicht verwendete Membervariablen müssen eliminiert werden und im Konstruktor initialisiert werden.
- MFC-spezifische Offsets bei den Positionierungsmethoden können entfernt werden da sie bei Überarbeitung der Darstellungsroutinen nicht mehr benötigt werden.
- Textdarstellung und Controlgrößen müssen richtiggestellt werden.
- Restliche Funktionalität implementieren (siehe 11.2)
- Quellcode übersichtlicher gestalten

10.2 Erweiterungsmöglichkeiten

- Umstellung auf wxWidgets 2.6.1
- Verwendung des wxDatePickerCtrl Control bei Datumsfeldern
- Inaktive Controls standardmäßig grau (beziehungsweise laut Systemeinstellung deaktiviert) darstellen
- Theme-Unterstützung (GTK2) implementieren
- Buttons mit Anzeigemöglichkeit von Bitmaps und/oder Text implementieren. Momentan werden nur Bitmaps als Beschriftung zugelassen.

10.3 Möglichkeiten zur Implementierung einer zeitgemäßen Oberfläche für CSPmed

Viele der in 10.2 genannten Erweiterungsmöglichkeiten betreffen Funktionalität, die dazu dient, dem Benutzer die Arbeit mit dem CSPmed System so einfach und angenehm wie möglich zu machen. Dazu ist es notwendig, die Applikation in Aussehen und Verhalten an jene Programme anzupassen, die der Anwender oft verwendet und als Standardapplikationen angesehen werden. Um mögliche Verbesserungen der Programmoberfläche zu ermitteln, sollte erst einmal festgestellt werden, in welchen konkreten Punkten das CSPmed System Probleme bei der Bedienung bereitet und wie diese Probleme prinzipiell und technisch konkret gelöst werden können.

10.3.1 Prinzipielle Nachteile der CSPmed Oberfläche

Unzureichende Rückmeldung: Oft ist nicht ersichtlich, welche Felder einer Eingabemaske ausgefüllt werden können beziehungsweise ausgefüllt werden müssen. Fenster die den Eingabefokus haben, sollten visuell hervorgehoben werden. Verschiedene Aufgaben sollte man in eigenen Fenstern erledigen, Funktionen die immer verfügbar sein sollen könnten in eigene Werkzeugleiste auslagern.

Ungewohnte Anordnung der Elemente: Es ist zwar eine Leiste mit Buttons vorhanden, diese wird allerdings immer am rechten Bildschirmrand angezeigt, statt in einer eignen verschiebbaren Werkzeugleiste unter der Menüleiste. Weiters existiert eine Zeile zur Ausgabe von Statusmeldungen, jedoch befindet sich diese über den Funktionstasten anstatt am untersten Rand des Fensters, wie dies bei den meisten Applikationen der Fall ist.

Informationsstrukturierung: Eingabemöglichkeiten sollten nur dann angezeigt werden wenn sie auch tatsächlich benötigt werden und Sinn machen.

Integration in die verwendete Oberfläche: Die Standarddialoge der jeweils verwendeten Oberfläche zum Beispiel bei der Farb- oder Dateiauswahl sollten verwendet werden.

Flexible Anpassung: Die Elemente sollten auf Größenänderung des Fensters in dem sie sich befinden reagieren und sich ebenfalls entsprechend verschieben. Die Oberfläche sollte sich durch installierbare Themes an die Bedürfnisse des Benutzers anpassen.

10.3.2 Lösungsansätze

Viele der in 10.3.1 angeführten Unzulänglichkeiten lassen sich durch Überarbeitung des Codes der Serverapplikationen oder durch moderate Änderungen am CSP medclient integrieren. Bei manchen Anforderungen an eine moderne Applikation stößt man jedoch an die Grenzen des bisherigen CSPmedclients. Vor allem die in 10.3.1 und 10.3.1 genannten Verbesserungen sind schwierig mit dem bisherigen Konzept vereinbar, wie im Folgenden erläutert werden soll.

Realisierung eines plattformunabhängigen Maskenlayouts

Ein Problem bei der Erstellung von Dialogen, die auf verschiedenen Plattformen angezeigt werden sollen ist, dass die Standardgrößen der Controls je nach Plattform unterschiedlich sein können. Eventuell sind auch noch Themes oder Windowmanager installiert, die ihrerseits Dekorationen zu den Controls oder Fenstern hinzufügen und dadurch die Größe ebenfalls beeinflussen. Dies alles wurde bei der Konzeption des WMS mit seinem koordinatenbasierten Layout der Texteingabemög-

lichkeiten natürlich noch nicht berücksichtigt. Eine pixelgenaue Ausrichtung der Controls auf einem Formular sieht also beispielsweise unter Microsoft Windows ansprechend aus, kann aber unter GNU/Linux bei der Darstellung mit Hilfe der GTK+ Bibliothek problematisch sein, da es zu unschönen Überschneidungen von Elementen kommen kann. Unterschiedliche Schriftarten und Schriftgrößen tun ein Übriges um die Problematik von verschobenen und sich überschneidenden Controls zu verschärfen, sodass klar ist, dass man sich von der Idee des koordinatenbasierten Formularlayouts verabschieden muss. Falls man doch daran festhalten möchte, müsste man für jede Plattform separat Korrekturwerte zu den Positionen der einzelnen Controls addieren, um ein ansprechendes Layout wiederherzustellen, was dem Konzept der einheitlichen Codebasis für alle Plattformen zuwiderläuft.
Von Seite der Firma CSP werden Applikationen die im Terminal verwendbar sein sollen nicht mehr neu entwickelt. Für die Darstellung soll ab sofort ausschließlich der CSPmedclient zuständig sein, jedoch sollen bestehende Terminalapplikationen weiter gepflegt werden können. Dies bedeutet, dass bestehende Formulare daher so gut wie möglich mit Hilfe der koordinatenbasierten Beschreibungsdateien dargestellt werden müssen, man das Layout also entweder an eine Plattform anpassen muss, oder, wie erwähnt, manuell je nach Plattform Korrekturwerte für die Koordinaten ermitteln muss.
Für die Erstellung plattformunabhängiger Dialoge stehen im wxWidgets Toolkit sogenannte Sizer zur Verfügung, deren Konzept weiter unten erklart wird. Um diese zu verwenden, benötigt man allerdings eine um einiges kompliziertere Beschreibungsdatei, die man üblicherweise automatisiert erstellt, indem man mit Hilfe von frei erhältlichen Programmen die Benutzerschnittstelle grafisch konstruiert.
Es stellt sich nun die Frage, ob und wie man das Sizerkonzept am Besten in das WMS integriert. Fest steht, dass die bestehende Codebasis weiterverwendet werden soll, man also die bisher existierenden WMS Methodenaufrufe weiterhin unterstützen muss. Die übliche Methode, eine Benutzerschnittstelle mit Hilfe des wxWidgets Toolkit zu entwerfen, ist mit einem Programm wie wxGlade die entsprechenden Dialoge visuell zusammenzustellen und danach eine XML Resourcendatei erzeugen zu lassen. Anhand dieser Resourcendatei werden dann von wxWidgets alle im Dialog enthaltenen Elemente erzeugt. Dazu parst eine wxWidgets Methode die XML-Resourcendatei und ruft für jedes enthaltene Control die `Create()` Methode auf sodass es dargestellt werden kann. Nun ist es möglich, Eventtables auf diese Controls zu definieren. Das Parsen der Resourcendatei, das hier das wxWidgets Toolkit übernimmt, macht in der aktuellen Version der CSPmedclient selbst, indem er die Beschreibungsdateien für Fenster und Masken liest und die entsprechenden Controls erzeugt. Um gleichzeitig die alten und die neuen XML Beschreibungsdateien verwenden zu können, könnte man den Code, den wxWidgets zum Parsen und Erstellen der Controls verwendet, in den CSPmedclient einbauen (beziehungsweise von dort aufrufen) mit dem Unterschied, dass keine wxWidgets Standardcontrols erzeugt werden, sondern auf die `Create()` Methoden der jeweiligen Controlklassen des CSPmedclients zurückgegriffen wird. Dies ist notwendig, um die so erzeugten Controls auch in den verschiedenen Datenstrukturen

des CSPmedclients einzufügen, auf die weitere WMS Methodenaufrufe zugreifen. Natürlich müsste für die Unterstützung des neuen Resourcedateiformats auch der Formularcompiler angepasst werden, der die Pufferstruktur für das Formular generiert.
Die Erstellung neuer Formulare würde sich mit einer Integration der sizerbasierten XML-Resourcendateien drastisch verkürzen, da nun mit bereits verfügbaren grafischen Tools gearbeitet werden könnte. Es könnte auch ein Großteil des bestehenden CSPmedclient Codes weiterverwendet werden, jedoch ist auch ein erheblicher Anteil an neuem Code zu implementieren. Der Ansatz erscheint zugegebenermaßen wenig elegant, da Methoden abgeändert werden müssen, die bereits im wxWidgets Toolkit enthalten sind, was eigentlich nur deshalb notwendig ist weil man von der "alten" WMS-API auf die modernere wxWidgets API zugreifen muss. Die obenstehenden Überlegungen stellen im Übrigen nur einen möglichen Lösungsansatz dar, es gibt keinerlei Sicherheit, dass eine Einbindung der sizerbasierten Positionierung tatsächlich so funktionieren kann, geschweige denn bereits konkreten Quellcode. Sie sollen lediglich als Anregung verstanden werden.
Ein weitaus radikalerer Ansatz wäre, die Notwendigkeit des CSPmedclients an sich in Frage zu stellen. Er ist ja im Prinzip nur eine Art Terminalemulation, die Ausgaben im Terminal auf eine grafische Oberfläche umwandelt. Man sollte also am Unterbau ansetzen und nicht allerlei Verrenkungen anstellen, um, bildlich gesprochen, ein System das bereits „auf wackligen Beinen" steht noch zusätzlich mit neuen Funktionen und Technolgien zu belasten. Um Programme, die auf einem Server laufen, auf einem entfernten Rechner darzustellen, existieren mittlerweile eine Reihe von Technologien, wie zum Beispiel die Ausgabe auf X-Servern über das Netzwerk oder VNC. Da, wie bereits erwähnt, die Ausgabe im Terminal ohnehin von der Firma CSP nicht mehr unterstützt wird, ist es eigentlich nicht zweckmäßig, Technologie die für die Ausgabe auf Terminals ausgelegt ist, weiter zu verwenden. Man könnte das gesamte CSPmed System also dahingehend verändern, dass direkt aus dem Serverprogramm auf grafische Bibliotheken zugegeriffen wird und ausschließlich diese zur Ausgabe verwendet werden. Falls man nun zum Beispiel von einem Windows Rechner aus die CSPmed Applikation ausführen möchte, startet man sie einfach wie gewohnt am Server und leitet die Ausgabe auf einen lokal laufenden X-Server um.
Performancemäßig würde der Windows Rechner durch den X-Server nicht um vieles mehr beansprucht werden, und auch auf dem CSPmed Server würde die Verwendung von grafischen Bibliotheken wohl keineswegs starke Einbrüche verursachen. Der große Vorteil, den ich in einer solchen Lösung sehe ist der, zwei proprietäre CSP-eigene Technologien, nämlich den CSPmedclient und die libcsprpc mit erprobten weit verbreiteten Technologien zu ersetzen. Dadurch würden auch Timingschwierigekeiten die bedingt durch die Verwendung von RPC vereinzelt auftreten können, eliminiert werden. Wichtig ist natürlich, dass bei einer Änderung der internen Architektur des CSPmed Pakets die tausenden Zeilen Quellcode, die das CSPmed Paket bereits umfasst, unverändert lauffähig bleiben. Ein möglicher Punkt, bei dem man hierbei ansetzen könnte wäre das WMS, das man mit dem

CSPmedclient zusammenführen könnte. Alle Aufrufe des WMS wie zum Beispiel `Ldform()` oder `Rdform()` könnten direkt am Server in entsprechende Aufrufe der wxWidgets Bibliothek umgesetzt werden, genauso wie es der CSPmedclient bisher tut, nur wie gesagt serverseitig und nicht am Client. Der Vorteil wäre, dass die libcsprpc nicht benötigt würde und man somit diesen alten, unverständlichen Code zu Grabe tragen könnte. Die Darstellung könnte auch auf andere X-Server auf den Client PCs im Netz umgeleitet oder via VNC dargestellt werden. Der Netzwerktraffic des RPC Protokolls wäre sicherlich um ein Vielfaches geringer als der des X-Protokolls oder von VNC, im lokalen Netz würde dies jedoch kaum ins Gewicht fallen.
Die Gründe für solch tiefgreifende Veränderungen sind vor allem, Möglichkeiten zur zeitgemäßen Gestaltung der Benutzerschnittstelle bei gleichzeitiger Kompatibiliät zum bestehenden Code zur Verfügung zu stellen. Bei der gegenwärtigen Struktur der Applikation und auch bei der oben vorgeschlagenen Zusammenführung des CSPmedclients könnte man die Aufrufe der WMS Methoden als Wrapper für das wxWidgets Toolkit sehen, das seinerseits wiederum ein Wrapper für die darunterliegenden Betriebssystem APIs wie GTK+, Win32 usw. ist. Zusätzlich zur Integration des CSPmedclients in das WMS könnte man alle CSPmed Applikationen gegen die GTK+ Bibliotheken linken, was es ermöglichen würde, parallel zu den bisherigen WMS aufrufen GTK+ Code einzufügen. Dieser Code könnte dann alle Vorteile von GTK+ nutzen, wie zum Beispiel Resourcedateien, die visuell erstellt werden können, standardisierte Dialoge und vieles mehr. Die entstehende Applikation würde dann vollständig auf das GTK+ Toolkit setzen und damit, falls alle anderen gelinkten Bibliotheken dies auch unterstützen, unter allen von GTK+ unterstützten Plattformen lauffähig sein. Ein Beispiel der Implementierung ist in 10.3.2 skizziert.

Konzepte zum dynamischen Anordnen von Interaktionselementen

Die sogenannten *Sizer* werden innerhalb von wxWidgets durch die Klasse `wxSizer` repräsentiert und stellen die empfohlene Methode zur Definition des Layouts von Elementen in Dialogen dar, da sie plattformunabhängig visuell ansprechende Dialoge ermöglichen und Unterschiede in Größe und Stil der einzelnen Elemente berücksichtigen. Viele Editoren zur Gestaltung von Dialogen setzen standardmäßig das Sizerkonzept ein und zwingen damit praktisch den Benutzer, ein plattformunabhängiges Layout zu erstellen.
Der Layoutalgorithmus, der von den Sizern in wxWidgets verwendet wird, ist sehr eng mit entsprechenden Konzepten in Javas AWT, dem GTK oder dem Qt Toolkit verwandt. Das Ganze basiert auf der Idee, dass unabhängige Unterfenster ihre minimal benötigte Größe zurückgeben, sowie die Fähigkeit, ihre Größe gemäß der Größe des Fensters in dem sie enthalten sind, zu verändern. Das bedeutet, dass der Programmierer die Größe eines Dialogs nicht manuell festlegt, sondern dass der Dialog einen Sizer zugewiesen bekommt, welcher dann seine beanspruchte Größe, und damit die Größe des Dialogs, zurückliefert. Der Sizer selbst fragt dazu seine

enthaltenen Objekte ab, welche Fenster, leerer Platz oder andere Sizer sein können, sodass eine Hierarchie von Sizern konstruiert werden kann. Der Grund warum Sizer für die Erstellung plattformunabhängiger Dialoge so gut geeignet sind ist der, dass jedes Element seine eigene minimale Größe bekanntgibt und der Algorithmus zur Positionierung Unterschiede in Schriftgröße oder Elementgröße auf den einzelnen Plattformen berücksichtigen kann. Zum Beispiel wird der Dialog automatisch größer dargestellt falls die Schrift und die einzelnen Elemente unter GNU/Linux (GTK) größer sein sollten als unter Microsoft Windows.
Im wxWidgets Toolkit existieren derzeit fünf verschiedene Arten von Sizern, wovon jede eine Möglichkeit repräsentiert, Elemente eines Dialogs anzuordnen. Manche Sizer werden auch nur dazu verwendet, zum Beispiel einen Rahmen um ein Dialogelement oder einen anderen Sizer zu zeichnen. Jeder Sizer ist eine Art Container, da er ein oder mehrere Dialogelemente aufnehmen kann. Diese Dialogelemente werden manchmal auch als Kinder des Sizers bezeichnet. Die Kinder haben, unabhängig davon wie sie umgebende Sizer darstellt, einige Eigenschaften, die zum Teil explizit gesetzt werden können. Darunter sind:

- Minimale Größe: Diese Größe wird entweder explizit im Konstruktor angegeben, manche Elemente können sie jedoch auch automatisch ermitteln.
- Rand: Für die Elemente kann man explizit einen Rand mit der Breite in Bildpunkten festlegen.
- Ausrichtung: Ist mehr Platz für die Darstellung eines Elementes vorhanden als dessen minimale Größe, kann es entweder am linken oder rechten Rand dieses freien Raumes angeordnet werden oder in der Mitte.
- Dehnungsfaktor: Wenn ein Sizer mehrere Kinder enthält und mehr Platz zur Verfügung steht, als die Kinder mindestens brauchen, kann man festlegen, wie der Platz prozentuell unter den Kindern aufgeteilt werden soll.

Folgende Sizerklassen stehen in wxWidgets zur Verfügung:

- `wxBoxSizer`: Dieser Sizer wird dazu verwendet, um Elemente nebeneinander oder untereinander anzuordnen.
- `wxStaticBoxSizer`: Bietet die gleiche Funktionalität wie `wxBoxSizer`, es wird jedoch ein Rahmen mit Text um die Elemente gezeichnet.
- `wxGridSizer`: Die enthaltenen Elemente werden in einem zweidimensionalen Gitter angeordnet, die Größe der einzelnen Gitterabschnitte entspricht der minimalen Ausdehnung des größten Elements im Sizer. Es ist entweder die Anzahl der Spalten oder die Anzahl der Zeilen fixiert, wenn ein Element zum Sizer hinzgefügt wird, wächst dieser in die entsprechend andere Richtung.

- `wxFlexGridSizer`: Dieser Sizer ist von `wxGridSizer` abgeleitet. Die Breite jeder Spalte und die Höhe jeder Zeile wird individuell nach den Ausmaßen des jeweils größten Kindes ermittelt. Zusätzlich können sich Zeilen und Spalten ausdehnen, falls dem Sizer eine andere Größe zugewiesen wird als die, die er ursprünglich angefordert hat.

In Java existiert eine ähnliche Technik um plattformunabhängige Layouts zu ermöglichen. Sie basiert darauf, dass man jedem Container (ein Bereich auf dem die Dialogelemente platziert sind) einen Layoutmanager zuordnet. Dieser Layoutmanager bestimmt wie die Elemente dargestellt werden sollen, also ihre Größe, Position und Ränder. Man kann in Java AWT aus fünf Layoutmanagern wählen, die beliebig kombiniert und deshalb auch zur Gestaltung komplexer Dialoge verwendet werden können:

- `FlowLayout`: Dieser Layoutmanager verändert die Größe der enthaltenen Elemente nie und ordnet diese horizontal nebeneinander an. Falls die Breite nicht ausreicht, setzt sich die Anordnung in der nächsten Zeile fort, falls keine solche existiert, weil der Container zu klein ist, werden die Elemente einfach nicht angezeigt.

- `BorderLayout`: Der Container, der diesem Layoutmanager zugeordnet ist, wird in fünf Sektionen unterteilt: *North*, *South*, *West*, *East* jeweils an den Rändern und ein *Center*, logischerweise in der Mitte des Dialogs. Dabei wird die Breite der Elemente in Norh und South sowie die Höhe der Elemente in West und East an die Größe des zugeordneten Containers angepasst. Die Elemente im Center teilen sich den Platz der übrig bleibt.

- `GridLayout`: Stellt die zugeordneten Elemente in Form eines Gitters dar mit der jeweils gleichen Größe. Es ist zwar möglich die Anzahl der Zeilen und Spalten anzugeben, es wird aber nur einer der Parameter wirklich berücksichtigt. Die Größe der Zellen wird anhand der maximalen Größen der einzelnen Elemente ermittelt.

- `CardLayout`: Dieser Layoutmanager kann nur ein Element auf einmal darstellen. Falls man mehrere Elemente zu einem Container hinzufügt, dem dieser Layoutmanager zugeordnet ist, kann man mittels `next()` und `previous()` zwischen diesen umschalten. Die Größe eines CardLayouts ermittelt sich aus den maximalen Größen der enthaltenen Elemente.

- `GridBagLayout`: Mit dem Layoutmanager `GridBagLayout` lassen sich grafische Komponenten tabellarisch darstellen. Der Unterschied zum `GridLayout` ist, dass die Komponenten über mehrere Spalten beziehungsweise über mehrere Zeilen gehen können. Er ist sehr komplex zu handhaben, deshalb sollte ein grafischer Editor verwendet werden um Dialoge mit Hilfe dieses Layoutmanagers zu erstellen.

Jeder dieser Layoutmanager ist noch umfangreich in seinem Verhalten konfigurierbar, es wurden nur deren prinzipielle Möglichkeiten und Verwendungszweck vorgestellt, weitere Details kann man der ausgezeichneten Dokumentation der entsprechenden Klassen entnehmen.
All diese Technologien und Eigenschaften machen es möglich, Dialoge zu entwerfen, die flexibel auf Größenänderungen der Elemente reagieren können, ohne dass man zusätzlichen Programmieraufwand betreiben muss.

Evaluierung der Lösungsansätze

Um die prinzipielle Funktionsfähigkeit dieser Überlegungen zu überprüfen, habe ich die Makefiles des CSPmed Systems dahingehend verändert, dass sämtliche für GTK+ benötigten Verzeichnisse inkludiert und die GTK+ Bibliotheken zum Programm `At_158` gelinkt werden. Weiters habe ich eine Funktionstaste hinzugefügt die, falls gedrückt, eine Callbackfunktion anspringt, in der GTK+ Code ausgeführt wird. Aktuell wird nur ein einfaches Fenster mit einem Button erzeugt und angezeigt, man sollte dies allerdings auf beliebige Dialoge erweitern können. Die entsprechende Callbackfunktion `fk_gtk` im Programm `At_158` sieht wie in Algorithmus 14 angegeben aus.
Startet man das derart modifizierte Programm im Terminal, wird das Fenster wie gewünscht angezeigt wenn der Programmfluss die Methode `gtk_main()` erreicht. Diese Methode startet eine GTK-internen Eventloop, der dafür Verantworlich ist, dass die zu den entsprechenden Signalen definierten Callbackfunktionen (in diesem Fall `delete_event()` und `destroy()`) ausgeführt werden. Falls das angezeigte Fenster geschlossen wird, wird auch der Eventloop durch den Methodenaufruf `gtk_main_quit()` beendet und die Applikation läuft wie gewohnt weiter.
Wenn man das obenstehende Programm jedoch mittels CSPmedclient startet, bewirkt das Ausführen der Callbackfunktion und somit des GTK+ Codes nur ein Einfrieren der Benutzerschnittstelle, das mittels des GTK+ Codes definierte Fenster wird nicht angezeigt. Der Grund für dieses Verhalten liegt darin, dass der CSPmedclient sein eigenes `gtk_main()` bei jedem `Rdform()` ausführt. Wird dann eine Funktionstaste gedrückt, wird `gtk_main()` beendet und die entsprechende für diese Funktionstaste definierte Callbackfunktion aufgerufen. Dadruch dass die `gtk_main()` Methode ja eine Schleife ist, die solange läuft, bis sie mittels `gtk_main_quit()` abgebrochen wird, erhält der CSPmedclient jetzt auch keine Calls vom Server mehr, wartet jedoch seinerseits in einer Schleife darauf, die die grafische Benutzeroberfläche nicht updatet. Diese Updaten könnte man natürlich ziemliche einfach implementieren, es stellt sich jedoch die Frage, wie sinnvoll das ist, da das obenstehende Beispiel ist nur als Demonstration der prinzipiellen Möglichkeit eines schrittweisen Umstiegs auf GTK+ gedacht ist. Falls man dies vor hat, muss man auf jeden Fall zuerst, wie bereits beschrieben, den CSPmedclient in das WMS integrieren und die libcsprpc Biblitheken entfernen, erst wenn die Serverlogik mit der Benutzerschnittstelle in einer Applikation vereint ist, kann

Algorithm 14 Code zur Einbindung von GTK+ Code in eine Serverapplikation

```
gint delete_event(GtkWidget *widget, GdkEvent *event,
gpointer data)
{
    return(FALSE);
}
void destroy(GtkWidget *widget, gpointer data)
{
    gtk_main_quit();
}
static fk_gtk(text,len,scan,akt_var)
char *text; int len, scan,akt_var;
{
    // Wird durch Betätigung der Funktionstaste
angesprungen
    GtkWidget *window;
    GtkWidget *button;
    window = gtk_window_new (GTK_WINDOW_DIALOG);
    gtk_signal_connect(GTK_OBJECT(window),"delete_event",
        GTK_SIGNAL_FUNC(delete_event), NULL);
    gtk_signal_connect(GTK_OBJECT(window),"destroy",
        GTK_SIGNAL_FUNC(destroy), NULL);
    gtk_container_set_border_width(GTK_CONTAINER(window),
10);
    button = gtk_button_new_with_label("Hello World");
    gtk_container_add(GTK_CONTAINER(window), button);
    gtk_widget_show(button);
    gtk_widget_show(window);
    gtk_main();
    return (WMS_AGAIN);
}
```

man beginnen, GTK+ Code einzufügen und zu testen. Dann sollten Probleme der oben geschilderten Art, wie sie jetzt bei der Verwendung mit dem CSPmedclient auftreten, nicht mehr vorkommen und das Programmverhalten jenem gleichen, wie es heute schon im Terminal ist.
Die Verwendung von GTK+ ist in diesem Fall naheliegend, da es sich hierbei um ein Toolkit handelt, das standardmäßig in C programmiert wird. Wie in 14 bereits teilweise ersichtlich ist, ist die Übersichtlichkeit sehr schlecht, wenn man WMS-Code und GTK+ Code mischt, da man dann sowohl Callbackfunktionen des WMS und solche von GTK+ unterscheiden muss. Weiters sollte man, wenn man schon dabei ist auf eine andere Grafikbibliothek umzusteigen, auch prüfen, inwiefern es möglich ist, für Neuentwicklungen C++ einzusetzen. Da die direkte Programmierung von GTK+ unter C doch etwas komplizert erscheint, könnte man wenn man C++ benutzt, einen entsprechenden GTK-Wrapper, wie zum Beispiel wxGTK, einsetzen, was wiederum Plattformunabhängigheit bringen würde. Weiters wäre natürich von Vorteil, dass zum Beispiel wxGTK besser dokumentiert sind ist als GTK selbst und man dann wxGTK sowieso schon zum CSPmed System gelinkt hätte, da es ja für die Emulation der alten WMS Calls verwendet wird.
Momentan sehe ich den Einsatz von GTK+ Code beziehungsweise wxWidgets/wxGTK Code zum Beispiel in solchen Fällen als sinnvoll an, wenn umfangreiche Formulare für den Ausdruck von Belegen implementiert werden sollen. Bisher mussten dazu (siehe 3.1.4) händisch eigene Fenster- und Formularbeschreibungsdateien geschrieben werden, diese wurden kompiliert, anschließend getestet, neu editiert und so weiter, bis alles schließlich an der richtigen Stelle am Formular sitzt. Mit Hilfe von GTK+ könnte man durch Verwendung von grafischen Tools diese zeitaufwändige lästige Arbeit vereinfachen und dem Benutzer gleichzeitig eine angenehme, zeitgemäße Benutzerschnittstelle bieten. Abgesehen davon könnte eine Implementierung von neuem Code in C++ die Übersichtlichkeit sehr steigern und es könnten fortschrittliche Testmethoden (siehe 7) eingesetzt werden, die die Qualität der Software bedeutend steigern.

11 Zusammenfassung

Ziel des neuen CSPmedclients ist die Implementierung der Funktionalität und des Aussehens des bisherigen Clients, der mit Hilfe Microsofts MFC implementiert wurde.

11.1 Bereits implementierte Funktionalität

Netzwerkfunktionalität: Die Verbindung zum Server und die Kommunikation (GNU/Linux und SCO) funktioniert einwandfrei. Sowohl die Kommunikation vom Server zum Client als auch umgekehrt (reverse callback) wird unterstützt.

Verwendbarkeit auf unterschiedlichen Plattformen: Für die Betriebssysteme GNU/Linux und Microsoft Windows existiert eine gemeinsame Codebasis, von der für weitere Verbesserungen oder Implementierungen ausgegangen werden kann.

Praktische Verwendbarkeit: Zur Darstellung der Benutzerschnittstelle wird die wxWidgets Bibliothek verwendet, die plattformunabhängig eingesetzt werden kann und sich für diesen Zweck recht gut eignet. Testweise wurde versucht, ein Programm (`At_158`) in Funktion und Aussehen so genau wie möglich dem MFC-basierten CSPmedclient unter Microsoft Windows anzupassen, was gelungen ist. Für die anderen Applikationen ist zum Einen die Implementierung weiterer RPC-Calls vonnöten, zum Anderen gibt es unvorhersehbare Unterschiede im Aussehen und Verhalten, die gesondert getestet und implementiert werden müssen (siehe dazu 10).

11.2 Noch nicht implementierte Funktionalität

Vollständige Bedienung mittels Tastatur: Die Navigation zwischen Formularelementen (z. B. Textfeldern) funktioniert bereits, ebenso die Funktionstasten für den Zugriff auf die definierbaren Buttons am unteren Bildschirmrand. Abkürzungstasten für Menüeinträge sind noch nicht implementiert, ebensowenig wie die Aktivierung von Buttons mittels Abkürzungstasten.

Formularelemente: Bis auf die Listview wurden alle Elemente implementiert. Die Combobox wurde nur teilweise implementiert, einige Funkionen zum Updaten ihres Textfeldes und zur Cursorsteuerung fehlen.

Darstellung unter den verschiedenen Plattformen: Da die Schriftgrößen und Schriftarten auf den einzelnen Plattfomen (getestet wurden MS Windows und GNU /Linux) unterschiedlich sind, kommt es deshalb zu Unterschieden in der Darstellung. Abhilfe schafft eventuell das Verwenden der standardmäßig definierten Schriftgrößen und -stile. Auf den verschiedenen Plattformen ist es jeweils möglich, Standardschriftgrößen- und stile zu definieren, die die einzelnen Applikationen dann verwenden. Auch die wxWidgets Bibliothek unterstützt den Zugriff auf diese Einstellungen und kann sie mit wenig Implementierungsaufwand auch verwenden. Ähnliches gilt für die Farbdefinitionen (zum Beispiel den Fensterhintergrund). Felder ohne 3D-Rahmen kann man vom CSPmedclient aus unter GNU/Linux nicht darstellen, da die GTK+ Widgets, auf die die wxWidgets Klassen unter GNU/Linux zurückgreifen, das Deaktivieren des Rahmens nicht unterstützen. Unter Microsoft Windows funktioniert dies jedoch wie gewünscht. Abhilfe für dieses Verhalten schafft eventuell ein eigenes GTK-Theme, oder, was mehr Implementierungsaufwand erfordert, die Darstellung als `wxStaticText` auf standardmäßig weißem Hintergrund statt als `wxTextCtrl`. Letztere Variante ist aber nur dann möglich, falls das Feld zusätzlich für Benutzereingaben nicht zur Verfügung stehen soll, also nur zur Ausgabe dient.

Auflösung: Da der Fokus der Implementierung zuerst auf der Erstellung einer lauffähigen Applikation unter GNU/Linux und Microsoft Windows gelegen ist, wurde nur in der Standardauflösung von 1280x1024 getestet. Da eine Positionierung der Formularelemente pixelgenau erfolgen muss und auf den verschiedenen Plattformen unterschiedliche Schriftarten und -stile (siehe oben) eingerichtet sein können, kommt es oft zu Überschneidungen von Feldern. Abhilfe schafft, wie oben bereits erwähnt, die Definition und Verwendung geeigneter Standardschriften. Auf jeden Fall sollte man auch die Positionierungsmethode der Formularelemente überprüfen und testen inwiefern man vom statischen pixelgenauen Positionieren der Elemenete abweichen und das sizerbasierte Konzept der wxWidgets verwenden kann. Dies hätte den Vorteil, auf Größenänderung der Fenster entsprechend reagieren und die Darstellung auf den verschiedenen Plattformen einheitlich gestalten zu können. Mehr zu dieser Problematik siehe 10.

Elementgrößen: Eng verbunden mit dem oben erwähnten Problem der Auflösung ist die Darstellung von Listboxen und Comboboxen in der jeweils richtigen Größe. Der bisherige CSPmedclient verwendet zum Beispiel Methoden der MFC zur Ermittlung der Schriftgrößen in Pixel um die Größe von List- und Comboboxen entsprechend einzustellen. Diese Methoden stehen allerdings nicht plattformübergreifend zur Verfügung, sodass man diese entweder manuell nachprogrammiert, was wiederum Ungenauigkeiten bei den unterschiedlichen Plattformen und somit manuelle Anpassungen mit sich bringt, da man auch so Dinge wie den Abstand zwischen zwei Textzeilen einkalkulieren muss, um etwa eine Listbox genau so groß wie unbedingt nötig darzustellen. Die andere, bessere Möglichkeit wäre, dem wxWidget Toolkit die Größenberechnung zu überlassen, was zum Teil umfangreiche Änderungen in den Klassenmethoden der betroffenen Formularelemente und dem Algorithmus der Erzeugung dieser Elemente nach sich ziehen würde, ich aber auf jeden Fall für notwendig erachte. Betroffen wären die Formularelemente Listbox, Combobox und eventuell auch die Listview. Letztere wurde noch nicht implementiert.

Bitmapbuttons: Einige Buttons mancher Applikationen enthalten keinen Text sondern Bitmaps. Diese wurden aus dem bisherigen Microsoft-Windows Client entnommen und besitzen eine statische Hintergrundfarbe, die unter Umständen nicht zum aktuell eingestellten Hintergrund der verwendeten Plattform passt und dadurch ein Rand erkennbar ist. Abhilfe würde nach einer ersten Einschätzung ein manuelles Bearbeiten der einzelnen Bitmaps machen, bei dem man einfach einen durchsichtigen Hintergrund definiert.

„Eye-candy": Ein Splashscreen, wie in der bisherigen Version für Microsoft Windows fehlt, ebenso wie ein Loginscreen, bei dem man Server, Programmname, Port usw. angeben kann. Diese Informationen müssen momentan auf der Kommandozeile spezifiziert und der Applikation als Parameter übergeben werden, was

weniger benutzerfreundlich ist. Weiters wurden Farben für Schrift und Hintergrund bereits implementiert, Unterstützung für blinkenden Text fehlt noch. Eine Implementierung dieser Features ist meiner Meinung nach aber nicht vorrangig und kann auch mit relativ wenig Aufwand bewerkstelligt werden.

Benutzerdefinierte Anpassung der Benutzerschnittstelle: Ein verhältnismäßig neues Feature des MFC-CSPmedclients ist, die Position, Schriftgröße und -stil von Formularelementen einzeln mittels der grafischen Umgebung festzulegen. Diese Funktionalität wurde noch nicht implementiert, zumal auch erst geprüft werden muss, inwiefern man hierzu die Konfigurationsdialoge der jeweiligen Plattform verwenden kann.

11.3 Weitere Vorgehensweise

Die bis dato implementierte Applikation weist von der Codestruktur große Ähnlichkeiten zum bisherigen CSPmedclient auf, wodurch sie verhältnismäßig schnell implementiert werden konnte. Es hat sich allerdings gezeigt, dass das verwendete Design auf den ersten Blick nicht immer optimal ist und man an vielen Stellen Verbesserungen durchführen könnte, um die Verständlichkeit und Wartbarkeit des Codes zu erhöhen. Weiters sind viele Änderungen, die obenstehend vorgeschlagen wurden, oft nur durch umfangreiche Änderungen an der Klassenstruktur möglich, da eine plattformunabhängige Implementierung zum Teil andere Voraussetzungen benötigt, die ursprünglich ja nicht vorhersehbar waren.
Falls man sich dazu entscheidet, den vorliegenden CSPmedclient weiterzuentwickeln, wären zwei Vorgehensweisen denkbar:

1. Man erweitert den vorliegenden Code um die fehlende Funktionalität. Im Prinzip wäre noch die Listview zu implementieren und die angesprochenen Probleme bei der Darstellung auszumerzen (Quick-Fix Methode). Weiters müssen auch noch die restlichen RPC-Calls implementiert werden.

2. Da nun ein lauffähiger Code vorliegt, wäre es am Besten, diesen zwecks Refactoring (siehe 8) durchzusehen. Da man beim Erstellen von neuem Code sowieso testen muss, würde das Refactoring keinen so großen Aufwand bedeuten, und vor allem die Flut von Membervariablen und Zugriffsfunktionen eindämmen, was zu einer stabileren Applikation führen würde. Es wurde versucht, Vereinfachungen im Code bereits bei der Portierung der Applikation einzubringen, jedoch muss erst einmal eine gewisse Codebasis die läuft und testbar ist vorhanden sein, bevor man Methoden des Refactoring anwenden kann. Diese Voraussetzungen wären nun also gegeben, sodass man dieser Variante auf jeden Fall den Vorzug geben sollte.

3. Zu Klären wäre weiters ob der Vorschlag der Integration des CSPmedclients in das WMS nicht der Weg sein soll, den man für die Zukunft des Programm-

paketes einschlagen soll und dies gemeinsam mit dem in Punkt 2 beschriebenen Vorgehen realisiert.

Literatur

[1] *Das WINE Projekt*, http://www.winehq.org, 11.10.2005

[2] *Das GNU (Gnu's Not UNIX) Projekt*, http://www.gnu.org, 11.10.2005

[3] *Firma Trolltech, Entwickelt das Qt Toolkit*, http://www.trolltech.com, 11.10.2005

[4] *Firma CSP, Entwickelt das CSPmed Softwarepaket*, http://www.csp.at, 11.10.2005

[5] *Homepage des wxWidgets Toolkits*, http://www.wxwidgets.org, 11.10.2005

[6] *Liste von Benutzern des wxWidgets Toolkits*, http://www.wxwidgets.org/users.htm, 11.10.2005

[7] *Liste aller von der OSI (Open Source Initiative) anerkannten Open Source Lizenzen*, http://opensource.org/licenses, 11.10.2005

[8] *FLTK Toolkit*, http://www.fltk.org, 11.10.2005

[9] *FOX Toolkit*, http://www.fox-toolkit.org, 11.10.2005

[10] *Applikationen und Projekte die das FOX Toolkit verwenden*, http://www.fox-toolkit.org/projects.html, 11.10.2005

[11] *Firma Sun, Entwicklt die Java Programmiersprache*, http://www.sun.com, 11.10.2005

[12] *Die freie Online Enzyklopedie Wikipedia*, http://www.wikipedia.org, 11.10.2005

[13] *Wikipedia-Eintrag zu Richard Stallman*, http://de.wikipedia.org/wiki/Richard_Stallman, 11.10.2005

[14] *Das CppUnit Unit-Test Werkzeug*, http://sourceforge.net/projects/cppunit, 11.10.2005

[15] *Die Definition Freier Software laut GNU*, http://www.gnu.org/philosophy/free-sw.de.html, 11.10.2005

[16] Kanglin Li, Mengqi Wu: *Effective GUI Testing Automation: Developing an Automated GUI Testing Tool*, Wiley November 2004 ISBN 0-782-14351-2

[17] Coulouris, George, Jean Dollimore und Tim Kindberg: *Distributed Systems Concepts and design*, Addison-Wesley Third Edition 2001

[18] Fowler, Martin: *Refactoring : improving the design of existing code*, Addison-Wesley 2003 ISBN 0-201-48567-2

[19] Brown, William J: *AntiPatterns : refactoring software, architectures and projects in crisis*, Wiley & Sons 1998 ISBN 0-471-19713-0

[20] Kerievsky, Joshua: *Refactoring to Patterns*, Addison-Wesley 2004 ISBN 0-321-21335-1

[21] Wake, William C.: *Refactoring Workbook*, Addison-Wesley 2003 ISBN 0-321-10929-5

[22] Glad, Anthony S.: *Cross-Platform Software Development*, Coriolis Group 1994 ISBN 0-442-01812-6

[23] Ezran, Michael, Maurizio Morisio und Colin Tully: *Practical Software Reuse*, Springer 2002 ISBN 1-85233-502-5

[24] Naugle, Matthew G.: *Network protocols*, McGraw-Hill 1999 ISBN 0-070-46603-3

[25] Boutaba, Raouf.: *Networking 2005*, Springer 2005 ISBN 3-540-25809-4

[26] *GNU Lesser General Public License, Inoffizielle deutsche Übersetzung*, http://www.gnu.de/lgpl-ger.html