Entwicklung eines Verfahrens zur effizienten Migration von Windows-Applikationen auf das Linux Betriebssystem

Diplomarbeit im Fach Informatik

an der
Eberhard-Karls-Universität Tübingen
Wilhelm-Schickard-Institut für Informatik
Lehrstuhl Technische Informatik

Betreut von
Prof. Dr.-Ing. Wilhelm G. Spruth

vorgelegt von David Gümbel

Tübingen, im Dezember 2004

Hiermit erkläre ich, daß ich die vorliegende Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

.....

Ort, Datum, Unterschrift

Zusammenfassung

Für Wirtschaft und Verwaltung ist oftmals eine Migration auf das freie Betriebssystem Linux unter wirtschaftlichen und technischen Gesichtspunkten interessant. Dabei sind Fachanwendungen, die unter Linux weiterbetrieben werden müßten, eine wesentliche Kernproblematik.

In der vorliegenden Arbeit wird eine Untersuchung dieser Problematik anhand der Fachanwendungen der Stadt Böblingen durchgeführt. Es wird ein Verfahren und seine Implementierung vorgestellt, mit dessen Hilfe die Migration von Windows-Applikationen auf Linux effizient durchgeführt werden kann. Dieses erlaubt die automatisierte Ermittelung der Problemstellen, die durch die unvollständige Implementierung der Windows-API durch die Freie Software Wine entstehen. Für die Analysen wird der Quelltext der Applikation nicht benötigt. Da das Verfahren statisch arbeitet, muß die Applikation für die Analyse nicht ausgeführt werden.

Beispielhaft wird in Kapitel 7 eine Anwendung des Verfahrens an einer vergleichsweise komplexen Kommunalsoftware durchgeführt (Seite 85ff). In welcher Weise sich mit dem Verfahren Aufwand und Kosten für Migration oder Portierung reduzieren lassen, wird im Kapitel „Diskussion und Ausblick“ ab Seite 101 erläutert. Es werden dabei auch Erfahrungswerte aus einem Migrationsprojekt präsentiert, bei dem das Verfahren eingesetzt wurde. Weitere Vorgehensmöglichkeiten werden ab Seite 107 erläutert.

Danksagung

An dieser Stelle möchte ich allen danken, die mich während der Entstehung dieser Diplomarbeit auf vielfältige Art und Weise unterstützt haben.

Mein besonderer Dank gilt meinem Betreuer Prof. Spruth, der sich bereit erklärt hat, dieses selbstgewählte Thema zu betreuen, und ohne den die Durchführung dieser Arbeit nicht möglich gewesen wäre. Seine Anregungen und seine konstruktive Kritik waren immer ein ausgezeichneter Boden für Ideen und mir persönlich stets ein Ansporn.

Aus den vielen Personen, mit denen ich über die Thematik gesprochen habe möchte ich Prof. Hauck und Stefan Munz herausheben, die mit mir diskutiert und mir neue Blickwinkel auf die Problematiken eröffnet haben. Meinen Dank möchte ich allen genannten und ungenannten Diskussionspartnern aussprechen. Dankbar bin auch Nina Lehmann für ihr stets wertvolles Feedback und ihre Unterstützung.

Für die Vermittlung wertvoller Kontakte möchte ich Prof. Bode (TU München) an dieser Stelle ebenfalls danken. Herrn Klaus Gödde (Stadt Böblingen) und den Mitarbeiterinnen und Mitarbeitern der IT-Abteilung der Stadt Böblingen gebührt ebenfalls mein Dank.

Für die freie Software, die in dieser Diplomarbeit Verwendung gefunden hat, und ohne welche sie nicht in diesem Umfang möglich gewesen wäre, gebührt den jeweiligen Autoren ebenfalls mein Dank. Dies gilt in ganz besonderer Weise den Entwicklerinnen und Entwicklern des Wine-Projektes.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Linux Das Linux Betriebssystem wurde Anfang der 90er Jahre vom finnischen Studenten Linus Torvald geschaffen[1]. Seitdem kann die Open Source Software sowohl im Server- als auch im Arbeitsplatz-Umfeld überdurchschnittlich hohe Wachstumsraten aufweisen. Mit Linux-Servern wurde beispielsweise im vierten Quartal 2003 weltweit 960 Millionen US-Dollar Umsatz erzielt, das sind 63,1 Prozent mehr als im Vergleichsquartal des Vorjahres[2]. Eine jüngst von den Marktforschern von IDC veröffentlichte Studie prognostiziert, daß der Umsatz mit Linux-basierten Desktops, Servern und Softwarepaketen von momentan geschätzten 15 Milliarden auf über 35 Milliarden US-Dollar im Jahr 2008 steigen wird[3].

Zu diesem Wachstum hat die außergewöhnlich hohe Stabilität und Flexibilität beigetragen. Besonders wirksam lassen sich mit Linux auch die Gesamtkosten (TCO, Total Cost of Ownership) reduzieren[4].

Linux ist anpassungsfähig und ist für den wirtschaftlichen Einsatz in nahezu beliebigen Anwendungsbereichen geeignet. Es läuft in kleinen embedded-Geräten wie PDAs, Routern, Print-Servern und Firewalls ebenso wie auf einer Reihe von Spielkonsolen, auf PCs und auf Großrechnern z.B. von IBM. Mit zunehmendem Reifegrad von Linux-basierter Software und einer stetig wachsenden Zahl von Open Source Programmen haben sich eine ganze Reihe von Kommunen und Unternehmen entschlossen, Linux einzusetzen. Prominente Beispiele sind die Stadt Schwäbisch Hall und das Migrationsprojekt der Stadt München, bei dem etwa 14000 Client-Rechner auf das Linux Betriebssystem umgestellt werden.

---

[1] Zur Geschichte von Linux siehe auch den Wikipedia-Artikel `http://en.wikipedia.org/wiki/Linux`, zuletzt besucht am 20.12.2004.

[2] Quelle: IDC, abrufbar unter `http://www.heise.de/newsticker/meldung/45061`, zuletzt besucht am 7.12.2004.

[3] `http://www.heise.de/newsticker/meldung/54330`, zuletzt besucht am 29.12.2004.

[4] Zur Reduzierung von Total Cost of Ownership mittels Linux siehe auch den Artikel [IT Manager's Journal].

1.1 Motivation

Für Wirtschaft und Verwaltung ist ein Austausch von proprietären Softwarelösungen durch Freie oder Open Source Software (OSS) nicht nur unter wirtschaftlichen Gesichtspunkten interessant. Es existieren neben der offenkundigen Möglichkeit zur Einsparung durch geringere Lizenzkosten auch noch eine ganze Reihe technischer Vorteile. Dazu sind vor allem die höhere Sicherheit durch den offenliegenden Quellcode, offene Formate, und die Flexibilität des Entwicklungsmodells zu zählen[5].

Migrationsprobleme Die Nutzung dieser Vorteile erscheint gerade bei Betriebssystemen attraktiv. Eine Migration z.B. auf das freie Betriebssystem Linux wird aber nachhaltig dadurch erschwert, daß es für den weitaus größten Teil der Benutzersoftware keinen Ersatz gibt, der dort lauffähig wäre. Diese sog. Fachanwendungen stellen ein erhebliches Hemmnis bei einer möglichen Migration auf Linux dar: Eine Portierung ist teuer, schwierig zu kalkulieren, und für viele kleinere Softwarehersteller wirtschaftlich nicht interessant. Zudem erscheint die Pflege von zwei Versionen (für Windows und für Linux) der gleichen Software weder ökonomisch noch technisch sinnvoll.

Zwar besteht die Möglichkeit, Windows-Programme unter Wine – einer freien Win32-API-Implementierung für Linux – weiter zu betreiben, aber es stellen sich hier neue Probleme: Wine bietet nicht den vollen Umfang aller Funktionen der Windows-API, sondern enthält unvollständig oder gar nicht implementierte Teilmengen. Welche davon von einem zu migrierenden Programm genutzt werden, läßt sich höchstens durch Analyse des Quelltextes herausfinden. Ist die Applikation in einer der populären Hochsprachen geschrieben, die Betriebssystemaufrufe vor den Programmierern verstecken (beispielsweise Visual Basic), hilft nur noch das Trial-and-Error-Verfahren, d.h. das Programm unter Wine zu starten und Meldungen in der Debug-Ausgabe von Wine bzw. Programmabstürze zu protokollieren und diesen dann manuell nachzugehen. Diese Vorgehensweise ist weder seriös kalkulierbar, noch führt sie zu einer sicheren Aussage bezüglich der Lauffähigkeit unter Wine.

Es wäre wünschenswert, eine Untersuchung hinsichtlich der Kompatibilität einer Windows-Software mit Linux/Wine möglichst ohne langwierige Tests durchzuführen. Zudem steht der Quelltext der Applikationen häufig nicht zur Verfügung, so daß anzustreben wäre eine solche Überprüfung auf Basis von in binärer Form vorliegender Software zu realisieren. Das in dieser Arbeit entwickelte Verfahren zielt daher darauf ab, die Analyse von

[5] „Linux und andere freie Software bieten die Chance für mehr Vielfalt und damit die Chance zu mehr Sicherheit. Daneben stärkt es die Verhandlungsposition jedes Kunden, wenn er gegenüber einem Anbieter auf eine Alternative verweisen kann. Ganz abgesehen davon, dass bei einer Open Source Software selbst praktisch keine Lizenzkosten anfallen“, so Bundesinnenminister Otto Schily in [Handelsblatt 2002].

Windows-Programmen bezüglich ihrer Ausführbarkeit unter Linux/Wine statisch (d.h. nicht zur Laufzeit), ohne Quelltext des Programmes, und möglichst umfassend durchzuführen. Es ermöglicht ein automatisches Auffinden von Problemstellen und ihre Zuordnung zu den sie verursachenden Software-Modulen, sowie eine Gewichtung nach der Schwere der Fehler.

1.2 Aufbau der Arbeit

Das in dieser Arbeit entwickelte Verfahren ist das Ergebnis eines Evaluierungsprozesses, der zusammen mit der Stadt Böblingen durchgeführt wurde. Dort wurde durch die bevorstehende Ablösung von Windows NT und Microsoft Office durch Nachfolgeprodukte und durch die dadurch verursachten Kosten die Möglichkeit von Linux als alternatives Betriebssystem interessant. Bei der Analyse der bei einer (fiktiven) Linux-Migration potentiell auftretenden Probleme wurde schnell deutlich, daß die Problematik der Fachanwendungen die wohl drängendste sein würde. In Kapitel zwei wird die Vorgehensweise bei der Entwicklung des hier vorgestellten Verfahrens erläutert und die Zielsetzungen definiert, mit denen die Entwicklung des Systems angegangen wurde.

Für eine mögliche Lösung der Fachanwendungsproblematik stehen grundsätzlich mehrere Technologien zu Verfügung. Es erscheint angemessen, deren Leistungsfähigkeit kurz zu beleuchten und so zu erläutern, warum Wine als Basis gewählt wurde. In Kapitel drei findet sich daher eine Diskussion der technischen Möglichkeiten zur Fachanwendungs-Migration.

Das vierte Kapitel der Arbeit bietet eine tiefergehende Erläuterung von Wine aus technischer Sicht und erläutert den von Wine verfolgten Ansatz sowie die Stellen, an denen Probleme bei der Ausführung von Windows-Software unter Linux/Wine auftreten können. Des weiteren werden dort andere Softwareprodukte und -konzepte kurz beschrieben, die für das entwickelte Programmsystem genutzt wurden.

Im fünften Kapitel wird diskutiert, welche Teilprobleme bei der Implementierung des Verfahrens zu lösen waren, und wie diese angegangen wurden. Es bietet damit auch eine Übersicht über das System.

Kapitel sechs stellt das entwickelte Verfahren mit seiner Implementierung im Detail vor. Jeder Teil der Analyse und des Programmsystems wird ausführlich erläutert und dargestellt. Dazu zählt u.a. die Objektstruktur des Systems, die Aufbereitung der Daten in für Menschen verständlicher Form, die Datenbankstruktur, und die verschiedenen Schnittstellen des Systems. Ebenso werden die gefällten Entwurfsentscheidungen diskutiert.

Die Durchführung von Analysen mittels des Systems wird anhand eines kommunalen Softwareprodukts im siebten Kapitel dargestellt. Im achten Kapitel wird erläutert, in welcher Weise sich Fachanwendungs-Migrationen mit Hilfe des vorgestellten Verfahrens effizienter durchführen lassen. Es werden

dabei Daten aus einem Migrationsprojekt präsentiert, bei dem das Verfahren zum Einsatz kam. Zudem werden weitere Vorgehensweisen für die Stadt Böblingen und Erweiterungs- und Anwendungsmöglichkeiten des Systems skizziert.

Kapitel 2

Problemstellung

2.1 Ansatz

Das in dieser Arbeit vorgestellte Verfahren ist das Resultat eines längeren Arbeits- und Evaluierungsprozesses, an dessen Anfang die einfache Frage stand, welche technischen Probleme es bei einer konkreten Migration einer Windows-basierten Infrastruktur auf Linux geben würde, und ob sich einige dieser Probleme auf technische Weise lösen ließen.

Durch die freundliche Vermittlung von Prof. Spruth konnte die Stadt Böblingen dafür gewonnen werden, ihre Infrastruktur als Basis für die Evaluierung eventueller Migrationsprobleme zur Verfügung zu stellen. In Böblingen war zu Beginn der Arbeit fast flächendeckend Windows NT 4 und Microsoft Office 97 im Einsatz, dessen Ablösung duch Windows XP und Microsoft Office XP unter anderem wegen dem Ende des Supports für NT seitens Microsoft im Raum stand. Diese Situation wurde als Chance gesehen, die Alternative Linux hinsichtlich ihrer Machbarkeit einer genaueren Betrachtung zu unterziehen.

Fachanwendungen Sehr schnell wurde klar, daß das größte Problem bei einer Migration im kommunalen Bereich das der Fachanwendungen sein würde. Unter Fachanwendungen versteht man Fachsoftware, die sehr spezielle Einsatzzwecke erfüllt, also in Kommunen beispielsweise Programme für die Verwaltung von Sozialhilfeanträgen. Kommunale Fachsoftware hat einige weitere Eigenschaften, die ihre Rolle problematisch machen:

- Anzahl. In Böblingen sind gut 100 Fachanwendungen im Einsatz, die die unterschiedlichsten Zwecke für die Verwaltung erfüllen.

- Spezialisierung. Die Programme sind in der Regel auf ihren Einsatzzweck höchst spezialisiert und bilden meist gesetzliche Vorschriften ab, die von Bundesland zu Bundesland variieren können. Daher ist es oft schwer, Programme zu ersetzen.

- Heterogenität. Die Einsatzzwecke der Programme sind extrem unterschiedlich und reichen von der Verwaltung der Stundenpläne der Grundschulen über CAD-basierte Baugenehmigungsverfahren bis zu standesamtlichen Aufgaben. Das schlägt sich auch auf technischer Ebene nieder. Von Anwendungen, die nur aus einer einzigen .EXE-Datei bestehen, bis hin zu komplexen Client/Server-Verfahren mit Datenbankanbindung ist praktisch jede Zwischenstufe vertreten.

- Unverzichtbarkeit. Die Fachanwendungen sind für die Verwaltung „unternehmenskritisch“, d.h. der Betrieb kann ohne sie nicht aufrechterhalten werden. Sie sind gewissermaßen die *raison d'être* der kommunalen IT-Infrastrukturen.

Hinzu kommen zwei weitere Punkte, die insbesondere für die Migration nach Linux gelten:

- Es existieren fast nie alternative FLOSS[1]-Programme, die die Fachanwendungen ersetzen könnten.

- In praktisch keinem Fall wird seitens des Softwareherstellers eine Linux-Version angeboten.

Somit sollte der Fokus der Diplomarbeit darauf liegen, eine möglichst umfassende technische Lösung für die Problematik der Fachanwendungen zu finden. Die hier vorab sinnvolle Diskussion der möglichen technischen Ansätze wird in Kapitel 3 vertieft. Der Ansatz den die freie Windows-API-Implementierung Wine verfolgt erwies sich hier als der vielversprechendste, weswegen auf dieser Technik aufgesetzt wurde.

2.2 Zielsetzung

Die Zielsetzung, eine möglichst automatische und formal abgesicherte Aussage über die Fähigkeit zur Ausführung von Windows-basierter Software unter Linux treffen zu können, sollte durch ein Programmsystem erreicht werden, das es zu entwickeln und zu implementieren galt. Dieses sollte die Analyse von Software in möglichst vielen Fällen ermöglichen und seine Ergebnisse so weit wie möglich automatisch, d.h. insbesondere mit geringem menschlichem Arbeitsaufwand, erbringen können. Gleichzeitig erscheint es wünschenswert, einmal analysierte Software nicht mehr für jede weitere Untersuchung vorliegen haben zu müssen, so daß die Speicherung von „Software-Abbildern“ und ihre spätere Benutzung in Analysen möglich sein soll. Ebenso sollen Analyseergebnisse abgespeichert und bei Bedarf abgerufen werden können.

[1] „Freie und Open Source Software“ wird im Folgenden mit dem dafür gebräuchlichen Akronym FLOSS abgekürzt.

Aufgrund des kontinuierlichen Entwicklungsprozesses an Wine ist es erforderlich, auch die Möglichkeit zu Analysen mit früheren und zukünftigen Wine-Versionen zu gewährleisten.

Wegen der vielfältigen denkbaren Anwendungsmöglichkeiten des Systems soll der Schwerpunkt auf einen sauberen und flexiblen algorithmischen Unterbau gelegt werden, während die graphische Aufbereitung eine weniger zentrale Rolle bei der Umsetzung spielen wird. Die Implementierung verschiedener Anwendungen soll über eine klar definierte technische Schnittstelle (API) gewährleistet werden. Gleichzeitig ist eine kommandozeilenbasierte Benutzung wünschenswert, um beispielsweise skriptgesteuerte Auswertungen, Neuauswertungen, oder das Abrufen bereits durchgeführter Analysen zu ermöglichen. Eine textuelle Ausgabe von Resultaten der Software-Untersuchung soll ebenfalls gewährleistet werden.

Trotz der erwarteten großen Datenmenge die es für Software-Analysen zu berücksichtigen gilt soll der Ressourcenverbrauch so gering wie möglich gehalten werden. Ziel ist es, die Implementierung so performant durchzuführen, daß sie auf marktüblichen PCs eingesetzt werden kann. Dies impliziert, daß eine möglichst plattformunabhängige Entwicklung des Systems wünschenswert ist. Da mit dem Programmsystem insgesamt eine Kostenreduktion von Linux-Migrationen angestrebt wird, wird dem Einsatz von kostenfrei verfügbaren Komponenten, insbesondere Freier und Open Source Software, Priorität eingeräumt.

2.3 Umsetzung

Um die Speicherung von Software-Abbildern sowie Analyseergebnissen zu ermöglichen erscheint es sinnvoll, eine Datenbank als Ablagemöglichkeit zu nutzen. Damit diese nicht zum Nadelöhr hinsichtlich der Ausführungsgeschwindigkeit wird, wird bei der Wahl des Datenbanksystems der Geschwindigkeit von Datenabruf und -speicherung Vorrang vor anderen Aspekten eingeräumt. Gleichzeitig ist eine durchdachte Repräsentation der im System verwendeten Daten notwendig, damit weder durch den Einsatz dieser (abstrakten) Datenstrukturen noch durch Speicherung in der Datenbank relevante Informationen verloren gehen oder sich zukünftig als relevant erweisende Informationen nicht ohne Weiteres im System abgelegt werden können.

Die Gewährleistung von Plattformunabhängigkeit wird sich in den zur Implementierung verwendeten Technologien, insbesondere den Programmiersprachen, niederschlagen müssen. Zusätzlich ist eine saubere Dokumentation der technischen Programmschnittstellen Voraussetzung für die Erfüllung der Anforderung, daß effizient weitere Anwendungen des vorgestellten Verfahrens programmiert werden können. Wegen der angestrebten Performanz des Systems sollen Zwischenergebnisse ebenfalls in der Datenbank abgelegt

werden, und eventuell vorhandene Parallelität ausgenutzt sowie nicht zeitkritische Teile nebenläufig implementiert werden.

Aufgrund der Neuhheit des zu implementierenden Systems erscheint es wahrscheinlich, daß zu späteren Zeitpunkten Erweiterungen und Veränderungen an den im System eingesetzten Komponenten oder an den Datenstrukturen vorgenommen werden sollen. Da einige der eingesetzten Technologien als Freie Software zur Verfügung stehen ist es ratsam, zukünftige Verbesserungen dieser Teilkomponenten durch die Open-Source Community in die Konstruktion des Systemes einzuplanen um so von ihnen profitieren zu können. Um dies zu gewährleisten wird eine modularisierte Umsetzung der Funktionalitäten angestrebt.

2.4 Vorgehensweise

Prototyp Es wurde nach der erläuterten Phase der Bestandsaufnahme in Böblingen und einer Evaluierung der technischen Lösungsansätze ein Ansatz entwickelt, mit dem die Migration von Fachanwendungen auf der Basis von Wine erheblich verbessert werden kann. Diese zunächst nur auf dem Papier existierende Sammlung von Algorithmen und Ideen wurde in einem ersten Schritt in Form eines Prototypen in Böblingen implementiert, um noch vorhandene Unsicherheiten bezüglich einiger Details des entwickelten Verfahrens zu klären. Der IT-Leiter der Stadt Böblingen, Herr Klaus Gödde, stellte hierfür einen Arbeitsplatz in der IT-Abteilung des Rathauses sowie die kommunale Software als Testbett zur Verfügung.

Der Prototyp hatte vor allem zum Ziel, die Strukturen von Windows-basierter Software automatisiert zu ermitteln und zu veranschaulichen, wofür u.a. der Graphenzeichner *yed* des am Lehrstuhl von Prof. Dr. Kaufmann entstandenen Spin-Offs *yworks* zum Einsatz kam.

Implementierung Nachdem so verifiziert worden war, daß der verfolgte Ansatz die erwünschten Ergebnisse bringen würde, wurde eine Reimplementierung des Verfahrens durchgeführt, die in dieser Arbeit vorgestellt werden soll. Diese Implementierung wurde umfassend mit Hilfe der Böblinger Kommunalsoftware getestet und verbessert. Gleichzeitig wurde mit Softwareherstellern Kontakt aufgenommen um von dieser Seite Feedback und Anregungen hinsichtlich möglicher Verbesserungen des Verfahrens zu bekommen.

München Zudem wurde Kontakt mit der Stadt München hergestellt, die in einer europaweit einmaligen Entscheidung eine Migration auf das Linux Betriebssystem beschlossen hat[2]. München stellte mehrere Anwendungen

[2]Die Stadt Schwäbisch Hall hatte bereits Ende 2002 eine Migration auf eine Linux-basierte Infrastruktur beschlossen; dieses Projekt ist mittlerweile weitestgehend abgeschlossen. Quelle: `http://www.heise.de/newsticker/meldung/32640`, zuletzt besucht am

zum Testen der Implementierung zu Verfügung. Die Zusammenarbeit mit der dortigen Stadtverwaltung erwies sich als für beide Seiten sehr gewinnbringend; das Münchner Feedback wurde dankbar in die Implementierung des Verfahrens aufgenommen.

6.12.2004.

Kapitel 3

Technologien zur Migration im Vergleich

Für den Betrieb von Windows-basierten Fachanwendungen unter dem Betriebssystem Linux gibt es aus technischer Sicht mehrere Möglichkeiten. Es erscheint geraten, die gewählte Basis-Technologie – Wine – mit anderen am Markt befindlichen Produkten und Ansätzen zu vergleichen, um sie hinsichtlich ihrer Mächtigkeit und ihrer Zukunftssicherheit einordnen zu können. Daher soll das folgende Kapitel die vier verbreitetsten technologischen Ansätze zum Betrieb von Windows-Applikationen unter Linux jeweils kurz vorstellen und vergleichen. Der Fokus liegt dabei auf den wesentlichsten technischen und wirtschaftlichen Aspekten. Für eine ausführlichere Darstellung, die jedoch neuere Entwicklungen nicht berücksichtigen kann, sei auf den Migrationsleitfaden der Bundesregierung [BMI 2003] verwiesen.

3.1 Terminalserver

Begriff Für den Begriff „Terminalserver“ gibt es bedauerlicherweise keine einheitliche Definition, was möglicherweise der Vielfalt der am Markt vorhandenen Lösungen geschuldet ist. Im Folgenden wird unter einem Terminalserver ein Serverrechner verstanden, der Programme ausführt und dabei deren Benutzeroberfläche anderen Rechnern (Clients) zur Verfügung stellt. Der Programmcode wird also auf dem Terminalserver ausgeführt, während der Benutzer mit dem Programm an einem physisch getrennten Clientrechner interagiert. Die Software, die diese Dienste bereitstellt, wird im Folgenden als Terminalserver-Software bezeichnet. Bestandteil dieser Software ist ein Clientprogramm, das die Darstellung des auf dem Terminalserver ausgeführten Programms auf dem Client übernimmt.

Diskussion Bei einer Migration von Windows-basierten Systemen auf Linux bietet sich insbesondere für Fachanwendungen der Einsatz eines Windows-

Terminalservers an, der die unter Linux nicht ausführbaren Fachprogramme für Linux-Clients zur Benutzung zur Verfügung stellt. Eine solche Lösung bietet neben der Möglichkeit fast beliebige Windows-Software weiter zu betreiben einige weitere Vorteile:

- Zentrale Verwaltung. Die Verwaltung der Client-Software und ihrer Einstellungen und Upgrades kann weitestgehend zentralisiert erfolgen.
- Clients. Die Hardwareanforderungen für die Clients sinken durch den Betrieb der Anwendungen auf dem Terminalserver. Zudem sind Client-Rechner durch die zentralisierte Verwaltung und Installation leicht austauschbar.
- Sicherheit. Werden die Daten zentral auf dem Terminalserver gespeichert, so verringert sich dadurch die Gefahr von Datenverlust. Zudem können die Clientsysteme ohne Wechselmedien bzw. ohne Permanentspeicher konstruiert werden (*Thin Clients*), wodurch die Gefahr von Manipulation oder Diebstahl von Daten durch Unbefugte sinkt.

Dem gegenüber stehen jedoch auch mehrere Nachteile:

- Abhängigkeit. Ausfälle des Terminalservers betreffen gleichzeitig mehrere Anwender und Programme (*Single Point of Failure*).
- Ressourcenbedarf. Die Terminalserver müssen über deutlich erhöhte Ressourcenausstattung insbesondere bezüglich des Arbeitsspeichers verfügen, da die Belastung durch die Anwendungen nun konzentriert auf dem Server anfällt.
- Netzwerkverkehr. Die Kommunikation zwischen Terminalserver und -client erfolgt über das Netzwerk. Insbesondere bei grafiklastigen Anwenungen kann sich so der Netzwerkverkehr drastisch erhöhen. Es existieren aber auch Anwendungen, bei denen er sich potentiell verringert.
- Anwendungen. Nicht alle Programme sind für die Ausführung auf Terminalserversystemen geeignet. Darunter fallen insbesondere solche, die Systemdateien im Exklusivzugriff geöffnet halten. Hier sind administrative Eingriffe oft unausweichlich.

Markt Es gibt am Markt eine Vielzahl von Terminalserverlösungen mit zum Teil sehr unterschiedlichen Leistungsmerkmalen. Für das in dieser Arbeit betrachtete Szenario – Betrieb von Windows-Anwendungen unter Linux mittels Terminalserver – sind aber Windows-basierte Terminalserver mit Linux-Clients von Belang. Für diesen Bereich bietet der Hersteller Citrix[1] seine Produkte an.

[1] Homepage von Citrix: `http://www.citrix.de/`, zuletzt besucht am 2.11.2004.

Citrix Terminalserver bieten aus technischer Sicht die bereits diskutierten Vor- und Nachteile. Das Lizensierungssystem der Produkte ist jedoch leider ausgesprochen komplex, weshalb beispielsweise in der Zeitschrift iX [iX 2004] sogar von „Lizenzurwald“ die Rede ist. Es bleibt festzuhalten, daß die Ausführung von Windows-Anwendungen unter Citrix mit Linux-Clients kaum deutlich kostengünstiger ist als die direkte Ausführung unter Windows.

Bewertung Obwohl Terminalserver aus technischer Sicht mindestens eine elegante Übergangslösung für das Problem der Fachanwendungen darstellen könnten sind am Markt leider keine Produkte erhältlich, die einen solchen Betrieb von Windows-Applikationen unter Linux deutlich kostengünstiger gestalten würden als einen Betrieb unter Windows. Mit Sicherheit wird bei Einsatz von Citrix-Terminalservern ein erheblicher Teil des Einsparpotentials an Lizenzkosten, das eine Linux-Migration bietet, verschenkt. Die komplizierte Lizenzstruktur der Citrix-Produkte kann ebenfalls als problematisch angesehen werden.

3.2 Hardwareemulation

Ein grundsätzlich immer zu berücksichtigender Ansatz bei der Ausführung von Software unter anderen Bedingungen als den für ihren Betrieb vorgesehenen ist eine Emulationslösung. Dabei bietet eine Emulation von Hardware die Basis für die Ausführung der Softwareumgebung, die für die Anwendungs-Software vonnöten ist. Im betrachteten Szenario – Ausführung von Windows-Software unter Linux – würde das bedeuten, einen Emulator unter Linux zu starten, der eine x86-basierte Maschine emuliert. Auf diesem virtuellen PC würde dann Windows in der gewünschten Version eingerichtet, um darunter dann Windows-Software zur Ausführung zu bringen.

Ansatz Ein klarer Vorteil dieser Herangehensweise ist der, daß damit praktisch beliebige Windows-Software unter Linux ausgeführt werden kann, denn es läuft ja de facto ein Windows unter Linux[2]. Man erkauft jedoch diesen Vorteil mit einer ganzen Reihe Nachteile:

- Performance-Verlust. Dadurch, daß der Emulator selbst eine Software ist, die Hardware emuliert, auf der ein Windows-Betriebssystem ausgeführt wird, entstehen gegenüber der direkten Ausführung von Windows auf der realen Hardware des PCs Einbußen in der Ausführungsgeschwindigkeit. Diese sind teilweise erheblich. Als Faustregel kann gelten, daß selbst bei nicht grafiklastigen Anwendungen mindestens

[2] Lediglich bei der Anbindung von eventuell benötigter Spezial-Hardware (Dongles etc.) an den emulierten PC gibt es möglicherweise Probleme.

die Hälfte der Performanz gegenüber direkter Ausführung auf der realen Hardware verloren geht.

- Usability. Es wird die Ausführung eines kompletten Windows-Betriebssystems emuliert und als Linux-Anwendung auf dem Bildschirm des Anwenders angezeigt. Das bedeutet, daß der Anwender auf seinem Linux-Desktop ein Fenster mit einem unter der Emulation laufenden Windows-Desktop angezeigt bekommt, auf dem dann die gewünschten Applikationen laufen. Diese „Fenster auf Desktop in Fenster auf Desktop"-Darstellung ist durchaus als mindestens verwirrend zu bezeichnen. Abbildung 3.1 zeigt eine Windows-Emulation unter Linux[3].

- Integration. Eng mit dem Usability-Aspekt verknüpft ist das Problem der mangelnden Integration in den Linux-Desktop. Es handelt sich bei dem emulierten Windows-System um einen virtuellen PC, der von dem Linux-System aus lediglich über das Netzwerk, nicht jedoch über Techniken wie Drag&Drop erreichbar ist. Gerade dies ist Anwendern schwer zu vermitteln und führt oft zu erheblicher Konfusion. Zudem ist z.B. das Kopieren von Dateien über Netzwerk gegenüber Drag&Drop aufwändiger und unkomfortabler.

- Wartung. Der virtuelle Windows-PC muß trotz seiner „Virtualität" selbstverständlich wie eine reale Windows-Installation gewartet werden. Hierzu gehören aus die allfälligen Sicherheitsprobleme mit ihrer Beseitigung genauso wie die Anfälligkeit für Viren.

- Kosten. Es fallen nach wie vor die Kosten für eine Windows-Lizenz an, die unter dem Emulator installiert wird. Zusätzlich kostet in der Regel die Emulationssoftware, so daß diese Lösung prinzipbedingt mindestens die gleichen, meistens aber sogar höhere Lizenzkosten aufweist als eine Installation unter Windows auf einem realen PC.

Markt Es existieren am Markt drei Produkte, die für eine Hardwareemulation grundsätzlich in Frage kommen. Dabei handelt es sich um ein kommerzielles Programm und zwei Open Source-Projekte.

- vmware. Die kommerzielle Softwarelösung vmware wird vom amerikanischen Hersteller vmware Inc. produziert[4]. Sie erlaubt es, unter verschiedenen Gast-Betriebssystemen (u.a. Linux) verschiedene andere Betriebssysteme (u.a. Windows) auszuführen. Dabei wird ein kompletter PC mit üblicher Hardwareausstattung emuliert und teilweise auf

[3]Quelle des Screenshots: `http://www.vmware.com/products/desktop/ws_screens.html`, zuletzt besucht am 24.10.2004.

[4]Website des Herstellers: `http://www.vmware.com/`, zuletzt besucht am 29.10.2004.

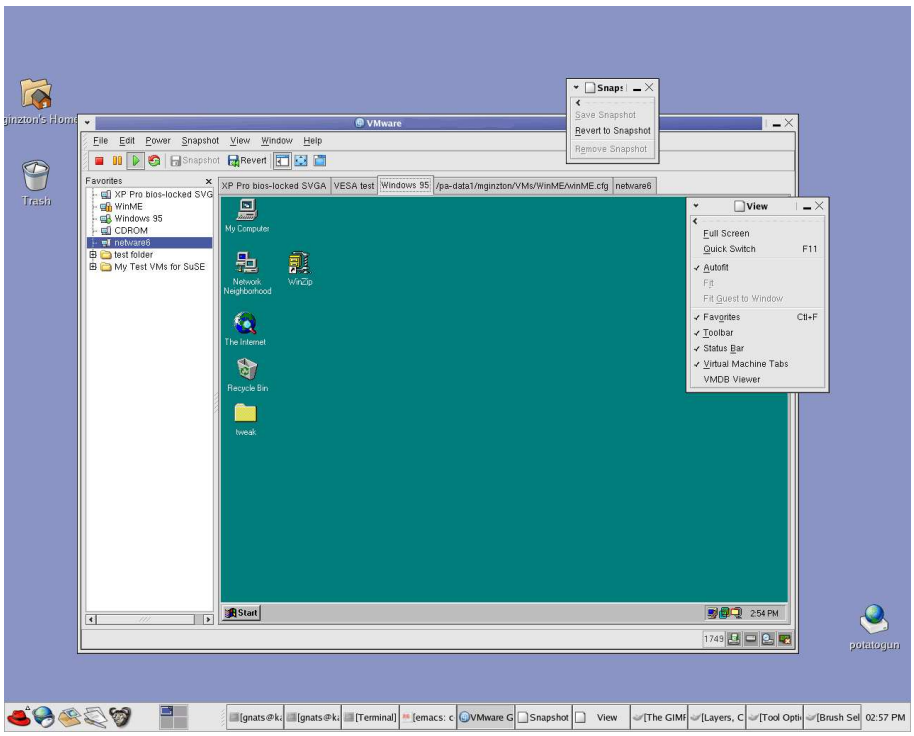


Abbildung 3.1: Screenshot von vmware Workstation 4. Ausgeführt wird ein Windows-Betriebsystem als Gast unter Linux mit der Gnome-Bedienoberfläche.

real vorhandene Hardware des Gast-PCs wie z.B Disketten- und CD-ROM-Laufwerke zugegriffen. vmware existiert als Workstation- und als Server-Variante.

- qemu. Bei qemu handelt es sich um ein Open Source-Projekt[5], das von Fabrice Bellard ins Leben gerufen wurde und mittlerweile in Version 0.6 vorliegt. qemu emuliert ebenfalls einen vollständigen PC und kann auf vorhandene Hardware wie Laufwerke ähnlich wie vmware zurückgreifen[6]. Das Projekt entwickelt sich bis dato durchaus dynamisch, so daß sich qemu möglicherweise zu einer echten Alternative zu vmware wandeln wird. Derzeit ist jedoch insbesondere die Ausführungsgeschwindigkeit der Emulation in der Regel zu langsam für ernsthafte Anwendungen.

- bochs. bochs ist – ähnlich wie qemu – ein Open Source-Projekt[7] und wurde von Kevin Lawton geschrieben. Technisch gesehen handelt es sich um einen Emulator für die Intel IA32-Architektur, der auf verschiedenen Plattformen lauffähig ist. bochs kann grundsätzlich beliebige Gastbetriebssysteme aufnehmen, besonders populär sind aber Linux und Windows. Die Performanz ist aber auch bei dieser Software, die bereits in der Version 2.1.1 vorliegt, als eher unzureichend insbesondere für graphische Anwendungen zu bewerten.

Bewertung Eine Emulationslösing ist vor allem deswegen attraktiv, weil unter ihr prinzipiell alle Windows-Software weiterhin unter Linux beetrieben werden kann. Dieser Vorteil wird aber mit einer ganzen Reihe von Nachteilen insbesondere hinsichtlich der Usability, der Wartung, und der Performance erkauft. Zudem wird nach wie vor eine Windows-Lizenz benötigt, so daß Emulationslösungen hinsichtlich der Lizenzkosten grundsätzlich teurer sind als der Betrieb under Windows alleine. Will man sich endgültig von Windows-Betriebssystemen lösen, so ist dieser Emulationsweg als Zwischenschritt denkbar, jedoch nicht als langfristige Alternative oder Lösung.

[5] Homepage des qemu-Projektes: `http://fabrice.bellard.free.fr/qemu/`, zuletzt besucht am 24.10.2004.

[6] Der Vollständigkeit halber sei erwähnt, daß qemu im sog. *User Mode* auch in der Lage ist, z.B. Wine auf anderen Architekturen als Intel IA32 auszuführen. Somit ist qemu auch ein nützliches Tool zum Cross-Compiling bzw. Cross-Debugging.

[7] Homepage des bochs-Projektes: `http://bochs.sourceforge.net/`, zuletzt besucht am 24.10.2004

3.3 Win4Lin

Bei Win4Lin handelt es sich um ein kommerzielles Produkt der Firma Netraverse[8], das zum jetzigen Zeitpunkt in der Version 5 vorliegt. Es ermöglicht den Betrieb von Windows und damit Windows-basierter Anwendungen unter dem Linux-Betriebssystem, verfolgt dabei aber aus technischer Sicht einen anderen Ansatz als Hardwareemulationen wie vmware.

Ansatz Win4Lin emuliert nicht wie z.B. vmware einen vollständigen PC, sondern stellt die vom Windows-Betriebssystem benötigten Dienste durch Module zur Verfügung, die in den Linux-Kernel geladen werden. Die Dateien des ausgeführten Windows-Betriebssystems werden während dessen Installation dergestalt abgeändert, daß die auf die von den Linux-Kernelmodulen zur Verfügung gestellte Funktionalität zurückgreifen. Ein wesentlicher Vorteil dieses Ansatzes liegt also darin begründet, daß die Ausführung in der Regel deutlich schneller vonstatten geht als unter vmware. Ebenso wie die der Ansatz der Hardwareemulation lassen sich mit Win4Lin grundsätzlich beliebige Windows-Programme ausführen. Es bestehen jedoch einige Nachteile und Einschränkungen:

- Windows-Versionen. Win4Lin unterstützt nur die Consumer-Varianten des Windows-Betriebssystems, also Windows 95, 98 und ME.

- Wartung. Weil die originären Windows-Module abgeändert werden, damit sie auf die Services der Win4Lin-Kernelmodule zurückgreifen, muß bei Updates dieser Windows-Komponenten mit Vorsicht vorgegangen werden. Wird beispielsweise durch ein Sicherheitsupdate via „Windows Update“ ein solches Modul überschrieben, kann das System in einen inkonsistenten Zustand geraten.

- Hardware. Es lassen sich maximal 128 MB RAM für das unter Win4Lin laufende Windows zuordnen. Die Schnittstellen USB und DirectX werden nicht unterstützt.

- Usability. Win4Lin führt die Windows-Installation genau wie vmware in einem eigenen Fenster aus, so daß hier die gleichen Einschränkungen gelten („Fenster auf Desktop in Fenster auf Desktop“-Darstellung). Abbildung 3.2 zeigt einen Windows-Desktop unter Linux und Win4Lin. Immerhin können aber Verzeichnisse im Linux-Dateisystem als Laufwerke dargestellt werden, was den Dateiaustausch vereinfacht.

[8] Homepage von Netraverse: `http://www.netraverse.com/`, zuletzt besucht am 24.10.2004.

Abbildung 3.2: Screenshot von Win4Lin 4. Ausgeführt wird ein Windows 98-Betriebsystem als Gast unter Linux mit der Gnome-Bedienoberfläche.

Bewertung Win4Lin verfügt gegenüber einer Hardwareemulation über den Vorteil einer höheren Performanz der Programmausführung. Es besitzt jedoch vergleichbare Nachteile. Zudem ist die Zahl der unterstützten Windows-Versionen, der verfügbare Arbeitsspeicher, und die verfügbaren Hard- und Softwareschnittstellen geringer. Wie z.B. vmware auch benötigt Win4Lin eine Windows-Lizenz, zu der sich noch die Kosten für Win4Lin addieren. Somit ist dieser Lösungsansatz hinsichtlich der Lizenzkosten ebenfalls grundsätzlich teurer als ein Betrieb unter Windows. Die Usability ist nur als leicht besser zu bewerten. Zudem ist zu befürchten, das mittelfristig viele Anwendungen nicht mehr unter den unterstützten Windows-Versionen lauffähig sein werden.

3.4 Wine

Bei Wine handelt es sich um ein Open Source-Projekt, das seit 1993 besteht. Es hat sich zum Ziel gesetzt, eine freie Nachimplementierung der Windows-API aufsetzend auf Unix und X11 durchzuführen. Somit stellt Wine keine Emulation dar, sondern bildet eine Zwischenschicht zwischen der Unix- bzw. Linux-API und X11 auf der einen Seite und Windows-Anwendungen auf der anderen Seite. Das Projekt verfügt über eine sehr aktive Community und hat inzwischen über eine Million Codezeilen.

Ansatz Wine bildet die Schnittstelle nach, die Windows-Applikationen vom Betriebssystem erwarten, und bildet die Funktionalität dieser API auf die von Unix und X11 zur Verfügung gestellte API ab. Es handelt sich also technisch gesehen um eine Win32-API-Implementierung unter Unix. Obwohl Linux die Hauptzielplattform ist, läßt sich Wine auch unter FreeBSD und Solaris kompilieren und einsetzen. Wine kann unmodifizierte Windows-Programme bei ihrem Start unter Linux in den Speicher laden und führt diese dann in Verbindung mit den von ihm bereitgestellten Modulen aus.

Aufgrund dieses Ansatzes kann Wine im Prinzip beliebige Windows-Software ausführen, sofern die von ihr benötigten APIs bereits nachimplementiert zur Verfügung stehen. Grundsätzlich können auch originale Windows-Module eingebunden werden. Da es sich bei Wine um eine *Nach*implementierung handelt, werden besonders neue Funktionalitäten in der Regel nicht oder noch nicht unterstützt. Wine implementiert mittlerweile aber die Windows-Bibliotheken mit den wichtigsten Funktionen, so daß viele und insbesondere ältere Software ausführbar ist.

Dadurch, daß prinzipiell sämtliche Windows-Software mit dem gewählten Ansatz unter Linux zum Laufen gebracht werden kann, und durch die direkte Einbindung in das Linux-Betriebssystem ist mit Wine die optimale Integration von Windows-Anwendungen in Linux zu erreichen. Es ergeben sich also folgende Vorteile von Wine:

- Integration. Mittels Wine werden Windows-Anwendungen optimal in den Linux-Desktop integriert. Abbildung 3.3 zeigt die Ausführung verschiedener Windows-Programme unter Linux/Wine.

- Performance. Es muß kein zusätzliches Betriebssystem ausgeführt werden wie bei einer Hardwareemulation oder unter Win4Lin, so daß die Ausführungsgeschwindigkeit unter Wine oft der unter Windows kaum nachsteht. Es sind sogar Programme bekannt, die unter Wine schneller laufen als unter Windows. Der Speicherbedarf der Applikationen ist ebenfalls der gleiche wie unter Windows.

- Zukunftssicherheit. Der von Wine gewählte Ansatz ist grundsätzlich auf alle Windows-Software anwendbar, so daß keine prizipielle Einschränkung auf gewisse Windows-Versionen besteht.

- Kosten. Wine ist Open Source Software bzw. seit 2002 sogar Freie Software unter der LGPL-Lizenz und steht deswegen unentgeltlich zur Verfügung. Der Quelltext ist ebenfalls frei verfügbar.

- winelib. Bestandteil von Wine ist u.a. die sogenannte winelib. Dabei handelt es sich um eine Bibliothek, die die von Wine implementierten Funktionen der Windows-API zur Verfügung stellt. Mittels winelib können im Quelltext einer Applikation sowohl Windows- als auch Unix-Betriebssystemaufrufe benutzt werden, was einen sehr eleganten und effizienten Weg zur Portierung von Windows-Software auf Unix bzw. Linux darstellt.

Dem gegenüber stehen einige Nachteile von Wine:

- Konfiguration. Wine ist ein sehr umfangreiches und komplexes Projekt, weshalb sich die Konfiguration von Wine oft sehr schwierig gestaltet und einiges an Know-How voraussetzt.

- Implementierung. Wines Implementierung der Windows-API ist prinzipbedingt nie vollständig. Es ist nicht möglich, im Vorhinein einer Windows-Applikation die von ihr benötigten APIs zuzuordnen, um so Aussagen über ihre Kompatibilität mit Wine zu treffen. Daher sind in der Regel langdauernde Tests von Windows-Software unter Wine nötig, um sie in den Produktivbetrieb zu übernehmen.

Markt Von Wine gibt es kommerzielle Varianten, für die vom Hersteller jeweils auch Support angeboten wird. Hier seien lediglich die zwei wohl populärsten genannt:

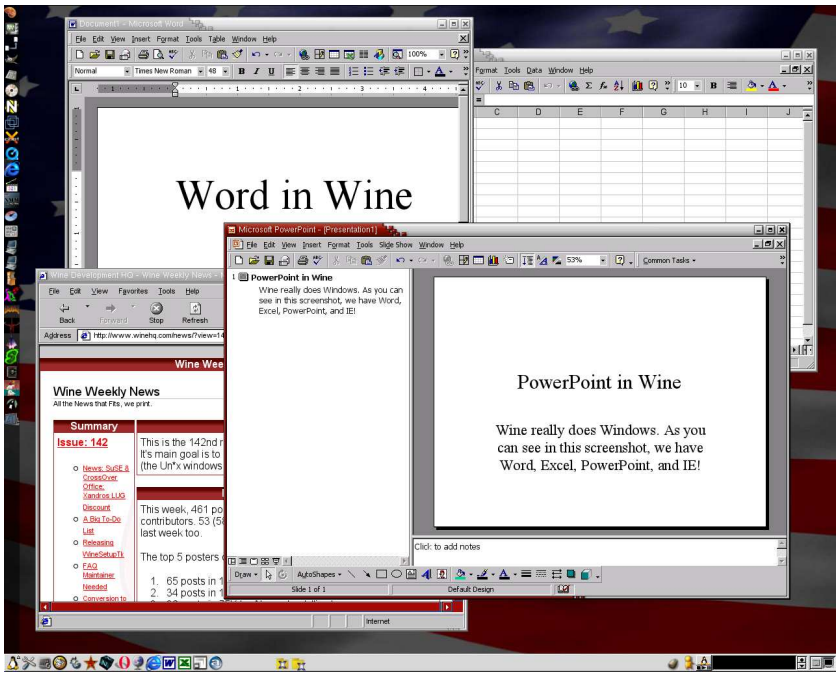


Abbildung 3.3: Screenshot eines Linux-Desktops mit mehreren Windows-Anwendungen, die unter Wine ausgeführt werden.

- Crossover Office. Hierbei handelt es sich um eine Version von Wine, die mit einer graphischen Oberfläche und einer sinnvollen Vorkonfiguration für viele populäre Anwendungen versehen ist. Unter COO, das von der amerikanischen Firma Codeweavers[9] auf den Markt gebracht wird, lassen sich zahlreiche populäre Windows-Programme ausführen, darunter Microsoft Office, Adobe Photoshop, Lotus Notes, sowie diverse Browser-Plugins. Der Quelltext der in Crossover Office enthaltenen Wine-Version ist aufgrund der Lizenzbestimmungen für Wine (LGPL-Lizenz) unter den gleichen Bedingungen verfügbar wie Wine.

- Cedega. Von der kanadischen Firma Transgaming[10] wird eine Variante von Wine gepflegt und vertrieben, der sich vornehmlich an Spiele-Benutzer richtet. Er stellt von Spielen üblicherweise benutzte APIs unter Wine zur Verfügung und erlaubt somit die Benutzung von Windows-basierten Spielen unter Linux.

Bewertung Aus technischer Sicht erscheint Wine der vielversprechendste Ansatz für die Integration von Windows-Anwendungen in das Linux Be-

[9] Homepage von Codeweavers: `http://www.codeweavers.com/`, zuletzt besucht am 24.10.2004.

[10] Homepage von Transgaming: `http://www.transgaming.com/`, zuletzt besucht am 24.10.2004.

triebssystem zu sein. Allerdings machen komplizierte Konfiguration und die unvollständige Implementierung der Windows-API in der Praxis Probleme. Für viele Szenarien ist eine lange Testphase vor dem eigentlichen Produktivbetrieb nicht durchführbar oder sinnvoll. Zudem sind solche Tests aufwendig und teuer, und können prinzipbedingt nie eine hundertprozentige Sicherheit hinsichtlich eventueller Probleme bei der Ausführung unter Wine liefern.

Kapitel 4

Grundlagen

Das Programmsystem das in dieser Arbeit vorgestellt wird benutzt mehrere bereits vorhandene und etablierte Open Source-Softwareprodukte für verschiedene Aufgaben. Darunter fallen das verwendete SQL-basierte Datenbanksystem MySQL, die freie Win32-API-Implementierung Wine, sowie das Framework JFreeReport für die automatisierte Generierung von Reports. Diese Softwareprojekte sollen in diesem Kapitel kurz vorgestellt werden um einen Einblick in Entwicklungsstand und -modell sowie die Features der jeweiligen Programme zu erhalten.

Des Weiteren ist es für das Verständnis des entwickelten Verfahrens zur Migration von Windows-Legacy-Applikationen auf Linux erforderlich, verschiedene technische Gegebenheiten genauer zu erläutern. Insbesondere das unter Windows verwendete Binärformat „Portable Executable“ ist von besonderem Interesse, da Eigenschaften dieses Binärformats benutzt werden, um benötigte APIs in Windows-Software zu ermitteln. Das Format kann nicht in seiner Gesamtheit präsentiert werden – hierfür sei auf das Standardisierungsdokument [PECOFF 1999] verwiesen –, die für das entwickelte Verfahren wesentlichen Eigenschaften werden aber dargestellt. Auch verschiedene tiefergehende Informationen zum Entwicklungsmodell und technische Einzelheiten im Wine-Projekt sind vonnöten und werden im entsprechenden Abschnitt dieses Kapitels ausgeführt.

4.1 Win32-API für Linux: Das Wine-Projekt

Bei Wine – einem rekursiven Akronym für *Wine Is Not an Emulator* – handelt es sich, wie der Name bereits andeutet, nicht um einen Emulator. Vielmehr ist Wine eine Implementierung der Windows-API unter Unix und X11, die es somit erlaubt, unmodifizierte Windows-Programme unter Linux auszuführen. Bestandteil des Projektes ist neben einem Loader für Windows-Executables und einer Nachimplementierung wesentlicher Teile der Win32-API auch eine Bibliothek namens winelib, mit Hilfe derer sich Windows-

Systemaufrufe in nativem Unix-Code verwenden lassen. Wine gehört zu den ältesten Projekten der freien Softwareszene.

Historie Noch vor dem Release der Linux-Kernelversion 1.0 begann das Wine-Projekt[1] in der zweiten Hälfte des Jahres 1993 seine Arbeit. Inspiriert von einem Projekt namens *Wabi*[2] der Firma Sun Microsystems, das die Ausführung von Windows-Programmen auf SPARC-Maschinen ermöglichte, zielte Wine darauf ab Windows 3.x-basierte Programme unter Linux und X11 lauffähig zu machen. Bereits 1994 übernahm Alexandre Juillard die Position des Maintainers, die er bis heute innehat.

Einen wichtige Entwicklung ergab sich 1998, als Corel die strategische Entscheidung traf, Linux für seine Produkte zu unterstützen. Dies bedeutete sowohl die Notwendigkeit der Entwicklung einer Linux-Distribution für Corels Applikationen als auch eine sehr weitgehende Unterstützung der Corel-Software unter Wine. Corel baute sehr gute Beziehungen zum Wine-Entwicklungsteam auf und nahm u.a. den Maintainer von Wine, Alexandre Juillard, unter Vertrag. Zwar stellte Corel sein Linux-Engagement im Jahr 2001 ein, doch Corels Einsatz hatte dem Wine-Projekt einiges an Dynamik verschafft.

Im Jahr 2002 entschied sich das Wine-Projekt nach längerer Diskussion unter den Entwicklern, künftige Versionen der Software unter der GNU Lesser Public License (LGPL) freizugeben. Damit war die Möglichkeit geschaffen, von der stetig wachsenden Welt der Freien Software zu profitieren, weil mit dieser Lizenz andere Freie Software leichter eingebunden werden kann. Gleichzeitig wurde so den Befürchtungen vieler Entwickler Rechnung getragen, es könnte kommerzielle Versionen von Wine geben, die ihre Weiterentwicklungen des Codes nicht in das Projekt zurückfließen lassen.

Derzeit hat Wine über 1.1 Millionen Zeilen Code und wächst stetig weiter. Jeden Monat werden von den Entwicklern zwischen vier und fünf Megabytes an Patches für das Projekt eingereicht, und der gesamte Quelltextbaum ist mittlerweile über 70 MB groß. Ein guter Teil der Entwicklungsarbeit wird von der Firma Codeweavers und deren Angestellten geleistet, die mittelbar aus Corels Linux-Engagement hervorgegangen ist und eine kommerzielle Version von Wine namens Crossover Office vermarktet und supportet. Unter Wine lauffähig sind so komplexe Programme wie Microsoft Office oder Lotus Notes.

Entwicklungsmodell Das Projekt folgt einem sehr offenem Entwicklungsmodell: Die Koordination der Entwickler erfolgt über öffentliche Mailingli-

[1] Eine ausführlichen Überblick über die Entwicklung des Wine-Projekts gibt die Seite `http://www.winehq.org/site/history`, zuletzt besucht am 2.11.2004.

[2] Für eine Übersicht über *Wabi* sei hier auf die Wabi-Dokumentation bei Sun verwiesen: `http://docs.sun.com/app/docs/doc/802-6306/6ia0mdt48?a=view`, zuletzt besucht am 2.11.2004.

sten und Newsgroups. Eingereichte Patches werden vom Maintainer Alexandre Juillard begutachtet und gegebenenfalls in den CVS[3]-Tree des Projektes aufgenommen.

Der CVS-Baum des Projektes ist ebenfalls im Lesemodus öffentlich für jedermann zugänglich. Einmal monatlich gibt es CVS-Snapshots zum Download, die den derzeitigen Stand der Entwicklung markieren, jedoch keinerlei Garantie bezüglich der Funktionalität machen. Es handelt sich hierbei nicht um offizielle Releases. Das Wine-Projekt hat jedoch in seiner gesamten Geschichte noch keine offizielle Version der Software herausgegeben, so daß man diese Snapshots durchaus als Versionen bezeichnen darf, wenn man ihren „Schnappschusscharakter“ im Gedächtnis behält.

Qualitätssicherung Die Entwickler von Wine verfolgen mehrere Strategien für die Qualitätssicherung, wie dies bei einem Projekt dieser Größenordnung unvermeidbar ist. Sämtliche eingereichten Patches werden vom Maintainer des Projektes hinsichtlich Korrektheit und Anwendung der für das Projekt geltenden Code-Richtlinien untersucht. Häufig wird von den Einreichern von Patches dann eine Verbesserung oder ein Rewrite verlangt.

Des weiteren wird versucht, die Korrektheit des Codes mit Test-Suiten abzusichern. Darunter werden Programme verstanden, die die Korrektheit der Funktionsweise von bestimmten API-Aufrufen überprüfen. Diese Testbette sind sowohl unter Linux als auch unter Windows lauffähig und müssen auf beiden Plattformen die gleichen Resultate liefern. Gelegentlich ist es wegen fehlender oder gar fehlerhafter Dokumentation seitens Microsoft erforderlich, bestimmte APIs im Wesentlichen durch Reverse Engineering nachzubilden, wobei die genannten Tests unverzichtbar sind. In einigen Fällen müssen für korrekte Funktionsweise von Windows-Programmen unter Wine sogar subtile Programmierfehler von Windows-APIs nachgebildet werden.

Bugreports werden mit dem in der Freien Softwareszene üblichen System Bugzilla[4] webbasiert verwaltet bzw. auf den Entwickler-Mailinglisten diskutiert. Bugzilla kommt außer bei Wine auch beispielsweise bei Mozilla[5], KDE[6], oder dem Linux-Kernel[7] zum Einsatz.

Struktur Ein aktueller Snapshot des Quelltextes ist stets von der Homepage des Projektes – `http://www.winehq.org` – als .tar-Archiv beziehbar. Entpackt man dieses Archiv, so findet sich der Quelltext in einem Verzeichnis

[3] Bei CVS handelt es sich um ein sehr verbreitetes System, mit deren Hilfe Entwickler gemeinsam an Programmen arbeiten können. Für eine Darstellung von CVS siehe die Homepage des Projektes `https://www.cvshome.org/`, zuletzt besucht am 7.12.2004, oder [Klaeren 2004, Seite 67ff.].

[4] Homepage von Bugzilla: `http://www.bugzilla.org`, zuletzt besucht am 2.11.2004.

[5] Homepage von Mozilla: `http://www.mozilla.org`, zuletzt besucht am 7.12.2004.

[6] Homepage von KDE: `http://www.kde.org`, zuletzt besucht am 7.12.2004.

[7] Homepage des Linux-Kernels: `http://www.kernel.org`, zuletzt besucht am 7.12.2004.

der Form wine-xyz, wobei xyz das Release-Datum darstellt. Beispielsweise würde die Wine-Release vom 8. April 2004 in das Verzeichnis wine-20040408 entpackt.

In weiteren Unterverzeichnissen finden sich dann Dokumentation (in documentation/) oder Programme, die mitgeliefert werden wie z.B. eine Nachimplementierung von Minesweeper, oder anderen etwas zentraleren Windows-Komponenten wie Notepad o.Ä. (programs/). Besonders interessant ist das Verzeichnis dlls/, in dem sich in zahlreichen Unterverzeichnissen Implementierungen verschiedenster Windows-Module von advapi32 bis wsock32 und x11drv finden.

Für jedes nachimplementierte Modul exisitiert ein sog. spec-File. Dabei handelt es sich um eine Datei, die tabellarisch aufgelistet in textuellem Format die in dem Modul zur Verfügung gestellten Funktionen (Exporte) aufführt. Spec-Files sind inhaltlich mit den Export-Tabellen von Portable Executables vergleichbar. Es werden neben Name und/oder Ordinalen der exportierten Funktionen auch falls nötig ihre Parameterlisten aufgeführt. Forwards, also die Weiterleitung von Exporten des Moduls auf Exporte eines anderen Moduls sind hier ebenfalls codiert.

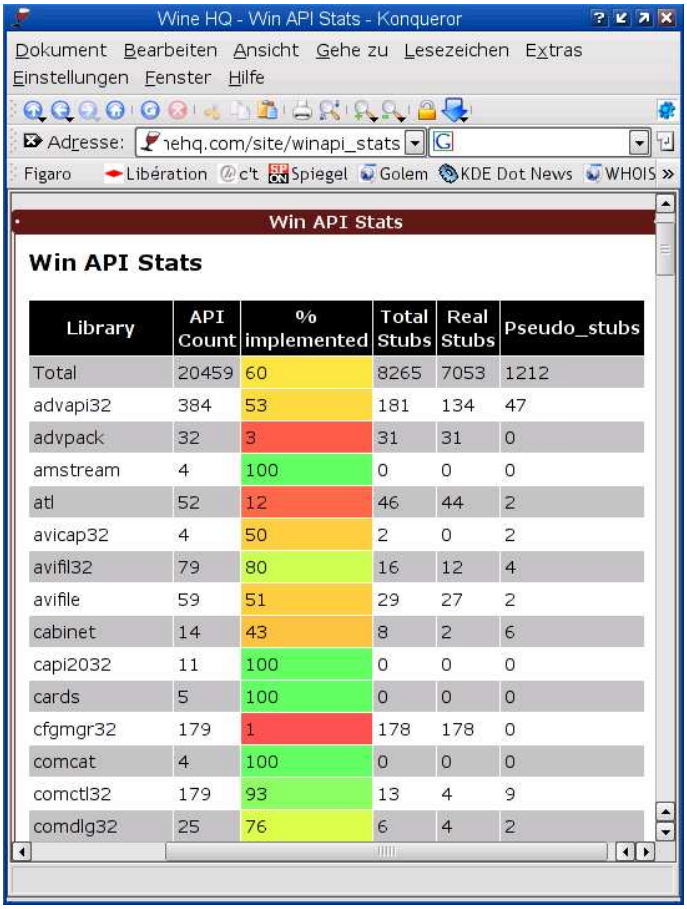
Spec-Files enthalten zudem noch Informationen über die Aufrufkonvention der jeweiligen Funktion, für die „stub“ eingetragen wird, falls es sich bei dem Export lediglich um einen Dummy-Eintrag handelt. Diese Stubs dienen dazu, bei der Kompilierung des Modul-Quelltextes einen gültigen Eintrag in der Export-Tabelle des Moduls zu erzeugen, damit der Loader das Modul lädt, wenn der entsprechende Dummy-Export von einem anderen Modul importiert wird. In spec-Files als stub bezeichnete APIs sind also solche, die in Wine noch nicht implementiert sind.

Darüberhinaus verfügt Wine über ein umfangreiches System von sog. Debug-Channels. Darunter versteht man Kanäle, in denen zur Laufzeit Debuginformationen nach Modul oder Teilsystem gruppiert übermittelt werden. Beispielsweise existiert ein Channel „loaddll“, der wenn aktiviert Informationen über geladene oder entladene Module ausgibt. Für die Kommunikation von Ereignissen oder auch Fehlfunktionen an diese Kanäle gibt es genaue Richtlinien bzw. Schnittstellen.

Um unvollständige Implementierung einer API zu Laufzeit an die Debug-Ausgabe zu kommunizieren gibt es in Wine das Makro FIXME. Eine Ausgabe zur Laufzeit sieht dann beispielsweise wie folgt aus:

```
fixme:cdrom:CDROM_GetInterfaceInfo not implemented for BSD
```

Unvollständige Funktionalitäten in einer API lassen sich somit im Quelltext von Wine an den Aufrufen von FIXME, die das Schlüsselwort „stub“ enthalten, ablesen. Eine weitere, granularere Unterscheidung sind sog. Semi-Stubs, die eine weiter fortgeschrittene Implementierung als die eines Stubs andeuten. Entsprechende Ausgaben sehen beispielsweise so aus:



Library	API Count	% implemented	Total Stubs	Real Stubs	Pseudo_stubs
Total	20459	60	8265	7053	1212
advapi32	384	53	181	134	47
advpack	32	3	31	31	0
amstream	4	100	0	0	0
atl	52	12	46	44	2
avicap32	4	50	2	0	2
avifil32	79	80	16	12	4
avifile	59	51	29	27	2
cabinet	14	43	8	2	6
capi2032	11	100	0	0	0
cards	5	100	0	0	0
cfgmgr32	179	1	178	178	0
comcat	4	100	0	0	0
comctl32	179	93	13	4	9
comdlg32	25	76	6	4	2

Abbildung 4.1: Die Win API Status-Seite auf WineHQ.com. Hier ist tabellarisch aufgelistet der Implementierungsstand der einzelnen DLLs in Wine zu sehen.

```
fixme:imm:ImmAssociateContext (0x7002a, 0x0): semi-stub
fixme:imm:ImmGetContext (0x7002a): stub
```

Die Verwendung des FIXME-Aufrufes im Quelltext sei exemplarisch an einer API aus der Nachimplementierung von MPR.DLL ausgeführt. Deutlich sichtbar ist hier, daß der Stub keinerlei echte Funktionalität implementiert. In der Regel sind Stubs API-Implementierungen, deren Aufruf immer fehlschlägt, d.h. die immer 0 oder `false` o.Ä. zurückgeben.

```
DWORD WINAPI MultinetGetErrorTextA( DWORD x, DWORD y, DWORD z )
{
    FIXME( "(%lx, %lx, %lx): stub\n", x, y, z );
    return 0;
}
```

Dank der Code-Richtlinien im Wine-Projekt, deren Einhaltung vom Maintainer auch strikt überwacht wird, ist es mit Hilfe der Spec-Files und dem

Quelltext möglich, sehr granular festzustellen, wie weit die Implementierung einer bestimmten API in Wine gediehen ist. Schön veranschaulicht wird dies anhand der in Abbildung 4.1 gezeigten „Win API Stats“-Seite, die etwas versteckt auf der Homepage des Projekts zu finden ist[8].

Benutzung Vorkompilierte Wine-Pakete existieren für die meisten verbreiteten Linux-Distributionen, darunter insbesondere SuSE und RedHat. Will man Wine selbst übersetzen, so läßt sich dies mittels der folgenden Kommandos erreichen, die im Hauptverzeichnis eines entpackten Wine-Snapshots auszuführen sind:

```
./configure
make depend
make
su -c 'make install'
```

Ausführen lassen sich Windows-Programme dann mittels Aufrufen wie

```
wine ./PROGRAMM.EXE
```

Voraussetzung dafür ist jedoch eine zuvor erfolgte Konfiguration von Wine.

Konfiguration Da es sich bei Wine um ein Projekt erheblichen Umfanges handelt, das zudem darauf abzielt, Funktionalitäten des gesamten Windows-Betriebssystems abzubilden, ist die Konfiguration von Wine eine schwierige und komplexe Aufgabe. Es ist jenseits der Zielsetzung dieser Arbeit, die Konfigurationsmöglichkeiten umfassend darzustellen. Im Grundsatz umfassen sie die Einstellungen der folgenden wesentlichen Punkte:

- Laufwerke. Wine kann beliebige Verzeichnisse als Windows-Laufwerke einbinden. Für diese virtuellen Laufwerke ist es möglich, den Typ des Laufwerks (Netzwerk-, Festplatte, CD-ROM etc.) ebenso wie den Dateisystemtyp, den Wine für diese Laufwerke meldet zu konfigurieren. Es lassen sich neben Festplatten auch Disketten-, CD-ROM-, und Netzwerklaufwerke konfigurieren.

- Registry. Wine legt ähnlich wie Windows einige seiner Konfigurationseinstellungen in der Registry ab. Diese kann mit *regedit*, einer Nachimplementierung des Windows Registry Editors, bearbeitet werden.

- Schnittstellen. Wine unterstützt sowohl serielle als auch parallele Ports, was aber vor Benutzung zunächst konfiguriert werden muß.

[8] Wine API Stats-Page: `http://www.winehq.com/site/winapi_stats`, zuletzt besucht am 2.11.2004.

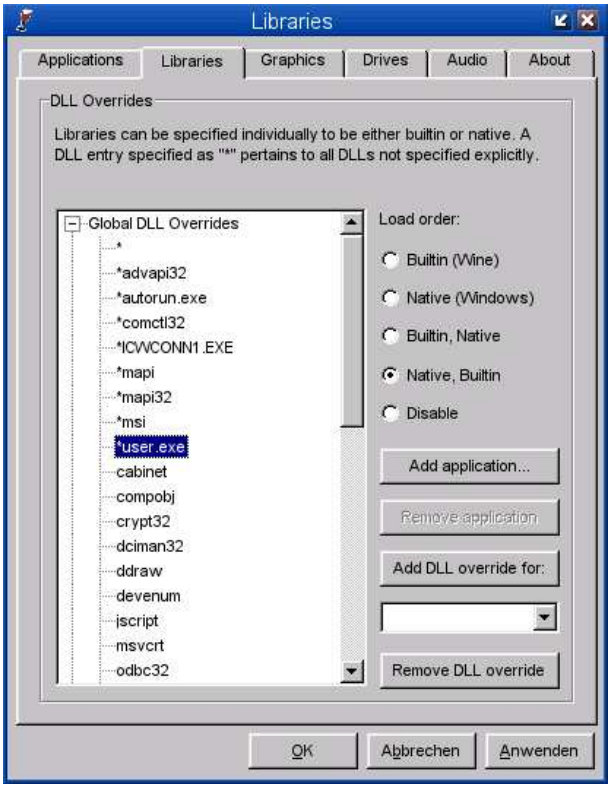


Abbildung 4.2: Screenshot von Winecfg, einem im Wine-Projekt enthaltenen graphischen Oberfläche für die Konfiguration von Wine.

- Sound. Wine bietet Schnittstellen zu zahlreichen Linux-Soundsystemen an, darunter die verbreiteten OSS- und ALSA-Schnittstellen[9] und dem KDE-Soundserver aRts. Die Soundausgabe von Windows-Software die unter Wine ausgeführt wird kann damit für die Applikation transparent unter Linux erfolgen.

- Windows-Version. Es besteht die Möglichkeit, das Verhalten einer vorher konfigurierten Windows-Version zu simulieren. Dies hat den Hintergrund, daß manche APIs z.B. unter den NT-basierten Windows-Varianten ein anderes Verhalten zeigen als unter den Consumer-Varianten.

- Drucker. Wine kann virtuelle Drucker bereitstellen und so Unix-Drucksysteme einbinden. Damit ist es möglich, aus Windows-Applikationen heraus unter der Benutzung aller Fähigkeiten von modernen Drucksystemen wie z.B. CUPS[10] zu drucken.

- Loader. Sicherlich der zentralste Punkt in der Wine-Konfiguration ist die Konfiguration der Verhaltens des Loaders. Konkret läßt sich für jedes Modul A einstellen wie sich der Loader verhalten soll, wenn von ihm verlangt wird, A zu laden. Der Loader kann stets auf die – möglicherweise unvollständige – Wine-Implementierung von A zurückgreifen (*builtin*), oder er kann eine im Dateisystem vorhandene Version von A laden (*native*), die aber möglicherweise nicht mit Wine kompatibel ist. Es ist auch möglich, Präzedenzregeln zu definieren und so der Wine-Version beispielsweise Vorrang zu geben und die native Version des Moduls nur dann zu laden, wenn die Wine-Implementierung nicht ausreicht (*builtin,native*). Im Folgenden sei ein Beispiel aufgeführt:

```
;; ole32.dll: Präferenz für Windows-DLL,
;; falls diese nicht vorhanden, Wine-Nach-
;; implementierung verwenden
"ole32" = "native, builtin"

;; Für cabinet.dll nie die Nachimplementierung
;; in Wine verwenden
"cabinet" = "native"

;; Default für alle anderen Module:
;; Nachimplememtierung in Wine verwenden
"*" = "builtin"
```

[9] Homepage des *Advanced Linux Sound Architecture* (ALSA) Projekts: `http://www.alsa-project.org/`, Homepage vom *Open Sound System* (OSS): `http://www.opensound.com/`, beide zuletzt besucht am 2.11.2004.

[10] CUPS ist das *Common Unix Printing System*, eines der verbreitetsten Drucksysteme unter Linux. Homepage des Projekts: `http://www.cups.org`, zuletzt besucht am 2.11.2004.

Zudem besteht die Möglichkeit, für bestimmte Applikationen andere Loader-Einstellungen zu verwenden als die globalen. Native Windows-DLLs arbeiten oftmals nicht fehlerfrei mit Wine-Modulen zusammen, so daß die korrekte Konfiguration von Wine mindestens als schwierig bezeichnet werden muß. Die Erstellung einer optimalen Konfiguration ist höchstens mit dem Quelltext und dem Entwicklungsstand von Wine intim vertrauten Entwicklern aufgrund ihrer Erfahrung möglich und bleibt zeitaufwändig.

Für die Konfiguration von Wine existieren auch graphische Oberflächen. Die im Wine-Projekt selbst enthaltene – *winecfg*, in Abbildung 4.2 dargestellt – implementiert eine recht umfassende GUI für die Konfigurationsmöglichkeiten von Wine. Das Programm ist aber noch nicht fertig gestellt, denn es fehlt insbesondere die Fähigkeit, die so graphisch erstellte Konfiguration auch in Registry bzw. Konfigurationsdatei zu schreiben. Eine andere, funktionalere GUI wäre das Programm *winesetuptk*, das im Internet auf der Seite des Wine-Projekts zum Download angeboten wird.

winelib Ein wichtiger Bestandteil von Wine ist die sog. winelib. Diese Bibliothek enthält die vom Projekt nachimplementierten Windows-APIs, die somit in eigenem Quellcode benutzt werden können. Man kann so eine auf der Windows-API basierende Applikation einfach unter Linux rekompilieren und dabei gegen die winelib linken, und erhält so eine Linux-Applikation. Es ist sogar möglich, im eigenen Quelltext Linux-API-Aufrufe und Windows-API-Aufrufe zu mischen, was eine Portierung von Windows-Applikationen auf Linux mittels winelib ebenso elegant wie einfach macht. Voraussetzung dafür ist, daß die zu portierende Applikation in C/C++ geschrieben ist.

4.2 Portable Executable Format

Das in dieser Arbeit vorgestellte Verfahren nutzt einige Eigenschaften des Portable Executable-Formats aus. Es handelt sich dabei um das Binärformat, das unter dem Windows-Betriebssystems verwendet wird. Es soll an dieser Stelle keine vollständige Einführung in das PE-Format erfolgen, sondern lediglich die Aspekte, die für die Entwicklung des Systems von Belang waren vorgestellt werden. Für eine detaillierte Beschreibung sei auf das entsprechende Standardisierungsdokument [PECOFF 1999] sowie auf den MSDN-Artikel [Inside PE 2002] verwiesen[11].

Beschreibung Das Portable Executable-Format ist ein Dateiformat, das in den 32- und 64-Bit-Versionen des Windows-Betriebssystemes benutzt

[11] Ebenfalls interessant ist der entsprechende Wikipedia-Eintrag, abrufbar unter `http://en.wikipedia.org/wiki/Portable_Executable`, zuletzt besucht am 6.11.2004.

wird. Dabei stellt PE eine modifizierte Version des Unix-Dateiformats Common Object File Format (COFF) dar, das unter Unix System V Release 3 benutzt wurde[12]. Es heißt deswegen „portable“, weil es auf allen Plattformen auf denen die genannten Windows-Varianten laufen eingesetzt wird, wobei für die 64-Bit-Versionen kleinere Erweiterungen nötig sind. Die Bezeichnungen PE und PE/COFF werden synonym gebraucht.

Im Wesentlichen definiert das PE-Format eine Datenstruktur, die für den Windows-Loader notwendige Informationen kapselt, um den enthaltenen ausführbaren Code zu handhaben. Dazu zählen insbesondere Tabellen für API-Import und -Export und Informationen für dynamisches Linken. Diverse andere Ressourcen (z.B. Icons) können ebenfalls in dem Format mit abgelegt werden.

Export Ein PE-Modul kann anderen Modulen seine Funktionalität ganz oder teilweise zur Verfügung stellen. Dies geschieht über den sog. Export von Funktionen des Moduls. Exporte, d.h. exportierte Funktionen, können von anderen Modulen eingebunden (Linking) und aufgerufen werden. Exporte eines Moduls werde in einem speziellen Abschnitt des exportierenden PE-Moduls aufgelistet, der sog. Export-Tabelle. Diese enthält neben der Einsprungaddresse des Exports auch noch seinen Namen (d.h. den Namen der exportierten Funktion) und eine *Ordinal* genannte Nummer des Exports. Die Angabe des Namens der Funktion ist nicht zwingend erforderlich, da Importe auch über die Nummer der exportierten Funktion erfolgen können.

Ein Modul kann eine exportierte Funktion auf eine exportierte Funktion y eines anderen Moduls C „weiterleiten“. Ein Modul das diesen Export importiert würde dann automatisch die Funktion y des Moduls C aufrufen. Beispielsweise wird unter den 32-Bit-Varianten NT, 2000 und XP die Funktion HeapAlloc in Kernel32.dll auf RtlAllocHeap in Ntdll.dll umgeleitet. Dieses Vorgehen wird *Forwarding* genannt.

Import Unter einem Import wird der Zugriff eines Moduls A auf eine Funktion x eines Moduls B verstanden. Um einen solchen Import erfolgreich durchführen zu können müssen also zwei Voraussetzungen gegeben sein:

1. Das Modul B muß im Suchpfad liegen, um so für den Loader des Betriebssystems auffindbar zu sein.

2. B muß die Funktion x exportieren, damit diese von anderen Modulen eingebunden werden kann.

Da Exporte grundsätzlich auf zweierlei Arten identifizierbar sind – Name und Ordinale – gibt es sowohl die Möglichkeit des Imports *by name* als

[12]Für genauere Informationen zu COFF unter Unix siehe `http://wwwstud.fh-zwickau.de/~linux/info/grundlagen/binary_formats/node2.html`, zuletzt besucht am 6.11.2004.

auch des Imports *by ordinal*. Weil die Angabe von Namen beim Export nicht verpflichtend ist lassen sich manche Funktionen ausschließlich per Ordinale importieren. Wird per Name importiert, was bei weitem die häufigste Variante ist, so wird der Name lediglich benutzt, um in der Export-Tabelle die Ordinale des referenzierten Exports nachzuschlagen, was einen leichten Geschwindigkeitsverlust bedeutet.

Import-Varianten Es existieren mehrere Möglichkeiten für den Programmierer, eine exportierte Funktion eines anderen Modules zu referenzieren:

1. Expliziter Import. Darunter versteht man eine manuelle Referenzierung des gewünschten Moduls im Code des aufrufenden Moduls. Unter Windows stehen hierzu die API-Aufrufe LoadLibrary[13] und GetProcAddress[14] zur Verfügung. Ersterer beauftragt den Loader, ein durch seinen als Parameter übergebenen Namen identifiziertes Modul in den Speicher zu laden und eine Referenz darauf zurückzugeben. Mittels GetProcAddress kann dann in diesem in den Speicher geladenen Modul-Abbild die Einsprungadresse der gewünschten Funktion ermittelt werden. Ein Anspringen dieser Adresse entspricht dann dem Funktionsaufruf. Bei explizitem Import handelt es sich also um einen Mechanismus, der vom Programmierer selbst kontrolliert und zur Laufzeit ausgeführt wird. Somit sind eventuelle Fehler vom Programmierer selbst zu behandeln.

2. Impliziter Import. Hierbei wird das auffinden des gewünschten Moduls dem Loader des Betriebssystems überlassen. Dieser stellt ebenfalls sicher, daß alle referenzierten Funktionen auch tatsächlich als Exporte vorhanden sind. Zudem trägt er Sorge, daß vom so importierten Modul benötigte (importierte) Module vorhanden und geladen sind. Importiert man beispielsweise Funktionen aus Gdi32.dll, so hat dieses Modul seinerseits Referenzen auf (d.h. Importe aus) User32.dll, Advapi32.dll, Ntdll.dll, und Kernel32.dll. Beim impliziten Import handelt es sich also um einen Mechanismus des Betriebssystems, der beim Starten des Programmes eingesetzt wird, und der vor der Ausführung des Programmcodes zum tragen kommt.

3. Verzögerter Import. Hierbei handelt es sich um eine Variante des expliziten Imports, den Microsoft Visual C++ ab Version 6.0 bietet. Der Code, der den expliziten Import zur Laufzeit durchführt wird hier automatisch generiert.

[13] Dokumentation zu LoadLibrary bei MSDN: `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/loadlibrary.asp`, zuletzt besucht am 2.11.2004.

[14] Dokumentation zu GetProcAddress bei MSDN: `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/getprocaddress.asp`, zuletzt besucht am 2.11.2004.

Insbesondere der implizite Import ist von Interesse, denn der gesamte Mechanismus dieser Art des Imports basiert auf Informationen, die *vor* der Ausführung des Programms zur Verfügung stehen. Somit ist die Frage, ob implizite Importe erfolgreich durchgeführt werden können, statisch, d.h. ohne Ausführung des betrachteten Programms, entscheidbar[15]. Da Wine die Funktionen des Windows-Betriebssystem und damit auch des Windows-Loaders nachzubilden sucht, wird unter Wine der gleiche Importmechanismus verwendet.

Gleichzeitig kann für eine statische Analyse, die bei dem hier vorgestellten Verfahren durchgeführt wird, auf die Berücksichtigung der expliziten Importe problemlos verzichtet werden. Fehler beim Import von Funktionen müssen hier ohnehin vom Programmierer abgefangen und behandelt werden.

4.3 MySQL: Eine freie SQL-Datenbank

Bei MySQL handelt es sich um eines der bekannntesten und beliebtesten FLOSS-Produkte. Die Datenbank wird vom schwedischen Unternehmen MySQL AB entwickelt und zur Verfügung gestellt. Das Unternehmen hat u.a. Niederlassungen in Schweden, den USA, Deutschland, Finnland, und Frankreich und beschäftigt über hundert Mitarbeiterinnen und Mitarbeiter. Mit über fünf Millioen Installationen ist die Datenbank eine der verbreitetsten weltweit. Im Folgenden wird mit MySQL der MySQL Server bezeichnet; die Firma vertreibt und produziert aber auch noch eine Reihe anderer Produkte mit ähnlichen Namen, u.a. die vormals als SAP DB bekannte MaxDB des deutschen Softwarehauses SAP, das diese Datenbank zusammen mit MySQL AB als Open Source entwickelt.

Entwicklungsmodell MySQL verfolgt ein zweiteiliges Entwicklungsmodell, das in gewisser Weise als typisch für Unternehmen mit FLOSS-Produkten bezeichnet werden kann. Auf der einen Seite steht hier die Community, der die Software und die Dokumentation unter Open Source- bzw. GPL-Lizenzen zur Verfügung gestellt wird. Mit relativ kurzen Release-Zyklen wird erreicht, daß die Software auf breiter Basis durch die Community getestet wird und somit wertvolles Feedback zurück in die Software fließt. Abbildung 4.3 zeigt eine schematische Darstellung[16] des Entwicklungsmodells.

Da MySQL AB weiterhin Inhaber des Urheberrechts an der Software bleibt, besteht die Möglichkeit, MySQL-Datenbanken unter kommerziellen Lizenzen für die Einbindung in nicht-FLOSS-Produkte und -Projekte zu erwerben. Ebenso verdient die Firma an Training und Consulting.

[15] Diese Tatsache macht sich z.B. das Tool „Dependency Walker“ zunutze, das Entwicklern bei der Distribution ihrer Software helfen soll, alle benötigten Module zu ermitteln und mit auszuliefern.

[16] Quelle der Abbildung: `http://www.mysql.de/company/development-cycle.png`, zuletzt besucht am 2.11.2004.

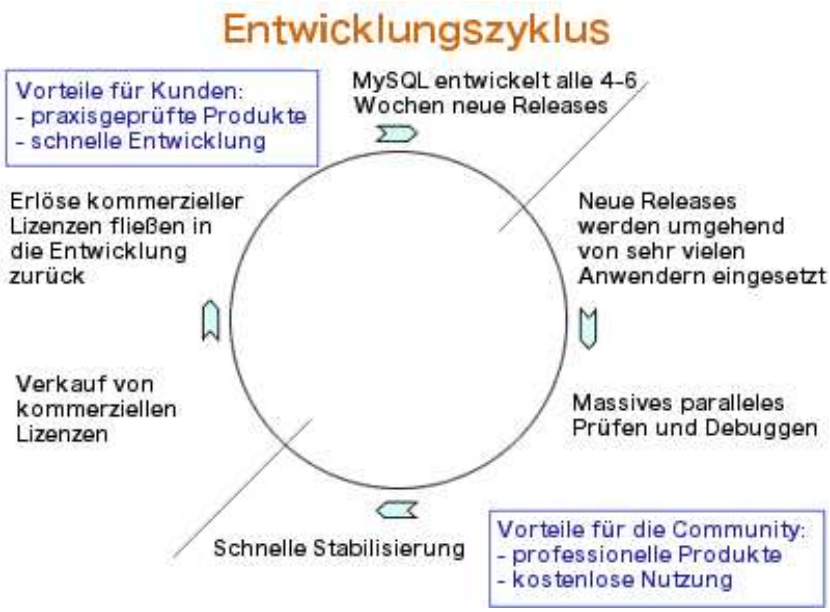


Abbildung 4.3: Graphische Darstellung des Entwicklungsmodells von MySQL.

Features MySQL ist für Linux, Windows, FreeBSD, sowie eine ganze Reihe weiterer Unix-Derivate verfügbar und kann deswegen als plattformübergreifende Datenbanklösung bezeichnet werden. Es implementiert eine umfassende Untermenge des SQL99-Sprachstandards und stellt verschiedene Storage-Mechnismen für die Daten zu Verfügung. Für mehrere Storage Engines werden außerdem Transaktionen unterstützt. Ein Rechtevergabesystem für die Zugriffe auf die Datenbank gehörn ebenso zum Leistungsumfang wie die Unterstützung von Query Caching, Mechanismen zur Replikation von Datenbanken, und SSL-gesicherte Übertragung von Abfragen und Daten zum Datenbankserver. Für eine genaue Übersicht der Funktionalitäten in MySQL sei auf die Homepage des Herstellers verwiesen[17]. Die Datenbank ist vor allem im Bereich von Webanwendungen, die keine besonders komplexe Funktionalität hinsichtlich der Abfrage (SQL-Unterstützung) benötigen, sehr beliebt.

Anbindungen Anbindungen an MySQL existieren in zahlreichen Programmiersprachen, namentlich in C, C++, Eiffel, Java, Perl, PHP, Python, Ruby, und Tcl. Es existieren ODBC-Konnektoren (MyODBC) sowie unter Java JDBC-Konnektoren. Für das in dieser Arbeit vorgestellte Programmsystem wurde auf den JDBC-Konnektor für die Datenbankanbindung un-

[17] Features von MySQL Server: `http://www.mysql.de/products/mysql/index.html`, zuletzt besucht am 2.11.2004, bzw. `http://dev.mysql.com/doc/mysql/en/Features.html`.

ter Java sowie auf das DBI-Interface für die Datenbankanbindung in Perl zurückgegriffen.

4.4 Hashfunktionen

Bei Hashfunktionen handelt es sich um nicht umkehrbare Funktionen, die eine mächtige Menge – z.B. Texte oder allgemein: Dateiinhalte – auf eine kleinere Menge abbildet. „Nicht umkehrbar“ bedeutet hierbei sowohl eine Nichtumkehrbarkeit im mathematischen Sinne als auch die Unmöglichkeit, mit vorhandenen Methoden aus dem Funktionsergebnis $f(x)$ Rückschlüsse auf x zu ziehen.

Hashfunktionen werden insbesondere in der Kryptographie häufig eingesetzt, weil sie es ermöglichen, die Integrität von Daten – beispielsweise einer Datei – zu sichern. Dazu wird ein Hashwert der Daten gebildet und digital signiert, was weit weniger aufwendig ist, als die kompletten Daten zu signieren. Ein vergleichbares Verfahren kommt beispielsweise bei dem verbreiteten Verschlüsselungsprogramm *Pretty Good Privacy* (PGP)[18] und seinem FLOSS-Pendant gnupg[19] zum Einsatz und ist auch in entsprechenden RFCs[20] so standardisiert. Zwei der am weitesten verbreiteten Hashfunktionen sind MD5 und SHA-1.

MD5 MD5 wurde vom Kryptologen Ron Rivest entwickelt, der auch an der Entwicklung des RSA-Kryptoverfahrens maßgeblich beteiligt war[21], und generiert 128 Bit lange Hashwerte. Es existieren mögliche Angriffe kryptoanalytischer Natur auf MD5, die auf der Erzeugung von Kollisionen basieren. Unter einer Kollision versteht man die Bildung des gleichen Hashwertes für unterschiedliche Daten. Weil bei den meisten modernen digitalen Signaturverfahren lediglich ein Hashwert der zu signierenden Nachricht signiert wird, wäre die Erzeugung einer Nachricht anderen Inhalts mit dem gleichen MD5-Hashwert de facto eine Möglichkeit zur Fälschung von digitalen Unterschriften. Die entdeckten Angriffe sind jedoch so geartet, daß zwar für zukünftige Entwicklungen von kryptographischer Software ein anderer Algorithmus gewählt werden sollte, jedoch für bestehende Systeme keine auch nur enfernt realistisch zu nennende Möglichkeit eines Angriffs existiert. Beispielsweise verifiziert die Linux-Distribution Gentoo[22] die Integrität herun-

[18] Homepage von PGP: `http://www.pgp.com`, zuletzt besucht am 2.11.2004.

[19] Homepage von gnupg: `http://www.gnupg.org`, zuletzt besucht am 2.11.2004.

[20] Als Einstiegspunkt sei hier die OpenPGP-RFC 2440 genannt, die unter der folgenden URL verfügbar ist: `http://www.faqs.org/rfcs/rfc2440.html`, zuletzt besucht am 2.11.2004.

[21] Wikipedia-Eintrag zu MD5: `http://de.wikipedia.org/wiki/MD5`, zuletzt besucht am 2.11.2004.

[22] Homepage von Gentoo Linux: `http://www.gentoo.org`, zuletzt besucht am 2.11.2004.

tergeladener Quelltextdateien mittels MD5, ebenso wie dies beim z.B. von SuSE oder RedHat verwendeten RPM-Paketformat der Fall ist.

SHA-1 SHA-1 wurde von der amerikanischen Standardisierungsbehörde NIST zusammen mit der National Security Agency entwickelt. SHA-1 ist der Nachfolger von SHA, in dem eine nicht näher publizierte Sicherheitslücke gefunden wurde. SHA-1-Hashes sind 160 Bit lang und damit länger als MD5-Hashes.

Kollisionswahrscheinlichkeit Geht man davon aus, daß die Hashfunktionen *perfekt* sind, d.h. für jede Eingabe x jede Ausgabe $f(x)$ gleich wahrscheinlich ist, so ergeben sich damit für die beiden Funktionen Kollisionswahrscheinlichkeiten von:

$$P_{collision}^{MD5} = \frac{1}{2^{128}} \tag{4.1}$$

$$P_{collision}^{SHA1} = \frac{1}{2^{160}} \tag{4.2}$$

Für die Implementierung des in dieser Arbeit vorgestellten Systems war es nötig, Dateien anhand ihres Namens *und* Inhalts möglichst effizient zu unterscheiden, ohne die komplette Datei speichern oder auswerten zu müssen. Zu diesem Zwecke wurden Hashfunktionen eingesetzt. Um die Wahrscheinlichkeit einer Kollision so gering wie möglich zu halten, wurde sicherheitshalber eine Konkatenation von MD5 und SHA-1-Hashwert zusammen mit dem Namen und anderen Informationen als Primärschlüssel in der Datenbank benutzt. Für Perl existieren die Module `Crypt::MD5` und `Crypt::SHA1\verb`, mittels derer die Berechnung von Hashwerten problemlos durchführbar ist.

4.5 JFreeReport

Bei JFreeReport handelt es sich um eine Bibliothek für die automatisierte Erstellung von Reports. Die Bibliothek ist Freie Software unter den Bedingungen der LGPL-Lizenz und im Internet sowohl im Quelltext als auch in kompilierter Form zu beziehen[23]. Derzeit liegt die in Java programmierte Software in der Version 0.8.4_11 vor. Zu den wichtigsten Features gehören:

- Eine Druckvorschau des gesamten erstellten Reports.
- Die Daten aus denen der Report zusammengestellt wird können aus Objekten, die das Swing-Interface TableMode implementieren entnommen werden. Das vereinfacht das Drucken und die Erstellung von Reports direkt aus GUI-Applikationen heraus erheblich.

[23] Homepage von JFreeReport: `http://www.jfree.org/jfreereport/`, zuletzt besucht am 2.11.2004.

- XML-basierte Reportdefinitionen. Schablonen für Reports werden in XML definiert und dann nach Bedarf mit den jeweils aktuellen Daten gefüllt.
- Ausgabemöglichkeit der Reports auf Drucker, Bildschirm, oder in verschiedene Export-Formate wie PDF, HTML, CVS, Excel oder Text.
- Gute Dokumentation.

Um einen Report zu erstellen muß zunächst ein XML-Template angelegt werden, die alle Elemente, die in dem zu erstellenden Dokument vorkommen, beschreibt. Darunter fallen Deckblatt, Überschriften, Absätze etc. Zur Laufzeit werden dann alle Daten, die in den Report einfließen sollen in Objekten gesammelt, die das Interface TableModel implementieren. Mittels der Methode `createReportDefinitionFromFile()` wird dann das Reportdokument erzeugt und kann mittels der Klasse PreviewDialog als Vorschau auf dem Bildschirm angezeigt oder direkt in die verschiedenen unterstützten Ausgabeformate ausgegeben werden. Eine Einführung in die Reporterstellung mittels JFreeReport ist in [Linux Magazin 2004] zu finden.

Kapitel 5

Grundsätzlicher Aufbau des Systems

Bevor die konkrete Umsetzung der eingangs gesetzten Ziele angegangen werden kann, muß veranschaulicht werden, welche einzelnen Teilprobleme zu lösen sind. Im Folgenden werden daher die Leistungen, die das System erbringen muß um den gestellten Anforderungen gerecht zu werden, erläutert. Aufgrund der Komplexität des Gesamtsystems soll der Einsatz der verschiedenen benutzten Komponenten und Technologien hier jeweils lediglich skizziert werden, bevor im nächsten Kapitel die konkrete technische Umsetzung erläutert werden kann. Der Fokus liegt somit auf einer zunächst allgemeineren und konzeptionelleren Darstellung, bei der auf manche technische Details bewußt verzichtet wird, um einen besseren Überblick zu gewährleisten.

Datenkapselung Als Basis für die Kompatibilitätsanalyse von Windows-Software müssen zunächst in geeigneter Weise Daten über die Software erhoben werden. Dies betrifft vor allem ihre Zerlegung in Module, anhand derer die Abhängigkeiten der Module untereinander ermittelt werden sollen. Es soll ein Abbild der Software dergestalt geschaffen werden, daß alle für die Modellierung der Abhängigkeitsstruktur und ihrer Analyse wichtigen Daten in der Datenbank des Systems abgelegt werden. Hierzu müssen diese Informationen nicht nur für jedes einzelne Modul erhoben, sondern auch in geeigneter Weise für die Ablage in der Datenbank konvertiert und schließlich geschrieben werden. Insbesondere muß für jedes in der Software enthaltene Modul seine zur Verfügung gestellte sowie seine von anderen Modulen benötigte Funktionalität, d.h. importierte und exportierte Funktionen, ermittelt, gekapselt, und gespeichert werden. Somit ist die Schaffung eines Datenformats zur Ablage von Informationen über Module und Funktionen einerseits, sowie die Definition eines Datentyps für Module und Funktionen andererseits nötig. Hierbei muß zwischen importierten und exportierten Funktionen ebenso unterschieden werden können wie zwischen

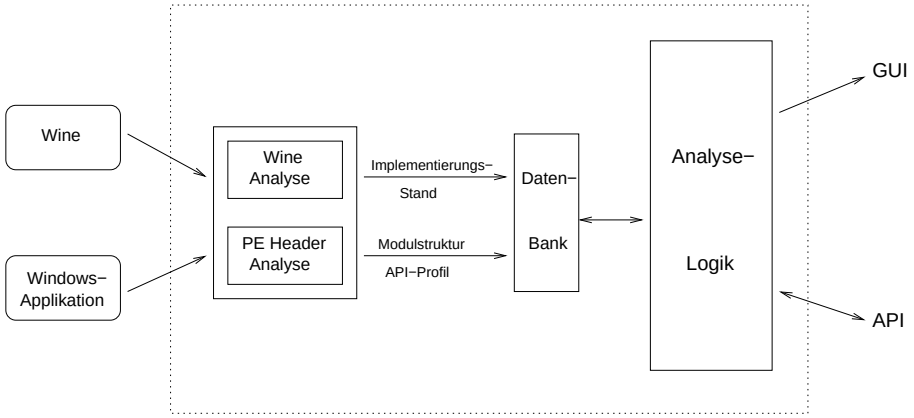


Abbildung 5.1: Schematische Darstellung des Aufbaus des Systems. Alle im gepunkteten Kasten enthaltenen Komponenten gehören zum entwickelten System. Es gewinnt statisch Daten aus Windows-Applikationen in binärer Form sowie aus dem Quelltext von Wine, und legt diese geeignet konvertiert und strukturiert in der Datenbank ab. Die Analyse-Logik ermittelt dann die Problemstellen, die bei der Ausführung der Software unter Linux/Wine behindern können, und stellt diese graphisch und über eine API zur Verarbeitung zur Verfügung.

namensgleichen Modulen unterschiedlichen Inhalts sowie vorhandenen und fehlenden Funktionen und Modulen. Des weiteren wird ein Datenformat für Softwareabbilder benötigt.

Für den Abgleich mit dem Implementierungsstand verschiedener Wine-Versionen muß zusätzlich ein Datenformat für Strukturabbilder von Wine-Versionen geschaffen werden. Dabei muß für jedes Modul und jede Funktion der Implementierungsstand möglichst granular in der Datenbank ablegbar sein, um die Qualität der Analyseergebnisse zu optimieren. Es ist zudem zu berücksichtigen, daß Wine zwar die gleiche Funktionalität herzustellen versucht wie die Windows-API, er jedoch aus technischer Sicht sowohl eine andere Struktur als auch ein anderes Binärformat besitzt als Windows und seine Module. Daher kann die Datenerhebung nicht auf die gleiche Weise erfolgen wie bei Windows-Software. Zudem sind äquivalente Informationen über Module anders codiert als bei PE-Modulen. Die Berücksichtigung zusätzlicher Informationen ist möglicherweise für die Verbesserung der Qualität der Analyse von Nutzen, wie z.B. mögliche Überladung von Funktionen oder die Typen ihrer Rückgabewerte. Des weiteren muß die Information über Wine in einer Weise abgelegt werden, die trotz der unterschiedlichen Struktur von Wine und Windows-Software einen Vergleich bzw. den Abgleich miteinander gestattet.

Damit eine Ablage von Analyseergebnissen in der Datenbank möglich wird, müssen die benutzten Datenstrukturen und ihre Ablageformate in einer Weise strukturiert werden, die jederzeit eine verlustfreie Konvertierung in die eine oder andere Richtung erlaubt. Um den Aufwand des Auslesens aus

der Datenbank und des Einfügen in die entsprechenden Datentypen im Speicher – also die „Instanziierung“ eines Softwareabbildes – gering zu halten, liegt es nahe, Analyseergebnisse in der im Arbeitsspeicher liegenden Form in der Datenbank abzulegen, anstatt ein eigenes Datenformat für diesen komplexen und mächtigen Datentyp zu definieren. Um sowohl Datenbankzugriffe als auch Operationen auf den Datenstrukturen (und damit die Analyse selbst) ressourcensparend durchführen zu können ist es erforderlich, die Informationen platzsparend zu codieren. Eine Kompatibilitätsanalyse von Windows-Software erfordert je nach Software-Größe die Berücksichtigung von etwa 20 000 bis 100 000 Modulabhängigkeiten, was diese Notwendigkeit noch unterstreichen mag.

Loader-Nachbildung Da bei einer Verwendung von Wine und Linux die auszuführende Windows-Software unverändert bleiben soll, muß von Wine genau die Umgebung bzw. die Teilmenge der Umgebung hergestellt werden, die die Software benötigt. Unter Windows werden Modulabhängigkeiten vom Loader des Betriebssystems aufgelöst bzw. fehlende Module bei Aufruf des Programms erkannt. Wine bildet daher die Arbeitsweise des Windows-Loaders nach[1] und kann somit nicht nur als Ersatz für Betriebssystemkomponenten sondern auch als Loader für PE-Module dienen.

Um ungelöste Abhängigkeiten und damit Probleme bei einer Weiterverwendung unter Wine zu erkennen, muß also das Verhalten des Windows-Loaders möglichst detailgetreu nachgebildet werden. Diese Nachbildung muß auf den eingeführten Datentypen operieren und dabei Informationen sammeln, die eine genaue Auswertung von Modulabhängigkeiten hinsichtlich Art und Schwere ermöglichen. Hierbei muß trotz der erwünschten Abstraktion eine detailgetreue Nachbildung von realen Software- bzw. Wine-Installationen erfolgen. Da das Verhalten des Windows-Loaders abhängig von verschiedenen Variablen wie z.B. Suchpfaden modifiziert werden kann, muß eine äquivalente Möglichkeit auch bei der Simulation gegeben sein. Gleichzeitig ist das Verhalten des Loaders wichtiger Bestandteil der Konfiguration von Wine, so daß eine Abbildung dieser Konfigurationsmöglichkeiten in der Art erfolgen muß, daß sich eine Konfiguration von Wine aus den Analyseergebnissen und der Parametrisierung der Loader-Nachbildung des Systems generieren läßt. Diese Loader-Konfigurationen müssen ebenfalls abspeicherbar sein. Wegen dem bereits angesprochenen Umfang der zu berücksichtigenden Information ist bei der Implementierung dieses zentralen Teils des Systems die Ausführungsgeschwindigkeit sehr wichtig.

[1] Diese ist bei MSDN unter der URL `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vccore/html/_core_the_search_path_used_by_windows_to_locate_a_dll.asp` genauer erläutert (letzter Besuch am 1.8.2004).

Modulstruktur Um die Daten auf denen später mit Hilfe der dafür definierten Strukturen operiert werden soll zu erheben wird ein Mechanismus benötigt, der jedes Modul einer Windows-Software in Augenschein nimmt. Die wichtigste Information die dabei erhoben werden muß ist die Aufstellung der exportierten, d.h. anderen Modulen zur Verfügung gestellten Funktionen, die Namen der vom analysierten Modul benötigten Module, sowie die Funktionen dieser benötigten Module, auf die Zugriff erforderlich ist. Es sind noch eine Reihe von weiteren Daten abzulegen, da z.B. der Loader Module in bestimmten Suchpfaden aufzufinden versucht, und somit der Pfad im Dateisystem, in dem sich ein Modul befindet, für die Analyse relevant ist. Insgesamt muß mit den gespeicherten Daten eine Analyse der Software jederzeit, d.h. auch mit früheren oder späteren Wine-Versionen möglich sein, ohne daß weiterhin die Module der Software physikalisch zur Verfügung stehen. Die Abhängigkeitsstruktur der Module untereinander muß vollständig nachvollziehbar sein, um so fehlende Module und Funktionen auffinden zu können. Es muß ermittelt werden, ob diese Fehlstellen von Wine ausgefüllt werden können. Dazu ist eine Übersicht über die Modulstruktur von Wine und die von ihr zur Verfügung gestellten Funktionalitäten nötig, um diese mit dem Anforderungsprofil der Software abzugleichen.

Wine-Implementierungsstand Neben der Ermittlung der Modulstruktur von Wine und der von ihr zur Verfügung gestellten Funktionen ist es für das Auffinden von Problemstellen, die sich durch die unvollständige Implementierung der Windows-API durch Wine ergeben essentiell, den Wine-Implementierungsstand möglichst granular zu ermitteln und in der Datenbank zur späteren Analyse abzulegen. Diese Ermittlung erfolgt in mehreren Stufen. Zunächst werden die von den Wine-Entwicklern benutzten sog. spec-Files bearbeitet, um aus ihnen Informationen über Parameterliste und Aufrufkonvention der Funktionen zu erhalten. Zusätzlich sind in diesen Dateien auch Funktionen ohne Implementierung markiert, so daß völlig fehlende Funktionalität hier bereits erkannt werden kann. Gleichzeitig ist in ihnen auch die Information abgelegt, ob eine Funktion zwar vorhanden ist (kein „stub“), aber dies sich lediglich in einem Forward, d.h. einer Weiterleitung, auf eine andere Funktion manifestiert. In diesem Fall muß der Implementierungsstand des Forward-Zieles ermittelt und eingesetzt werden.

Da Wine im Quelltext vorliegt und als Freie Software auch zukünftig vorliegen wird ist es naheliegend, die aus den spec-Files gewonnenen Informationen über Entwicklungsstand durch zusätzliche Betrachtung des Quelltextes zu verfeinern. Die strengen Code-Richtlinien für Wine erweisen sich hierbei als hilfreich, da Unvollständigkeiten in der Implementierung einer Funktion nur auf bestimmte Arten an die Debug-Ausgabe übergeben werden. Somit können diese Informationen als gute Anhaltspunkte für den Reifegrad der Implementierung einer Funktion gelten. Zur Ermittlung dieser

Informationen wird der Quelltext für je ein Modul nach einer Vorverarbeitung (Entfernung von Kommentaren u.Ä.) in Paare von Funktionsname und Funktionsrumpf zerlegt. In den Funktionsrümpfen wird jeweils nach Aufrufen gesucht, die fehlende Funktionalität nach außen kommunizieren (Debug-Ausgabe) und diese Zusatzinformation für jede Funktion ebenfalls in der Datenbank abgelegt.

Eine weitere Verfeinerung der Quelltextanalyse ist möglich, wobei gleichzeitig die hierdurch zusätzlich gewinnbare Qualität mit dem zu erbringenden Aufwand in immer ungünstigerem Verhältnis steht. Für die Implementierung des Verfahrens wurden daher die beschriebenen Maßnahmen für ausreichend befunden. Diese Einschätzung hat sich durch umfassende Tests in der Praxis bestätigt. Gleichzeitig wurde darauf geachtet, zusätzliche Informationen durch weitere Verfeinerung der Quelltextanalyse in einer möglichen Weiterentwicklung des Systems berücksichtigen zu können.

Kapitel 6

Design und Umsetzung

Nachdem im vorigen Kapitel eine Übersicht über die Leistungsmerkmale und den Umfang des Systems gegeben wurde, soll nun die konkrete Implementierung erläutert werden. Dabei werden sowohl die Entwurfsentscheidungen diskutiert als auch der Aufbau der Software und der Datenbank erläutert. Zunächst soll das Augenmerk der Datenbank und ihrer Struktur gelten, da diese einen zentralen Bestandteil des Gesamtsystems darstellt. Die PE-Header-Analyse als Basis der Datenerhebung für die spätere Analyse wird anschließend erläutert, ebenso die mit ihr methodisch verwandte Ermittlung des Wine-Implementierungsstands. Schließlich wird die Implementierung des Analysesystems mit der Loader-Simulation beleuchtet.

6.1 Datenbankstruktur

Wegen der nicht unerheblichen Menge an Daten und dem Ziel, wenn möglich FLOSS-Komponenten einzusetzen wurden für als Datenbank vornehmlich die Open-Source-Datenbank PostgreSQL und das freie MySQL in Erwägung gezogen. PostgreSQL zeichnet sich gegenüber MySQL durch vollständigere Implementierung des SQL-Sprachstandards und die seit längerem vorhandene Unterstützung von Transaktionen aus. Da dieses Produkt jedoch hinsichtlich des ebenfalls in alle Entwurfsentscheidungen einbezogenen Faktors Performanz durch mangelnde Ausnutzung von Parallelität sowie allgemein geringere Ausführungsgeschwindigkeit nicht vorteilhaft abschneidet, wurde auf das freie MySQL in der Version 4.0 zurückgegriffen. Dieses verfügt über eine hohe Geschwindigkeit bei Ein- und Ausgabe von Daten und die Möglichkeit zur parallelen Verarbeitung (*Threading*). Da abzusehen war, daß der SQL-Sprachumfang nur in geringem Maße ausgeschöpft werden mußte und die wichtigste Funktion der Datenbank die performante Ablage bzw. das Abrufen von Daten sein würde, war dies ein klarer Vorteil für MySQL. Zudem verfügt es über eine sehr aktive und große Community von Anwendern und Entwicklern, was sich nicht zuletzt in der ausgezeichneten und umfangrei-

chen Dokumentation sowie in zahlreichen GUI-basierten Administrationstools niederschlägt.

Aufgabe der Datenbank ist es, Daten über Module und ihre Importe und Exporte so zu speichern, daß alle wesentlichen Informationen, die zur Analyse der Abhängigkeitsstrukturen nötig sind, vorliegen. Dabei soll die Möglichkeit bestehen, Module gruppiert abzuspeichern (*Projekte*), um einzelne Softwareprodukte analysieren zu können. Gleichzeitig soll eine redundante Speicherung von Informationen vermieden werden[1]. Die Datenbank soll fertige Analyseergebnisse nach Projekten gruppiert aufnehmen können. Instanzen von Loader-Objekten, die mit den Strukturinformationen jeweils einer anderen Wine-Version vorinitialisiert sind, müssen ebenfalls abgelegt und abgerufen werden können. Dies dient der Verringerung von Fixkosten von Analysen und damit dem Gewinn von Performanz. Da Loader mittels verschiedener Parameter konfigurierbar sind, nimmt die Datenbank ebenfalls für jedes Projekt eventuelle Konfigurationen des Loaders auf. Für die Einhaltung von Konsistenzbedingungen soll von den Features der Datenbank Gebrauch gemacht werden. Da bei Analysen von Software stets weitere, auch nicht im engeren Sinne technische Informationen anfallen, die dennoch für ein späteres Nachvollziehen oder eine Wiederholung der Analyse unter modifizierten Bedingungen von Nutzen sein können, müssen diese Meta-Daten ebenfalls an geeigneter Stelle abgelegt werden. Hierzu zählen beispielsweise Installationsprotokolle oder eventuelle Anforderungen eines möglichen Kunden, der eine Analyse in Auftrag gegeben hat. Zur Erfüllung der genannten Zwecke wurden im Datenbanksystem sieben Tabellen angelegt. Die folgenden Abschnitte geben einen Überblick über die Funktionsweisen und die Struktur der einzelnen Tabellen. Die vollständigen Definitionen der jeweiligen Tabellen sind im Anhang A nachzuschlagen.

Modul-Tabelle In der Modul-Tabelle sind alle Informationen abgelegt, die eine Moduldatei betreffen. Dazu zählt insbesondere ihr Name im Dateisystem und ihre Dateigröße. Die eindeutige Identifizierung (*Primary Key*) eines Moduls erfolgt über seinen Namen, einen Hashwert, seine Zugehörigkeit zu einer bestimmten Windows- oder Wine-Version, und einem Typfeld, das angibt, ob es sich um ein Wine-Modul handelt. Der Hashwert wird zur Minimalisierung des Risikos von Kollisionen aus Konkatenation von MD5- und SHA1-Hashwert der Moduldatei gebildet. Für Wine-Module wird der Hashwert des zugehörigen spec-Files gespeichert. Da sich aber ein Großteil der spec-Files zwischen zwei Releases nicht ändert, wurde zudem die Versionsnummer von Wine als Unterscheidungsmerkmal einbezogen, ebenso wie das Typfeld, mit dem sich Wine und Nicht-Wine-Module unterscheiden lassen. Da es auch möglich sein soll, Windows-Module verschie-

[1] Einige Module kommen in beinahe jeder Windows-Software vor und sollen dennoch nur einmal abgespeichert werden müssen.

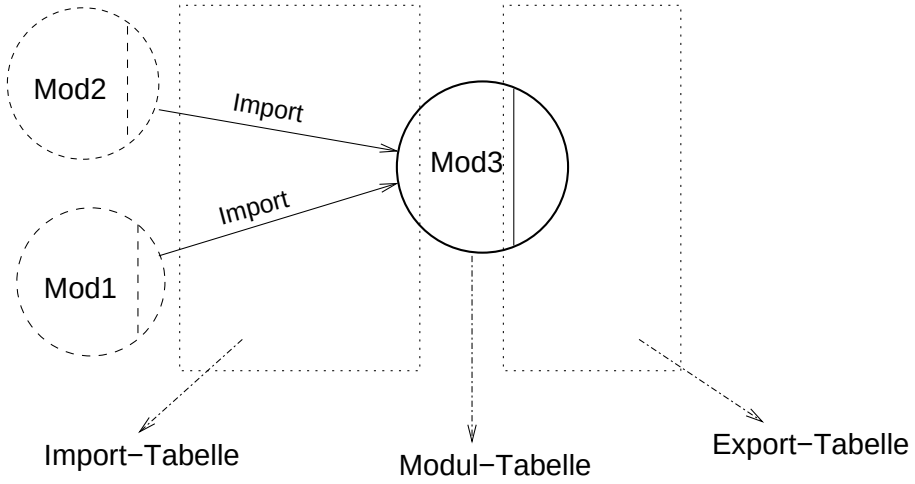


Abbildung 6.1: Schematische Darstellung des Zwecks von Modul-, Import-, und Export-Tabelle. Für ein Modul Mod3 enthält die Import-Tabelle der Datenbank Informationen über die Importe von Mod3, während die Export-Tabelle die Exportschnittstelle abspeichert. Die Modultabelle enthält Informationen über die Moduldatei selbst.

dener Windows-Versionen in der Datenbank abzulegen, und sich diese aber möglicherweise weder in Namen noch in Inhalt (Hash-Wert) zwischen zwei Windows-Versionen ändern, wurde auch für Windows-Module ein entsprechendes Versionsfeld eingeführt. Die Identifikation über den Hashwert ist zwingend notwendig, denn obwohl er unwahrscheinlich erscheinen mag, tritt der Fall von namens- und größengleichen Modulen mit verschiedenem Inhalt in der Praxis auf, was durch die gewählte Art der Modulidentifikation in der Datenbank abgebildet werden kann.

Export-Tabelle Vergleichbar den Einträgen in der Export-Tabelle von PE-Dateien speichert die Export-Tabelle im System Informationen über exportierte Funktionen der einzelnen Module. Ein Export ist eindeutig identifizierbar über seinen Funktionsnamen, seine Ordinalzahl, den Hashwert des Modules zu dem er gehört, dem Namen der Datei, in der er zu finden ist, und der Wine-Version, zu der er gehört. Ist ein Modul in einer Sprache geschrieben, die andere Sichtbarkeitsregeln besitzt als C – dies ist insbesondere bei allen blockstrukturierten Sprachen der Fall – so müssen die Namensräume zwecks Abbildung auf C-Sichtbarkeitsregeln in die Funktionsnamen kodiert werden. Ist zusätzlich noch Überladung möglich, müssen auch Funktionparameter und ihre Typen mit kodiert werden. Dies ist insbesondere bei C++ bzw. dem unter Windows verbreitetem Visual C++ der Fall. Die Kodierung folgt komplizierten Regeln[2], und ist zwar semantisch eindeutig, jedoch

[2]Siehe hierzu auch die Website von Dan Kegel, `http://www.kegel.com/mangle.html` (zuletzt besucht am 4.8.2004).

syntaktisch in mehreren Arten durchführbar[3]. Daher wird für die Identifizierung des Funktionsnamens die kodierte Version gespeichert und die von Menschen besser zu lesende dekodierte in einem Zusatzfeld abgelegt. Die Speicherung des Modulnamens dient der Vermeidung einer Abfrage in der Modultabelle, wenn der Modulname zu einem Export benötigt wird, sowie zusätzlicher Redundanz zur Identifizierung mittels Hashwert. Eine Zuordnung zum Modul bzw. seinem Eintrag in der Modultabelle ist deswegen über einen Hashwert möglich, weil Exporte (und Importe) Inhalte der Datei darstellen, so daß bei gleich großer und gleichnamiger Datei andere Exporte anderen Inhalt und somit einen abweichenden Hashwert implizieren.

Import-Tabelle Das Äquivalent bezüglich Importe ist die Import-Tabelle der Datenbank, die ähnlich zur Import-Tabelle in PE-Dateien und beinahe analog zur Export-Tabelle Informationen über von anderen Modulen benötigte, d.h. importierte, Funktionen speichert. Auch hier gilt wieder Gleiches für die Speicherung von kodierten Funktionsnamen: Zur Identifizierung wird der kodierte Name wegen seiner Eindeutigkeit herangezogen und gleichzeitig aber der dekodierte als menschenlesbare Zusatzinformation abgelegt. Da bei Importen grundsätzlich nur der Name des Moduls bekannt ist, aus dem die API benötigt wird, wird dieser abgespeichert und dient auch der eindeutigen Identifizierung eines Imports. Die Zuordnung zu einem Modul erfolgt im Rahmen der Analyse in Abhängigkeit von Pfad der importierenden Datei und Einstellungen des Loaders (Suchpfade etc.). Weil entgegen aller Logik der Fall von einem Modul mehrfach importierter (gleicher) APIs in der Praxis auftritt und im System auch abbildbar sein soll, wurde als weiteres Unterscheidungsmerkmal die Einsprungstelle für den Import benutzt. Diese ist aus technischer Sicht die Codestelle, an die bei Aufruf einer importierten API gesprungen wird, und kann somit zusätzlich zum Namen und den weiteren, bereits genannten Merkmalen als Identifikator benutzt werden.

Projekt-Tabelle Um die Möglichkeit zur Gruppierung von Modulinformationen zu schaffen, wurde die Projekt-Tabelle im System angelegt. In ihr sind Informationen zu den Modulen, die in einem Projekt – d.h. einer zu analysierenden Software – gehören, abgelegt. Neben der Modulidentifikation über den Hashwert gehören auch Informationen dazu, die sich auch für inhaltsgleiche Module in unterschiedlichen Projekten unterscheiden, wie z.B. der Pfad im Dateisystem, in dem das Modul zu finden war. Wegen der in umfangleichen Softwareprodukten teilweise enorm langen (im Grundsatz sogar beliebig langen) Pfadinformation wurde von dieser lediglich ein Has-

[3]Beispielsweise läßt sich – in Anlehnung an ein Beispiel auf o.g. URL – der kodierte Funktionsname '?FA10_i_i@@YAHQAH@Z' dekodieren als 'int FA10_i_i(int a[10])' , aber auch als 'int FA10_i_i(int b[10])' etc.

hwert zur eindeutigen Identifizierung in den Primärschlüssel dieser Tabelle einbezogen und der komplette Pfad zusätzlich abgelegt. Neben diesen Informationen identifiziert der Projektname das Modul bezüglich eines Projektes eindeutig.

Projekt-Meta-Tabelle Die Gruppierung von Modulinformationen zu einem Projekt bedingt die Notwendigkeit, zu dieser neu geschaffenen Entität Meta-Informationen abspeichern zu können. Diesen Zweck erfüllt die Projekt-Meta-Tabelle. Eindeutig identifiziert durch den Projektnamen lassen sich hier in jedem Datensatz projektspezifische Zusatzinformationen ablegen. Für die Parametrisierung des Loaders ist der Pfad des Windows-Systemverzeichnisses (auch des System32-Verzeichnis für Windows NT) unverzichtbar, weil diese in den Standard-Suchpfad eingebunden werden. Diese Pfade werden folglich in der Projekt-Meta-Tabelle gespeichert. Da Analyseergebnisse für ein Projekt auch für verschiedene Wine-Versionen abgespeichert und abgerufen werden sollen, existiert hierfür ebenfalls Platz in der Tabelle. Die Möglichkeit für die Ablage eines Installations- bzw. Testprotokolls ist ebenso gegeben wie für weitergehende Informationen zur Software selbst. Dazu zählen beispielsweise Versionsnummer, Lizenzinformationen, benötigte Zusatzkomponenten, und einer textuelle Beschreibung der Software und ihrer Funktionen.

Wine-Meta-Tabelle Für jede Analyse von Windows-Software bezüglich ihrer Lauffähigkeit unter einer bestimmten Wine-Version wird ein Loader benötigt, dem die Modulstruktur dieser Wine-Version kennt. Wegen des enormen Umfangs von Wine – etwa 20 000 Funktionen – ist der Aufwand des Auslesens all dieser Informationen ein signifikanter Punkt in der Performanz des Gesamtsystems. Weil dieser Aufwand für jede Analyse anfällt, wurde die Möglichkeit geschaffen, vorinitialisierte Loader-Instanzen für jede Wine-Version als Meta-Information in der Datenbank abzulegen. Diesem Zweck dient die Wine-Meta-Tabelle.

Konfigurationen-Tabelle Der Loader von Windows ist ebenso wie der von Wine auf verschiedene Arten parametrisierbar, um sein Verhalten zu beeinflussen. Für die im System implementierte Loader-Simulation gilt dies folglich gleichermaßen. Da es ein Ziel des Systems ist, eine möglichst gute Wine-Konfiguration z.B. durch geschicktes Weiterbenutzen von Windows-DLLs mit APIs die Wine noch nicht implementiert zu erstellen, und für spätere Testläufe eines Projekts auch mehrere dieser Konfigurationen von Nutzen sein könnten, ist es sinnvoll, diese Konfigurationen auch dauerhaft in der Datenbank ablegen zu können. In der Konfigurationen-Tabelle geschieht genau dies, wobei eine Konfiguration durch ihren Namen und den Namen des Projektes, zu dem sie gehört, eindeutig identifiziert ist. Durch

eine Konfiguration können Suchpfade gesetzt und Module aus dem Projekt logisch entfernt bzw. hinzugefügt werden, was in textuell repräsentierter Form abgespeichert wird.

6.2 PE Header Analyse

Wie bereits angedeutet ist es für den gewählten Ansatz der statischen Analyse nötig, gegebene Windows-Software zunächst in ihre Modulstruktur zerlegen zu können, um anschließend die Abhängigkeiten dieser Module untereinander zu bestimmen. Kennt man somit die Anforderungen eines jeden Moduls an die anderen Module der Software sowie die Funktionalitäten, die die anderen Module der Software zur Verfügung zu stellen in der Lage sind, so ergibt sich automatisch ein Abbild der Abhängigkeiten und Zusammenhänge der Module untereinander. Gleichzeitig lassen sich so auch Module und von ihnen zur Verfügung gestellte Funktionen ermitteln, die in der betrachteten Modulmenge nicht vorkommen und somit „von außen“, das heißt hier: vom Betriebssystem zur Verfügung gestellt werden müssen. Die Sammelung dieser Informationen, zusammen mit ihrer geeigneten Speicherung in der Datenbank des Systemes wird in dieser Arbeit als *PE Header Analyse* bezeichnet.

6.2.1 Ansatz

Aus den in Abschnitt 4.2 erläuterten Definitionen des Standards für Windows Executables – sog. *Portable Executables*, kurz *PE* – ist ersichtlich, wo die Informationen über die Ahängigkeiten eines Moduls abgespeichert sind. Jede PE enthält in ihrem Header in der Import-Tabelle Informationen über andere benötigte Module. Dort wird für jedes benötigte Modul eine eindeutige Identifizierung derjenigen Funktionen hinterlegt, die für die Ausführung des Moduls benötigt werden. Die Identifizierung kann laut Standard über einen Namen (Zeichenkette) oder eine Zahl (Ordinal) erfolgen. Eine solchermaßen importierte Funktion aus einem anderen Modul wird im Folgenden als Import bezeichnet.

Komplementär zu der Import-Tabelle existiert in den PE ebenfalls eine Export-Tabelle. In dieser sind Informationen darüber abgelegt, welche Funktionen das Modul anderen zur Verfügung stellt, und wie sich diese Funktionen jeweils eindeutig identifizieren lassen. Hier ist ebenfalls eine Identifizierung sowohl über Name als auch über eine Ordinalzahl möglich. Diese exportieren Funktionen werden analog zu den Importen im Folgenden als Exporte bezeichnet. Eine Auflösung der Abhängigkeiten eines Moduls geschieht unter Windows zur Laufzeit duch den Loader, der beginnend bei dem aufgerufenen Programmodul (Programm.EXE) jeweils für jedes importierte Modul die Importe sicherzustellen versucht und an geeigneter Stelle in den Speicher lädt. Fehlende Modulabhängigkeiten sind somit Fehler, die

Abbildung 6.2: Windows-Fehlermeldung beim fehlgeschlagenen Versuch des Loaders, eine Modulabhängigkeit aufzulösen. Ein benötigtes, d.h. importiertes Modul (shlwapi.dll) konnte nicht geladen werden.

zur Laufzeit auftreten. Die typische Fehlermeldung hierzu, die wohl beinahe jeder Windows-Benutzer bereits einmal gesehen hat, ist in Abbildung 6.2 zu sehen.

Die Ermittlung der Abhängigkeitsstruktur einer Modulmenge geschieht damit folgendermaßen: Zumächst wird eine Liste aller Module einer Software erstellt. Über diese Liste wird nun iteriert, wobei in jedem Iterationsschritt zunächst die zur Verfügung gestellte Funktionalität – die Exporte – ausgelesen wird. Diese Informationen werden geeignet codiert in der entsprechenden Tabelle in der Datenbank (der Export-Tabelle) abgelegt. Das Gleiche passiert mit den Importen, die in der Import-Tabelle der Datenbank abgelegt werden. Informationen über das jeweilige Modul werden in der Modul-Tabelle des Systems eingepflegt. Damit lassen sich bereits analysierte Module überspringen, sowie eine Zuordnung von Exporten und Importen zu ihrem jeweiligen Modul vornehmen. Um die gleiche Modulmenge (= Software) zu einem späteren Zeitpunkt wieder analysieren zu konnen, wird die Liste der Module in der Projekt-Tabelle der Datenbank abgelegt. In Pseudocode dargestellt wird also wie folgt vorgegangen:

```
1: Erstelle Modulliste ML
2: for m in ML do
3:    if m nicht in Datenbank.Modultabelle then
4:       Lese Importe(m)
5:       Lese Exporte(m)
6:       Datenbank.Exporttabelle ← Exporte(m)
7:       Datenbank.Importtabelle ← Importe(m)
8:       Datenbank.Modultabelle ← Modulinformationen(m)
9:    end if
10:   Datenbank.Projekttabelle ← Modulzugehörigkeit zu Projekt
11: end for
```

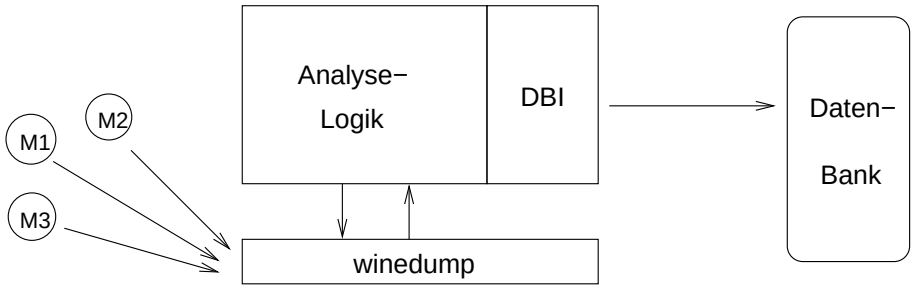


Abbildung 6.3: Schematische Darstellung der Implementierung der PE Header Analyse. Die Module M1, M2 und M3 werden von der Analyse-Logik mittels winedump hinsichtlich ihrer Importe und Exporte durchleuchtet, und diese Daten nach geeigneter Konversion in der Datenbank des Systems abgelegt. Hierbei kommt die Datenbankschnittstelle DBI zum Einsatz.

6.2.2 Implementierung

Das Auslesen dieser Informationen ist offensichtlich für jede Software lediglich genau ein Mal nötig. Die Zahl der zu betrachtenden Module und ihrer Importe und Exporte ist oftmals zwar groß – bis zu mehreren 10 000 –, jedoch wurde bei der Implementierung dieses Teils des Systems getreu dem KISS-Prinzip[4] auf eine einfachere Implementierung mehr Wert gelegt als auf erhöhte Performanz. Für die Erstellung der Modulliste und die Iteration wurde somit die Skriptsprache Perl verwendet. Diese verfügt über ausgefeilte Mechanismen zur Stringbehandlung und unterstützt sehr umfassend reguläre Ausdrücke, was sich hier als sehr hilfreich darstellte. Perl ist zudem für mehrere Plattformen verfügbar, was dem eingangs gesetzten Ziel, die Implementierung möglichst plattformunabhängig zu halten, entgegenkommt. Mittels der in Perl vorhandenen Datenbankschnittstelle DBI war auch eine einfache Anbindung an die Datenbank möglich, wobei durch die Abstraktheit der Schnittstelle weiterhin die Möglichkeit gewahrt bleibt, grundsätzlich beliebige SQL-Datenbanken zu verwenden. Für das Parsing der PE Header wurde das kommandozeilenbasierte Programm *winedump* aus dem Wine-Projekt eingebunden. *winedump* verfügt insbesondere über die Fähigkeit, PE Header textuell darzustellen. Diese Darstellung wird durch das Perl-Programm geparst und die so gewonnene Informationen weiter verarbeitet. Abbildung 6.3 zeigt die Implementierung schematisch.

6.3 Ermittlung des Wine-Implementierungsstands

Während sich mittels der PE Header Analyse aus der Windows-Software die benötigten Informationen für eine statische Analyse extrahieren lassen, ist die Sachlage bei Wine anders. Bei den in kompilierter Form vorliegenden

[4]KISS = Keep it small and simple.

Softwaremodulen von Windows-Programme kann ohne Weiteres davon ausgegangen werden, daß die Exporte Funktionen darstellen, die hinsichtlich ihres Implementierungsstandes unkritisch sind. Wine jedoch zielt auf eine Nachimplementierung der Windows-API unter Unix ab und ist deswegen prinzipbedingt der durch Microsoft vorgegebenen Entwicklung hinterher. Folglich existieren in Wine sehr unterschiedliche Implementierungsstände der Funktionsmenge der Windows-API. Der Implementierungsstand einer jeden Funktion kann grob unterschieden werden in:

1. nicht vorhanden
2. leerer Funktionsrumpf
3. Funktionsrumpf, der nur Rücksprung (`return false`) o.Ä. enthält, aber keine sonstige Funktionalität
4. ansatzweise Implementierung der Funktionalität
5. Implementierung mit bekannten Fehlern oder Lücken
6. vollständige Implementierung

Da es sich bei Wine um ein außerordentlich umfangreiches Projekt handelt, das bei der Anfertigung dieser Arbeit bereits über eine Million Zeilen Code und etwa 20 000 Funktionen beinhaltet, gelten für die Codierung dieser Informationen gewisse Richtlinien. Dies ist insbesondere deswegen von Bedeutung, weil an einem solchen Open Source Projekt zahlreiche Entwickler rund um den Globus beteiligt sind, die sich häufig nie persönlich kennenlernen. Der Entwicklerkreis ist zudem einer gewissen Fluktuation unterworfen.

Den Umstand, daß für die Analyse des Implementierungsstands von Wine grundsätzlich stets der Quelltext des Projekts bzw. der zu betrachtenden Wine-Version zur Verfügung steht, bietet sich als Lösungsansatz geradezu an. Gleichzeitig sind in sog. spec-Files für jedes Wine-Modul, das eine Windows-DLL nachbildet, gewisse Grobinformationen über die von dem Modul zur Verfügung gestellten Funktionen enthalten. Es ergeben sich also zwei Quellen für die Ermittlung des Implementierungsstandes

1. spec-Files
2. Sourcecode des Wine-Projektes

Ein weiterer Aspekt der Berücksichtigung finden muß ist die Verwendung von sog. Forwards. Im Standard des Portable Executable Formates für Windows-Binaries ist die Möglichkeit vorgesehen, daß ein Modul eine exportierte Funktion auf eine andere Funktion eines anderen Moduls weiterleitet, d.h. „forwardet“. Ein Aufruf der ursprünglichen Funktion wird dann umgewandelt in einen Aufruf der Zielfunktion dieses Forwards.

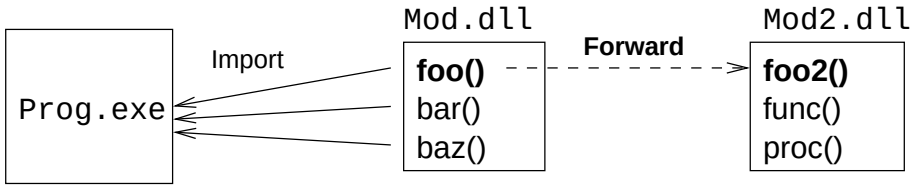


Abbildung 6.4: Schematische Darstellung eines Forwards. Das Programm Prog.exe importiert Funktionen eines Moduls Mod.dll. Die von diesem Modul exportierte Funktion foo() wird auf die Funktion foo2() des Moduls Mod2.dll geforwardet.

Für Wine-Module existiert dieses Stilmittel grundsätzlich auch. Ebenfalls üblich ist es hier, sog. *Handles* zu verwenden. Dabei handelt es sich um eine Art Forwards, die innerhalb des gleichen Moduls vonstatten geht. Daher muß bei der Ermittlung des Implementierungsstands einer API in Wine immer berücksichtigt werden, ob exportierte Funktionen auf andere „weitergeleitet“ werden. In diesem Fall wäre der Implementierungsstand des Zieles des Forwards zu betrachten.

6.3.1 Analyse von spec-Files

In Wine selbst sind neben den nachimplementierten Funktionen die Windows-API auch einige interne Funktionen enthalten, deren Implementierungsstand hier nicht berücksichtigt werden soll. Daher ist es zunächst erforderlich, Funktionen der Windows-API – also solche, die im Sinne dieses Textes als Exporte gelten können – aufzufinden, um diese dann in späteren Analyseschritten genauer zu untersuchen.

Die Exporte einer in Wine nachimplementierten Windows-DLL sind in einer für jede solche „Wine-DLL“ vorhandenen spec-Datei codiert. Das Format für diese Dateien ist textuell, aber nichtsdestotrotz eher kryptisch. Diesbezüglich findet sich Dokumentation auf der Wine-Homepage[5]. Tabelle 6.1 enthält einige Beispiele von Einträgen in solchen Dateien, zusammen mit ihrer Bedeutung. Für jeden Export einer Wine-DLL enthält das zugehörige spec-File Informationen über seinen Namen, die Art des Exportes (by name oder by ordinal), ggf. die Signatur der Funktion, sowie ggf. das Ziel, auf das diese Funktion geforwardet wird. Ist die Funktion lediglich eine mit leerem Rumpf (stub), so wird dies auch im spec-File vermerkt.

Damit sind alle Informationen vorhanden, die benötigt werden, um eine Liste der Exporte zu erstellen, deren Implementierungsstand später genauer beleuchtet werden sollte. Ebenso können auf dieser Ebene bereits Forwards erkannt und später weiter verfolgt werden. Die Codierung des Implemen-

[5]Die spec-Files sind im Zusammenhang mit der winelib dokumentiert. Daher bezieht sich die vorhandene Dokumentation auf die Erstellung von spec-Files, nicht auf das Lesen. URL: `http://www.winehq.org/site/docs/winelib-user/spec-file`, zuletzt besucht am 2.11.2004.

Eintrag	Bedeutung
1113 stdcall GetTypeByNameA(str ptr)	Export
@ stdcall PlaySoundA(ptr long long)	Export by name
1109 stub GetAddressByNameA	Stub
3 stub @	Stub, exportiert by ordinal
1 stdcall accept(long ptr ptr) ws2_32.accept	Forward

Tabelle 6.1: Beispiele für Einträge in spec-Files. In der rechten Spalte ist jeweils kurz die Bedeutung des Eintrages genannt.

tierungsstatus' *stub* ist ebenfalls hilfreich und wird in diesen Analyseschritt eingebunden.

In der Implementierung der Analyse von spec-Files wird zunächst eine Liste der vorhandenen spec-Files erstellt. Bis auf die Endung sind die Namen dieser Dateien mit den Namen der mit ihrer Hilfe generierten Wine-DLLs identisch, so daß dadurch auch jeweils der Modulname bekannt ist. Es wird somit über jedes Element der Liste iteriert, und das Modul in der Modultabelle der Datenbank eingetragen. Eine eindeutige Identifizierung kann anhand von Name, Hashwert des spec-Files, sowie Wine-Version des Moduls vorgenommen werden. Mittels der Informationen im spec-File werden dann in der Export-Tabelle der Datenbank die Exporte des Moduls vermerkt, zusammen mit der Information, ob es sich um einen stub handelt. In Pseudocode wird also wie folgt vorgegangen:

```
1: Erstelle Liste der spec-Files SL
2: for m in SL do
3:    Datenbank.Modultabelle ← Modulinformationen(m)
4:    Lese Exporte(m)
5:    Datenbank.Exporttabelle ← Exporte(m)
6: end for
```

6.3.2 Auflösung von Forwards

Durch den ersten Analyseschritt – die Analyse von spec-Files – sind sämtliche zu berücksichtigende Funktionen ermittelt und in der Datenbank eingetragen. Auch der Implementierungsstand *stub* ist bereits ermittelt worden. Die Möglichkeit, Funktionen auf andere zu „forwarden“ macht jedoch einen weiteren Schritt in der Analyse erforderlich, nämlich die Auflösung dieser Forwards bzw. Handles. Für jede Funktion `foo()` ist eine Überprüfung nötig, ob diese möglicherweise auf eine andere Funktion `bar()` weitergeleitet wurde. Falls dem so ist, muß in der Datenbank nicht der Implementierungsstand von `foo()` bei `foo()` vermerkt werden, sondern der von `bar()`.

Um dies zu erreichen, wird analog zum vorherigen Schritt erneut über alle Exporte in den spec-Files iteriert. Für jede dieser Funktionen wird ermittelt, ob es sich um einen Forward handelt. Ist dem so, wird der Imple-

mentierungsstand des Forward-Ziels in der Datenbank eingetragen. Damit können Forwards bei der späteren Analyse genau so behandelt werden wie normale Exporte[6]. In Pseudocode wird also wie folgt vorgegangen:

```
1: Erstelle Liste der spec-Files SL
2: for m in SL do
3:    Lese Exporte(m)
4:    for e in Exporte(m) do
5:       if IstForward(e) then
6:          Datenbank.Exporttabelle(e) ← ImplStand(ForwardZiel(e))
7:       end if
8:    end for
9: end for
```

6.3.3 Analyse des Quelltextes

Die durch die Betrachtung der spec-Files gewonnen Informationen reichen für eine rundimentäre Beurteilung des Implementierungsstands der Exporte von Wine aus. Da allerdings grundsätzlich der gesamte Quelltext zur Verfügung steht, wurde dieser ebenfalls in die Analyse einbezogen. Im Quelltext selbst gibt es bestimmte Möglichkeiten, Unvollständigkeiten in der Implementierung einer Funktion an entsprechende Debug-Kanäle oder an die Standardausgabe zu kommunizieren. Für jede im spec-File als exportiert aufgeführte Funktion wird also versucht, ihre Implementierung in der korrespondierenden Quelltextdatei zu finden. Darin enthaltene bestimmte Schlüsselwörter (*stub*, *FIXME* etc.) werden als Indikatoren für den Implementierungsstand der Funktion benutzt.

Zunächst stellt sich das Problem, für ein bestimmtes Modul (d.h. spec-File) den korrespondierenden Quelltext zu finden. Dieser ist häufig auf mehrere Dateien aufgeteilt. Grundsätzlich befindet sich aber im Quelltextbaum von Wine jedes Modul mitsamt seinem spec-File in einem Unterverzeichnis des Verzeichnisses dlls/, so daß für die Analyse eines Modules jeweils der Quelltext im gleichen Verzeichnis wie das spec-File Verwendung findet. Es werden alle .c-Dateien in diesem Verzeichnis berücksichtigt. Die Quelltexte dieser Dateien werden in den Arbeitsspeicher geladen und alle Kommentare entfernt. Anschließend werden sie in einer Assoziativliste abgespeichert, die für eine jede Funktion ihre Signatur auf ihren Funktionsrumpf abbildet. Bei der Iteration über alle Exporte des Moduls – mittels des spec-Files

[6]Das Programmsystem ist in der Lage, Forwards in Windows-Applikationsmodulen zu erkennen und nachzuverfolgen. Bei solchen handelt es sich aber um genau die im Standard für Portable Executables spezifizierte Technik. Die Möglichkeiten, die Wine für seine eigenen Module bietet – die streng genommen keine PEs sind – sind zwar analog modelliert und somit sehr ähnlich, jedoch nicht notwendigerweise im jedem Detail identisch. Daher wurde für die Forwards in Wine-Modulen dieser Weg in der Implementierung gewählt.

– wird für jeden Export versucht, in dieser Assoziativliste den Rumpf der Funktionsimplementierung zu finden.

Für die Betrachtung des Rumpfes werden drei Indikatoren im Quelltext berücksichtigt:

- das Schlüsselwort „stub“, das eine fast vollständig fehlende Implementierung der Funktionalität signalisiert.
- das Schlüsselwort „semi-stub“, das als Indikator für eine kaum vollständig fertiggestellte Funktionalität dienen kann.
- das Schlüsselwort „FIXME“, das den Entwicklern bekannte Fehler in der Implementierung signalisiert.

Für jeden dieser Indikatoren wird im korrespondierenden Datensatz in der Export-Tabelle der Datenbank eingetragen, ob er gefunden wurde. Um den Fall „kein Indikator gefunden“ von dem potentiell möglichen Fall „Quelltext der Funktion nicht gefunden“ zu unterscheiden weden die drei Indikatorfelder in der Datenbank bei nicht aufgefundenem korrespondierendem Quelltext mit NULL belegt. Somit ließen sich diese Funktionen mit weniger genau ermitteltem Implementierungsstand in späteren Analyseschritten gesondert berücksichtigen.

6.4 Objektstruktur

Die bereits erläuterten Teilschritte der Analyse haben im Wesentlichen zum Ziel, die zu betrachtende Datenmenge einzugrenzen, zu strukturieren, und dann geeignet in der Datenbank des Systems abzulegen. Diese Schritte erfolgen für eine Software bzw. für eine Wine-Version jeweils genau einmal und sind deshalb aus Performanzgesichtspunkten als unkritisch einzustufen.

Die Ermittelung des gesamten API-Profils einer Software und dessen Abgleich mit Wine erfolgt dagegen in der Regel mehrmals. Zudem muß er sich nach verschiedenen Gesichtspunkten parametrisieren lassen, um die relevanten Konfigurationsmöglichkeiten von Wine zu berücksichtigen. Damit soll der Tatsache Rechnung getragen werden, daß Wine für viele Softwareprodukten die Windows-API noch nicht ausreichend vollständig implementiert um die Funktionalität sicherzustellen, weshalb diese Lücken mit Modulen einer meist vorhandenen und lizensierten Windows-Version geschlossen werden sollen. Das bedeutet konkret, daß sich eine Software nicht nur gegen verschiedene Wine-Versionen abgleichen lassen muß, sondern auch gegen verschiedene Konfigurationen von verschiedenen Wine-Versionen. Diese Analysen müssen performant durchführbar sein.

Das Ziel dieses Abschnittes ist es, zunächst die Klassen zu benennen und zu beschreiben, die die für die Analyse nötigen Entitäten wie z.B. Module, Importe und Exporte etc. modellieren. Die Datenbankschnittstelle, mittels

der die zuvor in der Datenbank abgelegten Informationen in Instanzen dieser Objekte übertragen wird, wird hier ebenso beschrieben wie der Mechanismus, mittels dem diese Übertragung erfolgt.

6.4.1 Entitäten

Da es Ziel des hier vorgestellten Verfahrens ist, im wirklichen Betrieb von Applikationen unter Linux/Wine auftretende Probleme durch unvollständige Implementierung der Windows-API durch Wine gewissermaßen „vorherzusagen“, erscheint es sinnvoll, den hierfür relevanten Teil von Wine und die daran beteiligten Einheiten nachzubilden. Von zentraler Bedeutung ist hier der Loader von Wine, der für das Laden von Modulen aus Wine und aus der Windows-Applikation zuständig ist. Die genannte Fehlerklasse würde zum Teil ggf. vom Loader zur Laufzeit erkannt, nämlich dann, wenn er die Importe eines Moduls der Applikation vergeblich in den Speicher zu laden versucht[7].

Der Loader von Wine versucht, das Verhalten des Windows-Loaders nachzubilden. Folglich ist es erforderlich, ein Objekt `Loader` zu erstellen, das sich bezüglich der wesentlichen Aspekte gleich verhält wie der Windows-Loader. Damit dieses Objekt sinnvoll tätig werden kann müssen die Entitäten mit denen es operiert ebenfalls nachgebildet werden. Dies sind insbesondere Wine-Module und PE-Module sowie deren Importe und Exporte.

Da alle Objekte mittels Daten aus der Datenbank initialisiert werden, die in vorherigen Analyseschritten dort eingetragen wurden, wird ein Mechanismus benötigt, der in geeigneter Weise eine Schnittstelle zur SQL-basierten Datenbank des Systems bietet. Für die Initialisierung der Objekte ist ein Mechanismus erforderlich, der diese gemäß dem Designziel möglichst großer Performanz dieses Analyseschrittes (unter Ausnutzung eventuell vorhandener Parallellität) nebenläufig erledigt.

Wine selbst bzw. insbesondere der Loader von Wine läßt sich auf verschiedene Arten parametrisieren. Auf diese Weise läßt sich beeinflussen, ob Wine im Zweifelsfall sein eigenes, nachimplementiertes Modul verwendet oder auf ein im Dateisystem vorhandenes zurückgreift. Für das Auffinden eventueller Lücken in der Implementierung und für die Ermittelung einer optimalen Konfiguration unter Benutzung von echten Windows-Modulen ist es nötig, diese Parametrisierung im System abbildbar zu machen. Zu diesem Zweck wird ein `Configuration`-Objekt eingeführt.

Für die Auswertung von Analysen ist es wünschenswert, Softwareprodukte mit ihren Modulen und weiteren Informationen wie Hersteller, Versionsnummer etc. zu gruppieren. Für diese Projekte wird ein entsprechendes Objekt eingeführt.

[7] Fehler durch zwar vorhandene, aber unvollständig implementierte APIs blieben dabei aber weiterhin unentdeckt.

Funktionen Sowohl Importe als auch Exporte sind Funktionen, die von einem PE-Modul oder einem Wine-Modul exportiert bzw. aus einem anderen Modul importiert werden. Die Schnittmenge der Eigenschaften, die beide besitzen ist groß. Daher wurde zunächst eine Oberklasse `Function` erstellt, von der die `ExportedFunction` und die `ImportedFunction` jeweils erben. In UML-Notation[8] graphisch veranschaulicht wird dies in Abbildung 6.5. instanziiert werden die `Function`s jeweils mit den über sie in der Datenbank gespeicherten Informationen wie Name, Ordinal, Einsprungspunkt etc.

Während einer Analyse werden die jeweils exportierten Funktionen je nach Wine-Version und Konfiguration unterschiedlich oft von anderen Modulen referenziert. Diese Information ist also zwar der jeweiligen `Function` zuzuordnen, jedoch abhängig von verschiedenen anderen Parametern und somit dynamisch. Daher werden diese Informationen nicht in der Datenbank abgelegt, sondern zur Laufzeit der Analyse vom jeweils referenzierten `Function`-Objekt gespeichert und in einer Hashtable verwaltet. Zugänglich sind die Namen der referenzierenden Module über die Methode `getReferences()`; die Zahl der Referenzen, die für die Gewichtung einer möglichen Problemstelle wichtig sind, ist über `getReferenceCount()` abrufbar.

Exporte, d.h. exportierte `Function`s, können vom exportierenden Modul auf einen anderen Export umgeleitet werden (Forward). Dieser wesentliche Unterschied wird in der `ExportedFunction` abgebildet. Mittels der Methode `isForward()` läßt sich abfragen, ob es sich bei dem Export um einen Forward handelt; analog hierzu existiert die Methode `isHandle()`. `getForwardDll()` und `getForwardFunction()` liefern das Zielmodul und die Zielfunktion des Forwards. Über die Methode `getImplStatus()` läßt sich der Implementierungsstatus der Funktion (Spec-Stub[9], Stub, Semi-Stub, FIXME etc.) als Zahlenwert repräsentiert abfragen.

Trifft die Analyselogik auf eine benötigte Funktion (d.h. eine `ImportedFunction`), die vom exportierenden Modul nicht zur Verfügung gestellt wird, so handelt es sich um eine Problemstelle. Es ist wünschenswert, auf diese Problemstellen – die nichts anderes sind als fehlende (`Exported`)`Functions` – mittels der gleichen API zugreifen zu können wie auf andere `Function`s auch. Hierfür wurde die Klasse `VirtualFunction` eingeführt. Diese läßt sich aus einer `ImportedFunction` instanziieren, die im Konstruktor mit übergeben wird, und bildet das fehlende (daher „virtuelle“) Gegenstück zu dem Import. Diese Eigenschaft läßt sich über die von Function geerbte und überschriebene Methode `isVirtual()` abfragen, die für `VirtualFunction`s stets `true` zurückgibt.

[8] Die hier präsentierten Diagramme enthalten aus Gründen der Übersichtlichkeit nur die wichtigsten Operationen und Variablen. In Abweichung von der UML-Notation werden mit den Präfixen „+ “ und „-“ keine Sichtbarkeitsbereiche bezeichnet.

[9] Eine Funktion, die in ihrem spec-File als *stub* markiert ist, wird im Folgenden als Spec-Stub bezeichnet.

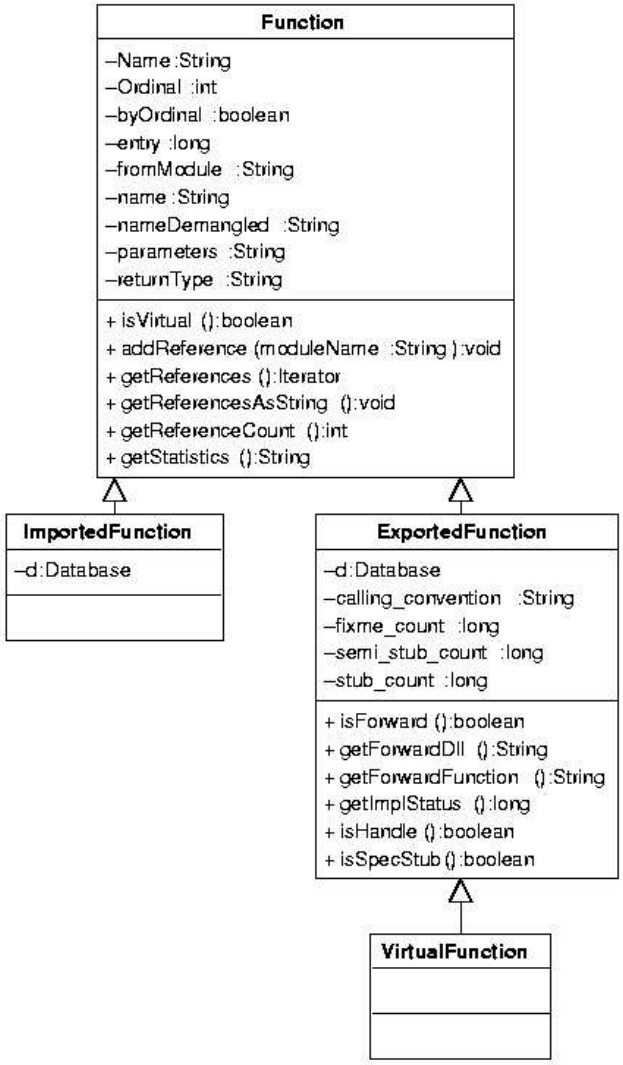


Abbildung 6.5: UML-Darstellung der Klassen, die Funktionen in PE-Modulen modellieren.

Module Wine-Module, die die Windows-API nachimplementieren, bilden im Wesentlichen[10] die gleichen Strukturen nach, wie ihre Originale unter Windows. Da es sich bei diesen Windows-Modulen genau wie bei Modulen von Windows-Applikationen um Portable Executables handelt, werden sowohl Wine- als auch PE-Module in der gleichen Struktur im System abgebildet: der Klasse `Module`. Ein solches `Module` wird zunächst wie z.B. eine `Function` auch mit den über es in der Datenbank vorhandenen Informationen wie Name, Größe etc. instanziiert. Ein Modul ist jedoch nichts anderes als eine Art Container für die Funktionalitäten, die das Modul mit seinen Exporten bereitstellt. Zudem besitzt jedes Modul noch eine gewisse Anzahl von Importen, die als `ImportedFunction`s dargestellt werden – diese Zusammenhänge sind als UML-Diagramm in Abbildung 6.6 dargestellt. Diese Informationen müssen aus der Datenbank geholt und in für jedes Im- bzw. Export in eine neue Instanz der Klasse `ImportedFunction` bzw. `ExportedFunction` gepackt werden. Bevor dieser Instanziierungsprozeß nicht abgeschlossen ist, ist eine `Module`-Instanz nicht für eine Analyse verwendbar, d.h. sie ist nicht `isInitialized()`. Der Mechanismus für diese Initialisierung wird über die Methode `initialize()` angestoßen und wird in Abschnitt 6.4.3 detailliert dargestellt.

Eine fertig initialisierte `Module`-Instanz stellt eine Liste seiner Exporte als `ArrayList` von `ExportedFunction`s über die Methode `getExports()` zur Verfügung. Analog hierzu sind die Importe über einen `Iterator` über `ImportedFunction`s zugänglich, den die Methode `getImports()` zurückgibt. Wie auch bei Funktionen gibt es für Module Informationen, die dynamisch zur Laufzeit der Analyse ermittelt werden, und die zwar dem Modul zuzuordnen sind, jedoch von verschiedenen Parametern abhängen und deswegen nicht in der Datenbank abgelegt werden. Hierzu zählen die Module, die jeweils ein betrachtetes Modul referenzieren – diese Information hängt natürlich davon ab, welche anderen Module gerade zusammen mit dem betrachteten analysiert werden. Des weiteren ist es insbesondere für Wine-Module ein häufiger Fall, daß in einem Modul nicht vorhandene (d.h. nicht implementierte) Funktionen referenziert werden. Ob dies so ist und welche Module betroffen sind ist ein Ergebnis, das eine Analyse erst ermitteln muß, und das sich natürlich für jede Analyse unterscheidet.

Die Methode `getReferencingModules()` akzeptiert als Parameter ein Analyse-Objekt und gibt alle Module zurück, die das betrachtete referenzieren. Dies geschieht mittels Iteration über alle Exporte und dem jeweiligen Abfragen der Methode `getReferences()`. Falls das übergebene Analyse-Objekt null ist, werden nur die Referenzen auf implementierte Funktionen berücksichtigt, da die Informationen über die Referenzierung nicht implementierter Funktionen jeweils nur einer Analyse-Instanz bekannt sind.

Vergleichbar mit den `VirtualFunctions`, die fehlende Exporte eines Mo-

[10] „Im Wesentlichen“ heißt hier: „in den für die Analyse wesentlichen Aspekten“.

duls modellieren und dafür die gleiche API zur Verfügung stellen wie für Exporte existiert eine Klasse `VirtualModule`. Werden während einer Analyse Module gefunden, die gänzlich fehlen, d.h. im Dateisystem unter dem gesetzten Suchpfad nicht aufzufinden sind, so werden diese mittels einer Instanz von `VirtualModule`, einer von `Module` abgeleiteten Klasse, dargestellt. Ein `VirtualModule` enthält konzeptbedingt immer nur `VirtualFunction`s. Diese bilden die Exporte ab, die das Modul deswegen nicht zur Verfügung stellt, weil es nicht vorhanden ist.

Somit kann ermittelt werden, welche Funktionen aus einem fehlenden Modul überhaupt benötigt werden. Ein `VirtualModule` stellt die Liste dieser Funktionen über `getVirtualExports()` zur Verfügung. Ob ein Modul „virtuell“ ist, kann analog zur Implementierung dieser Problematik in `Function`s über die Methode `isVirtual()` abgefragt werden. Einem neu erstellten `VirtualModule` werden über seine Methode `addVirtualExport()` neue `VirtualExport`s hinzugefügt. Diese werden aus den `ImportedFunction`s instanziiert, die die fehlenden Exporte referenzieren – daher akzeptiert `addVirtualExport()` als Parameter eine `ImportedFunction`.

Projekt Es wurde davon ausgegangen, daß in der Regel jeweils eine bestimmte Windows-Software auf ihre Ausführbarkeit unter einer bestimmten Wine-Version untersucht werden soll. Dies impliziert, daß sowohl bestimmte Programmmodule, nämlich die der zu untersuchenden Wine-Version, und die Module einer bestimmten Wine-Version gemeinsam gruppiert betrachtet werden sollen. Um das Zusammenspiel dieser beiden zu analysieren muß neben dieser Gruppierung auch eine Initialisierung aller beteiligter Komponenten – Module, Funktionen, etc. – erfolgen. Diese Aufgabe erfüllt das `Project`-Objekt. Es enthält optional ein `Configuration`-Objekt, das die gerade erwünschte Konfiguration der verwendeten Wine-Version abbildet. Ihm zugeordnet ist ein `ProjectMeta`-Objekt, das weiterführende Metainformationen über das Projekt wie z.B. Name des Softwareherstellers, Installationsprotokoll etc. verwaltet.

Ein Projekt ist in der Datenbank eindeutig gekennzeichnet durch seinen Namen. Die Menge aller Module, die zu ihm gehören, läßt sich aus der Projekte-Tabelle entnehmen. Für jede Wine-Version gilt, daß alle Informationen über ihre Module direkt aus der Modul-Tabelle entnommen werden können. Somit kann über die Datenbankschnittstelle ein `Project`-Objekt direkt instanziiert und seine Initialisierung angestoßen werden. Hierbei werden alle beteiligten Module der Windows-Software aus der Datenbank entnommen, als `Module` instanziiert und jeweils deren Initialisierung gestartet. Schließlich werden diese Instanzen über `addModule()` dem `Project`-Objekt hinzugefügt. Die Wine-Module werden ebenfalls mit Informationen aus der Datenbank instanziiert. Alle Module werden außerdem einer `Loader`-Instanz, die dem Projekt zugeordnet ist, bekannt gemacht. `Loader` werden

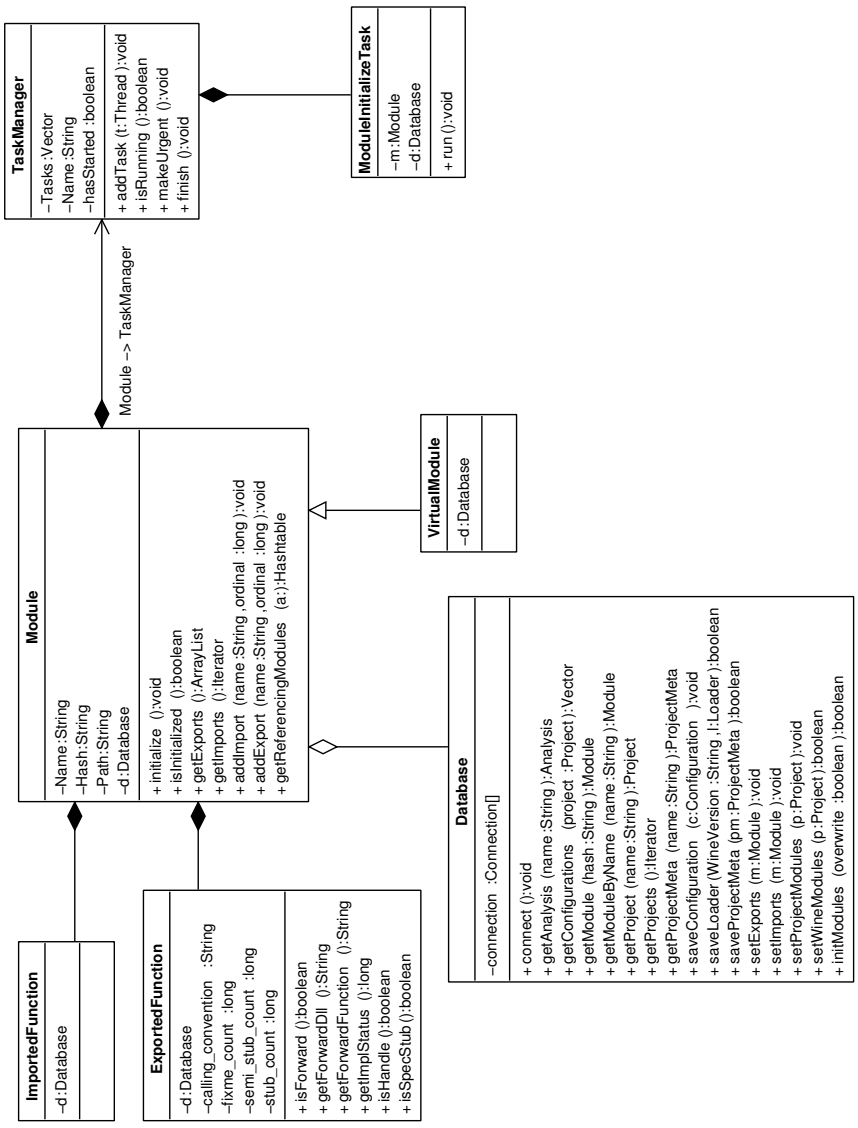


Abbildung 6.6: UML-Darstellung der Klassen, die Module modellieren. Für echte PE-Module und für Wine-Module werden die gleichen Klassen verwendet.

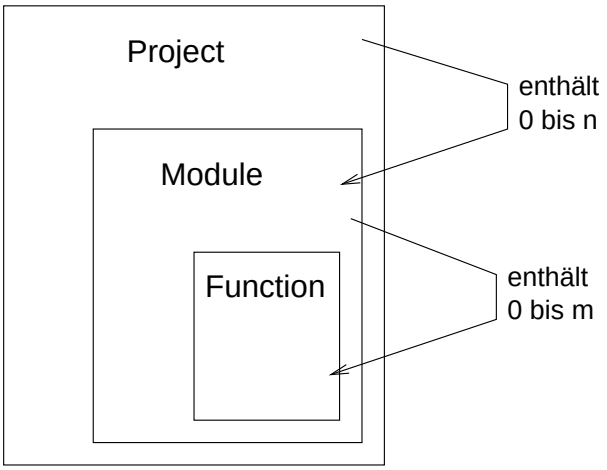


Abbildung 6.7: Schematische Darstellung der Aggregationsbeziehungen zwischen Projekt, Modul, und Funktion. Damit die jeweils umschließende Entität als initialisiert und somit für Analysen verwendbar gelten kann, müssen alle enthaltenen Entitäten initialisiert sein.

für jedes Projekt jeweils neu instanziiert und initialisiert. In der Datenbank vorhandene `Loader`-Instanzen, die mit den Modulen einer bestimmten Wine-Version vorinitialisiert sind, können bei Bedarf verwendet werden, um diesen Prozeß zu beschleunigen (hierzu siehe 6.5).

Nach Beendigung dieser Initialisierungsprozedur steht das `Project`-Objekt für die Verwendung in einer Analyse zur Verfügung.

Loader Das `Loader`-Objekt des Systems stellt eine Nachbildung der Funktionsweise des Windows- bzw. Wine-Loaders dar. Im Gegensatz zu seinen „Vorbildern“ operiert der `Loader` hier jedoch nicht auf wirklichen PE- oder Wine-Modulen, sondern auf den Entitäten des Systems. Er lädt also keine Executables in den Speicher, sondern dient lediglich dazu nachzuprüfen, ob der Windows- bzw. Wine-Loader dies unter den gegebenen Umständen – Windows-Software X, Wine-Version Y mit Konfiguration Z – erfolgreich tun könnte. Bei dieser Überprüfung ans Tageslicht kommende Probleme können so vom Analyse-Objekt gesammelt und entspechend ausgewertet werden.

Einer `Loader`-Instanz werden über `addModule()` PE-Module bekannt gemacht; für Wine-Module geschieht dies über `addWineModule()`. Ob ein Modul A ein anderes Modul B überhaupt importieren kann, läßt sich über die Methode `getImportedModule()` abfragen. Diese erhält A und den Namen von B als Parameter, und liefert entweder eine Instanz des importierten Moduls zurück, oder `null`. Ob zu einer importierten Funktion X ihr Gegenstück gefunden werden kann, kann dann beim Modul mittels `hasFunction()` abgefragt werden. `Loader` sind prinzipbedingt umfangreiche Objekte: Obwohl sich `Loader` auch ohne Wine-Module benutzen lassen um beispielsweise das

gesamte API-Profil einer Windows-Applikation zu ermitteln, ohne dies mit einer bestimmten Wine-Version zu vergleichen, so ist doch die Analyse von Windows-Software mit Wine der Regelfall. Wine allein besitzt etwa 200 Module mit insgesamt gut 20 000 Funktionen, was somit die Mindestzahl an Modulen und Funktionen darstellt, die vom `Loader` zu verwalten ist. `Loader`-Instanzen, die mit den Modulen einer bestimmten Wine-Version vorinitialisiert sind, lassen sich persistent in der Datenbank ablegen und wieder reinstanziieren, was in erheblichem Umfang Zeit und Ressourcen spart (siehe hierzu Abschnitt 6.5).

Konfiguration Um die Möglichkeiten von Wine, das Verhalten des Loaders zu konfigurieren, im System abzubilden, wurde eine Klasse eingeführt, die diesen Teil der Wine-Konfiguration für die Analysen nutzbar macht. Für ein Projekt soll in der Regel nicht nur eine, sondern meist mehrere Konfigurationen getestet und auch abgespeichert werden können. Daher ist es möglich, Konfigurations-Objekte textuell repräsentiert in der Datenbank abzulegen und aus einer solchen textuellen Repräsentation auch wieder zu instanziieren.

Eine Konfiguration erlaubt das logische Entfernen von physisch vorhandenen Modulen aus dem Projekt, so daß diese Module dann von Wine bereitgestellt werden müssen. Umgekehrt können auch fremde Module beispielsweise aus einer Windows-Version die beim Kunden vorhanden und lizensiert ist verwendet werden. Diese können dem Projekt virtuell hinzugefügt werden. Da das Windows-Systemverzeichnis immer im Suchpfad des Wine-Loaders liegt, werden solche Module logisch im Windows-Systemverzeichnis abgelegt und haben somit Vorrang vor Wine-eigenen Modulen.

Eine weitere Möglichkeit das Verhalten des Wine-Loaders zu beeinflussen ist das Setzen von Suchpfaden für Module. `Configuration`-Objekte enthalten ebenfalls eine Liste von Pfaden, die vom `Loader` analog zum Verhalten des Wine-Loaders durchsucht werden.

Analyse Für die Auswertung der in die Entitäten des Systems eingebundenen Informationen, d.h. insbesondere für des Abgleich des API-Profils der Windows-Applikation mit dem Implementierungsstand der gewählten Wine-Version, wurde ein Objekt geschrieben. Dieses `Analysis`-Objekt wird mit einem `Project`-Objekt instanziiert und führt, sobald das `Project` fertig initialisiert ist, nach Aufruf von `analyze()` einen Abgleich der APIs durch. Diese können später bei Bedarf wieder in den Speicher geladen werden und über die gewohnten Schnittstellen abgefragt werden.

Fehlende oder unvollständige Module können beispielsweise anschließend über `getIncompleteModules()` oder `getMissingModules()` abgefragt werden. Vergleichbare Schnittstellen existieren für die fehlenden Funktionen von Modulen. Für jede dieser ermittelten Schwachstellen läßt sich zudem

ihre Gewichtung, d.h. die Zahl der Referenzen auf diese fehlende oder unvollständige API, abfragen. Zudem ist es möglich, hinsichtlich des akzeptablen Implementierungsstatus von Wine-APIs ein Mindestniveau (Spec-Stub, Stub, Semi-Stub etc.) vorzugeben.

Ob eine Ausführung mit Wine problemlos möglich wäre, weil von der Analyse keinerlei fehlende APIs entdeckt wurden, läßt sich mit `runsWithWine()` abfragen. Des weiteren kann man mittels `saveResults()` die Speicherung des Analyse-Objekts in der Datenbank veranlassen. Aus Platzgründen kann dabei auf die gleichzeitige Abspeicherung des von dem in der Analyse enthaltenen Projekts referenzierten `Loader`-Objekt verzichtet werden. Für eine detailliertere Beschreibung der Analyse sei an dieser Stelle auf den Abschnitt 6.7 verwiesen.

6.4.2 Datenbankschnittstelle

Für alle Entitäten des Systems galt es, eine Schnittstelle zum unterliegenden SQL-basierten Datenbanksystem zu finden. Grundsätzlich wird lediglich eine Instanz dieses sog. `Database`-Objekts erzeugt, und diese dann von allen Objekten an eventuell neu erzeugte Objekte im Konstruktor durchgereicht, so daß bei einer Analyse alle Objekte die gleiche Instanz von `Database` referenzieren. Das `Database`-Objekt selbst unterstützt mehrere (konfigurierbar viele) gleichzeitige Verbindungen und nutzt so die Threading-Fähigkeiten der unterliegenden Datenbank. Die Verbindung erfolgt hierbei über die Schnittstelle JDBC, wodurch das unterliegende Datenbanksystem MySQL grundsätzlich auch mit geringem Aufwand durch ein anderes ersetzt werden könnte, sollte sich das als nötig erweisen.

Aufgrund der zahlreichen verschiedenen Objekte, die das `Database`-Objekt als Schnittstelle zur Datenbank benutzen, verfügt `Database` über eine ganze Reihe von APIs. Grundsätzlich liefern diese Entitäten des Systems zurück, beispielsweise wird mittels `getModule()` ein (durch seinen Hashwert eindeutig identifiziertes) Modul mit den über es in der Datenbank gespeicherten Informationen instanziiert, der Initialisierungsprozeß des Modul-Objekts angestoßen, und die Modul-Instanz zurückgegeben. Jede Methode von `Database` ruft zunächst die Methode `connect()` auf, die eine Verbindung zur Datenbank herstellt falls noch keine existiert, bzw. nach dem Round-Robin-Prinzip für jede Datenbankanfrage eine andere Verbindung zur Verfügung stellt. `Database`-APIs wurden durchgängig nach diesem Muster programmiert.

6.4.3 Initialisierungsmechanismus

Es sollte bis hierhin deutlich geworden sein, daß vorgelagerte Analyseteile – namentlich die PE Header Analyse und die Wine-Quelltextanalyse – Informationen in der Datenbank des Systems zur späteren Durchführung der

eigentlichen Kompatibilitätsanalyse ablegen. Grundsätzlich stellt sich nun die Aufgabe, diese Informationen in Instanzen der Entitäten des Systems, also `Function`, `Module`, `Project` etc. zu übersetzen. Wie in Abbildung 6.7 verdeutlicht, aggregieren diese Objekte jeweils andere in sich, so daß das umschließende Objekt erst initialisert ist, wenn alle seine enthaltenen Entitäten ebenfalls initialisiert sind. Gemäß dem Designziel hoher Performanz der Kompatibilitätsanalyse sowie dem der Ausnutzung vorhandener Parallelität ist es geraten, die Initialisierung nebenläufig ablaufen zu lassen.

Einer jeden zu initialisierenden Entität wird also ein Objekt zugewiesen, das die für seine Initialisierung notwenigen Aufgaben sammelt und nebenläufig ausführt. Diesem sog. `TaskManager` kann mittels `addTask()` eine Aufgabe hinzugefügt werden; er kann auch mittels `hasFinished()` befragt werden, ob die von ihm zu überwachenden Aufgaben bereits erledigt wurden. Die Aufgaben, die ein solcher `TaskManager` zu überwachen hat, sind von `java.lang.Thread` abgeleitet und spezifisch für die Eintitäten zu denen sie gehören. Es gibt somit einen `ModuleInitializeTask` und einen `ProjectInitializeTask`. Nach Beendigung ihrer Initialisierungsaufgabe werden die von `Thread` abgeleiteten Klassen aus dem Speicher entfernt[11].

6.5 Objektspeicherung

Analysen Es ist wünschenswert, bereits durchgeführte Analysen in der Datenbank abzulegen, um später wieder auf diese zugreifen zu können. Zwar wäre es möglich, diese Ergebnisse beispielsweise in Textform zu speichern, jedoch ist es bei weitem effizienter, auch für diese gespeicherten Analysen die bereits vorhandene API weiterhin zu gewährleisten. Daher wurde die Möglichkeit geschaffen, Analyseergebnisse – also Instanzen von Analyse-Objekten – persistent in der Datenbank abzulegen.

Eine Analyse wird immer bezogen auf ein Projekt und eine bestimmte Wine-Version durchgeführt. Somit sollen zu einem Projekt mehrere Analysen in der Datenbank verfügbar sein. Um dies zu gewährleisten werden Analyse-Objekte in einer Hashtable abgelegt, die als Schlüsselwert die verwendete Wine-Version benutzt. Die Zuordnung einer solchen Hashtable zu einem Projekt erfolgt mittels des Projektnamens über die entsprechende Zeile in der Datenbanktabelle.

Ein nicht initialisiertes `Project`-Objekt kann mittels `getAnalysisForWine()` dazu veranlaßt werden, in der Datenbank nach einer zu ihm passenden Analyseinstanz mit der übergebenen Wine-Version zu suchen. Bei Erfolg liefert diese Methode ein Analyse-Objekt zurück;

[11] Dies dient der Verringerung von Speicherbedarf und ist zudem der Tatsache geschuldet, daß Threads in Java nicht serialisierbar sind. Die initialisierten Objekte sollen aber teilweise serialisiert in der Datenbank abgelegt werden können.

schlägt sie hingegen fehl, wird sie zu `null` ausgewertet. In diesem Fall ist es erforderlich, das `Project`-Objekt und die Analyse selbst zu initialisieren. Es ist möglich mittels der genannten API nur Analyse-Objekte aus der Datenbank zu holen, die einen dazu passenden `Loader` mit abgespeichert haben. Die Speicherung von Analyse-Objekten in der Datenbank erfolgt über die API `saveAnalysisForWine()` von `Project`, das seinerseits dem `ProjectMeta`-Objekt die Aufgabe der endgültigen Speicherung überträgt.

Loader Wie bereits in Abschnitt 6.4.1 ausgeführt ist der Prozeß des Instanziierens von allen Wine-Modulen und Funktionen einer bestimmten Wine-Version als Fixkosten einer jeden Analyse zu betrachten. Diese Initialisierung des `Loader`s ist zudem aufgrund der großen Zahl an Modulen und Funktionen in Wine, die ja immerhin die API eines ganzen Windows-Betriebssystems[12] nachbilden sollen, ein recht zeitraubender Prozeß. Es liegt also nahe, selbigen zu beschleunigen.

Um dies zu erreichen wurde die Möglichkeit geschaffen, fertig initialisierte `Loader`-Instanzen in der Datenbank in einer gesonderten Tabelle – der Wine-Metatabelle – abzulegen und bei Bedarf wieder in den Speicher zu laden. Diese werden dann bei der Ausführung der `analyze()`-Methode der Analyseobjekte automatisch benutzt. Der hierdurch gewonnene Geschwindigkeitsvorteil hängt vom Umfang der betrachteten Windows-Software ab. Als Orientierung sei hier lediglich angegeben, daß die Analysedauer von durchschnittlich umfangreichen Softwareprodukten unter der Benutzung von vorinitialisierten `Loader`n von etwa 160 auf etwa 25 Sekunden schrumpft.

Module Die Klasse `Module` modelliert genau die Informationen, die aus der jeweiligen PE-Datei bzw. dem Wine-Modul gewonnen werden können. Somit ist es möglich, das System zusätzlich dadurch zu beschleunigen, daß initialisierte Modulinstanzen in der Datenbank abgelegt werden. Diese Instanzen werden bei Bedarf direkt aus der Datenbank heraus in den Speicher geladen. Eine Vorinitialisierung der Module läßt sich über die Kommandozeilenschnittstelle des Systems veranlassen. Der hierdurch erzielbare Geschwindigkeitsvorteil ist insbesondere für umfangreiche Programme erheblich[13].

6.6 Datenaufbereitung

Die im System gespeicherten Daten, insbesondere aber die Ergebnisse von Kompatibilitätsanalysen können grundsätzlich auf verschiedene Arten für ei-

[12] Genauer betrachtet bildet Wine sogar die API mehrerer Windows-Betriebssysteme ab, denn Wine strebt die Kompatibilität mit Windows 95, 98, NT 3.51, NT 4.0, 2000 und XP an.

[13] Beispielsweise reduziert sich die Analysedauer von einem umfangreichen Programm wie Prosoz/S auf einem Pentium IV mit 2 GHz von ca. acht Minuten auf etwa 55 Sekunden.

ne weitere Verarbeitung zugänglich gemacht werden. Von den fünf Möglichkeiten, die am naheliegendesten erscheinen, wurden vier im System implementiert.

6.6.1 GUI

Vornehmlich für Forschungszwecke und zum Test des Systems gedacht, wurde eine einfache GUI implementiert, die es erlaubt, alle in der Datenbank gespeicherten Projekte abzurufen und Analysen dieser Projekte zu starten. Ergebnisse dieser Analyse werden graphisch dargestellt. Zu allen entdeckten Problemstellen sind interaktiv weitere Informationen abrufbar, wie beispielsweise die Module, die die Problemstelle referenzieren. In Abbildung 6.8 ist ein Screenshot der GUI zu sehen.

Der Benutzer wählt hier zunächst aus den in der Datenbank vorhandenen Projekten eines zur Analyse aus, und selektiert anschließend die Wine-Version, mit der die Analyse durchgeführt werden soll. Durch einen Klick auf den „Go“-Button wird die Analyse gestartet. Ergebnisse werden sofort in die GUI eingetragen. „Modules in Project“ gibt eine Übersicht über alle Module, die zur analysierten Software gehören, während „Problems“ gefundene problematische Module tabellarisch auflistet. „Diagnosis“ erlaubt dann die erwähnte interaktive Darstellung auch einzelner Funktionen in den Modulen sowie weiterer Informationen zu den Problemstellen. Über einen Klick auf „Save Results“ wird das Analyse-Objekt in der Datenbank abgelegt.

6.6.2 Kommandozeile

Insbesondere für skriptgesteuerte Auswertung oder zum Abruf einzelner Informationen erschien es sinnvoll, einen textbasierten Zugang zum System zu ermöglichen. Gerade die Möglichkeit neue Analysen in der Datenbank vorhandener Windows-Software vollautomatisch – beispielsweise mit neuen Wine-Versionen – durchführen zu können ist durch eine Kommandozeilen-Schnittstelle leicht zu gewährleisten. Es wurden daher die folgenden Features für die Kommandozeile implementiert:

1. Auflisten von Projekten. Über den Switch -list-projects werden alle in der Datenbank vorhandenen Projekte ausgegeben.

2. Auflisten von Modulen. Mittels der Übergabe des Switches -list-modules [project] werden alle Module, die im angegebenen Projekt enthalten sind ausgegeben.

3. Analyse und Speicherung. Die Übergabe von -analyze-with-wine [project] [wineversion] analysiert das angegebene Projekt mit der angegebenen Wine-Version und gibt das ergebnis textuell auf der Konsole aus. -analyze-and-save-with-wine dient dem gleichen Zweck, speichert

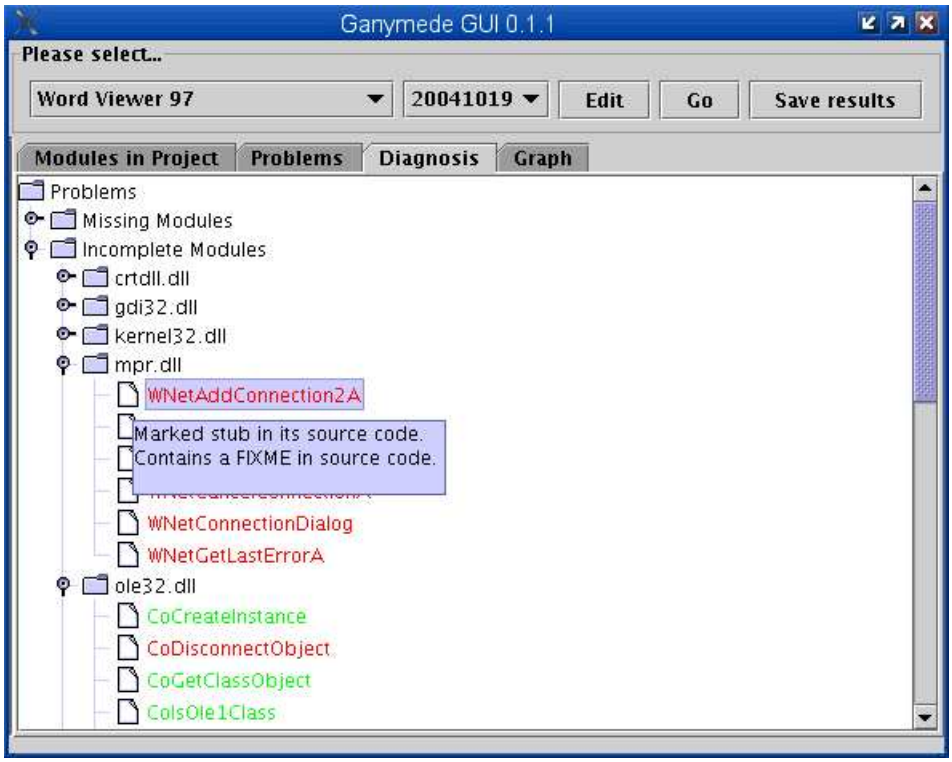


Abbildung 6.8: Screenshot der GUI des Systems. Dargestellt wird ein Analyseergebnis des Microsoft Word Viewers mit Wine Version 20041019. Die Anzeige erfolgt hier gruppiert nach Modulen, die unvollständige Implementierungen von Windows-APIs enthalten. Durch Doppelklick auf ein Modul oder eine API sind weitere Informationen wie z.B. die referenzierenden Module abrufbar.

aber zusätzlich noch das Analyse-Objekt mit den Ergebnissen in der Datenbank.

4. Loaderinitialisierung. Der Switch -init-loader [wineversion] erstellt eine `Loader`-Instanz mit den Modulen der angegebenen Wine-Version und legt diese zur späteren Verwendung als vorinitialiserte `Loader` in der Datenbank ab.

5. Modulinitialisierung. Mittels -init-modules [project] werden alle noch nicht vorinitialisierten Module des angegebenen Projektes initialisiert und diese Instanzen in der Datenbank gespeichert. Alternativ können mittels -init-modules-with-overwrite [project] auch bereits gespeicherte Instanzen überschrieben werden.

6. GUI-Start. Mittels -gui kann die GUI des Systems gestartet werden.

6.6.3 Report

Es ist naheliegend, die gesammelten Informationen auch Dritten in Papierform zugänglich zu machen. Beispielsweise könnte ein Analyseergebnis in einem Dokument verwendet werden, das die Kompatibilität dieser Software mit Wine begutachten soll. Hierfür ist es wünschenswert, daß die Informationen aus dem Analyseergebnis automatisch in ein Format gebracht werden, das von Menschen verstanden werden kann, um dieses hernach auszudrucken. Daher wurde eine automatische Generierung von Reports implementiert, die alle wesentlichen Informationen des Analyseergebnisses tabellarisch gegliedert automatisch ausgibt.

In der Implementierung wurde auf das freie Framework JFreeReport (siehe Abschnitt 4.5) zurückgegriffen. JFreeReport verwendet XML-basierte Templates, um daraus dann Reports zu erstellen, die in verschiedenen Formaten abgespeichert werden können, darunter HTML, PDF und Microsoft Excel. Um aus Analyse-Objekten automatisch Reports generieren zu können wurde im System die Klasse ReportDocument implementiert. Diese akzeptiert im Konstruktor eine Instanz der Analyse-Klasse, von der es über seine API alle Informationen für den Report abruft. Der Report kann wie in Abbildung 6.9 am Bildschirm als Preview angezeigt werden und dann vom Benutzer in dem gewünschten Format abgelegt werden. Eine automatische Ablage z.B. als PDF-Dokument ist mittels der API von JFreeReport ebenfalls problemlos möglich.

6.6.4 API

Der vielseitigste und wahrscheinlich eleganteste Weg, Informationen des Systems zugänglich zu machen, ist die Einführung einer API. Die bisher beschriebenen Arten der Datenaufbereitung machen von unterschiedlich

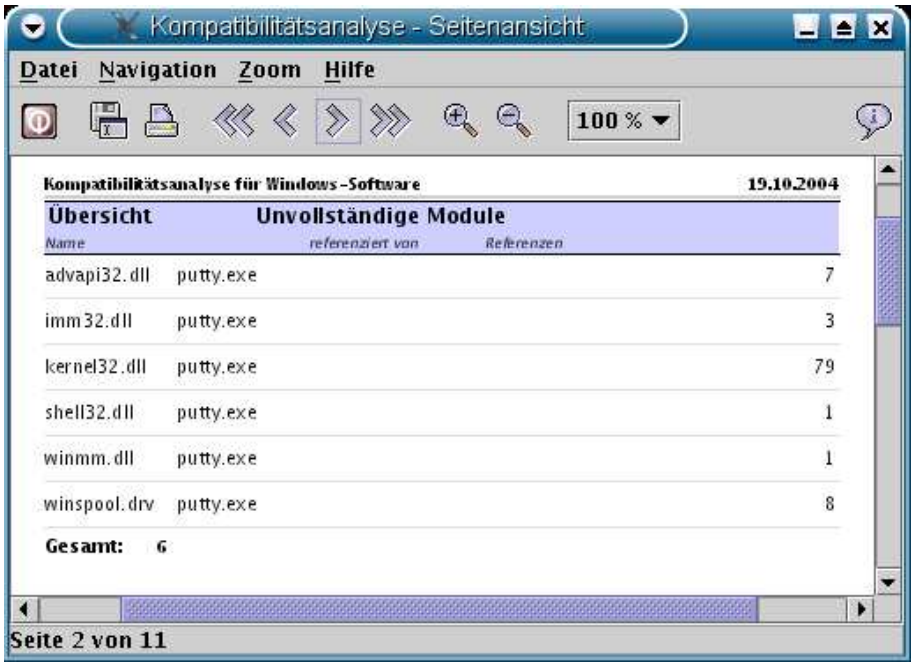


Abbildung 6.9: Screenshot der Preview eines vom System automatisch generierten Reports.

großen Teilen der API des Systems Gebrauch. Aufbauend auf der API des Systems existiert ein von Stefan Munz programmiertes Tool namens *sysiphus*, das dazu dient, mit Hilfe der Analyseergebnisse und verschiedener Konfigurationen von Wine sowie vorhandenen Windows-DLLs die optimale Wine-Konfiguration zu ermitteln. Die einzelnen Funktionen der API sind in der JavaDoc-Dokumentation des Systems dokumentiert.

Grundsätzlich wird vor jeder Analyse zunächst eine Instanz des `Database`-Objekts erzeugt. Diese kann dann wie folgt nach den abgespeicherten Projekten befragt werden:

```
// new Database instance
Database d = new Database();

// get all stored projects and print their name
Iterator i = d.getProjects();
while (i.hasNext()) {
   Project p = i.next();
   System.out.println( "Projektname: "+p.getName() );
   }
```

Projektabruf aus Datenbank Beim obigen Abrufen der Projekte werden keine initialisierten Projekt-Instanzen zurückgegeben. Die Initialisie-

rung[14] muß manuell angestoßen werden, wobei hierbei eine Wine-Version, mit der die Analyse erfolgen soll, mit angegeben werden muß. Die Instanziierung und Initialisierung eines `Project`-Objekts erfolgt dann also mittels

```
// new Database instance
Database d = new Database();

// fetch project from database
Project p = d.getProject(pname);

// initialize Project instance with Wine version 20040813
p.initialize("20040813");
```

Analyseabruf aus Datenbank Will man eine Analyse hinsichtlich der Kompatibilität von Software A mit Wine-Version B durchführen, so besteht die Möglichkeit, daß diese Analyse bereits als Objekt in der Datenbank abgespeichert wurde. Dann könnte man diese gespeicherte Instanz einfach wieder in den Speicher laden und verwenden:

```
// new Database instance
Database d = new Database();

// fetch project from database
Project p = d.getProject(pname);

/* retrieve Analysis instance from DB. That instance must
   have used Wine version "20040813" and need not contain a
   Loader instance. */
Analysis a = p.getAnalysisForWine("20040813", false);

// error checking
if (a == null) {
   System.err.println("No matching instance found in DB.");
   }

/* update project reference to the one we
   have just retrieved from the DB */
else { p = a.getProject(); }
```

Neue Analyse Eine Analyse „from scratch“ implementiert der folgende Beispielcode. Dabei wird die Initialisierung des `Project`-Objekts von der

[14] Der Initialisierungsmechanismus ist detailliert in Abschnitt 6.4.3 beschrieben.

Analyse beim Aufruf von `analyze()` automatisch durchgeführt, sofern das erforderlich ist.

```
// new Database instance
Database d = new Database();

// fetch project from database
Project p = d.getProject(pname);

// new Analysis instance
a = new Analysis(p);

/* start analysis with Wine version 20040813
   and retrieve first guess of best configuration */
Configuration c = null;
c = a.analyzeWithWine("20040813");
```

Speicherung von Analysen Eine erfolgreich durchgeführte Analyse kann in der Datenbank abgespeichert werden, um sie zu späteren Zeitpunkten wieder in den Speicher zu laden und zu verwenden. Zu einem Projekt können mehrere Analyse-Objekte gespeichert werden, die jeweils andere Wine-Versionen zusammen mit der im Projekt gespeicherten Windows-Version analysieren. Die Verwaltung dieser Analyse-Objekte übernimmt das `ProjectMeta`-Objekt, das jedem `Project` zugeordnet ist. Dieses kann über die Methode `getPm()` von `Project` abgerufen werden. Mit einer Analyse-Instanz a für ein `Project` p zeigt der folgende Code das Abspeichern von a :

```
/* set the Analysis instance for project p
   using Wine 20040813 to a */
p.getPm().setAnalysisForWine("20040813",a);

// save Analysis to DB. Include its Loader.
if (!a.saveResults(true)) {
  System.err.println("Error saving Analysis to DB.");
  }
```

6.6.5 Graph

Abstrahiert man über die technischen Gegebenheiten die das System modelliert, so läßt sich leicht erkennen, daß sich die Strukturen auf denen operiert

wird auch als Graph darstellen lassen. Die formale Definition eines Graphen oder die Diskussion verschiedener Eigenschaften dieser Strukturen sind nicht Gegenstand diesr Arbeit und sollen daher an dieser Stelle nicht diskutiert werden. Die Modellierung als Graph dürfte sich aber nach Ansicht des Autors bei einigen potentiellen Weiterentwicklungen des Systems als interessant erweisen, weswegen dieser Aspekt hier nicht ganz unerwähnt bleiben soll.

Es würde sich bei den Graphen um Abhängigkeitsgraphen von Windows-Software handeln, in dem die Knoten definiert sind durch APIs, die die einzelnen Module zur Verfügung stellen. Kanten in diesen Graphen würden die Relation „importiert von“ modellieren, so daß es sich offensichtlich um gerichtete Graphen handeln würde. Diese Graphen sind zu umfangreich, um sie bildlich darzustellen. Sie sind jedoch durchaus hilfreich dabei, eine Anschauung davon zu erhalten, was genau eigentlich modelliert wird. Vor der Implementierung des in dieser Arbeit dargestellten Verfahrens wurde daher ein erster minimalistischer Prototyp entwickelt, der die Abhängigkeitsstrukturen von Windows-Software in vereinfachter Form in das GML-Format schreibt. Dieses Format kann vom Graphenzeichner yed der Firma yworks[15], einem Spin-Off des Lehrstuhls von Prof. Dr. Kaufmann[16] verarbeitet und dargestellt werden. Abbildung 6.10 zeigt die Darstellung eines kleinen Abhängigkeitsgraphen von Windows-Software, der lediglich Module, nicht jedoch die darin enthaltenen Funktionen berücksichtigt.

6.7 API-Abgleich

Nachdem in den vorangegangenen Abschnitten sowohl die Struktur des Systems als auch seine wichtigsten Mechanismen beschrieben worden sind, soll nun das Augenmerk auf den interessantesten Teil gelegt werden: Die Verabeitung und Auswertung der gesammelten Informationen zum Zweck der Kompatibilitätsanalyse von Windows-Software mit Wine. Zunächst wird beschrieben, wie genau ein API-Abgleich im System abläuft, wobei dies gegebenenfalls auch anhand von Quelltextstellen erläutert wird. Das System macht nach der Analyse zudem einen ersten Vorschlag zur Verbesserung der Konfiguration. Dieser Vorschlag wird als `Configuration`-Objekt gekapselt von der Methode `analyze()` des Analyse-Objekts zurückgegeben.

6.7.1 Analyse-Methode im Analysis-Objekt

In diesem Abschnitt soll mit Unterstützung durch den Quelltext beleuchtet werden, wie das Analyse-Objekt bei der Analyse von Windows-Software zusammen mit einer bestimmten Wine-Version vorgeht. Die Benutzung ins-

[15] Homepage von yworks: `http://www.yworks.de`, zuletzt besucht am 19.10.2004.

[16] Hoempage des Lehrstuhls von Prof. Dr. Kaufmann: `http://www-pr.informatik.uni-tuebingen.de`, zuletzt besucht am 19.10.2004.

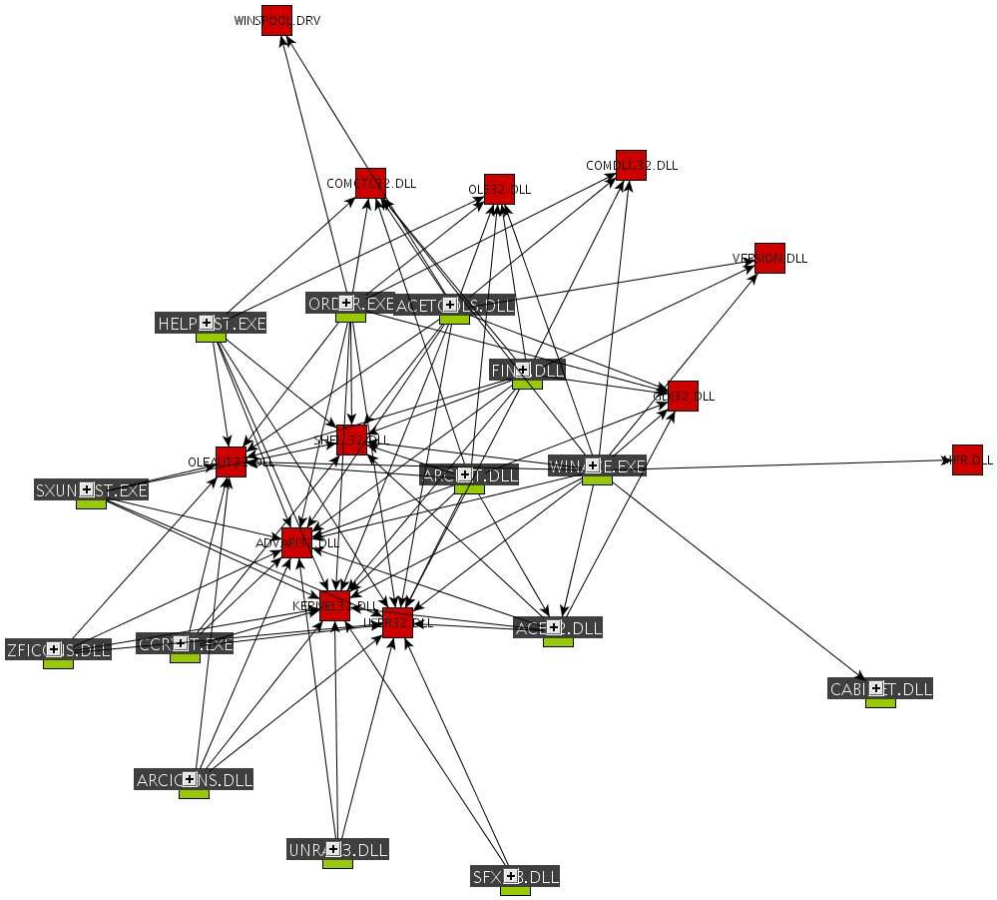


Abbildung 6.10: Modul-Abhängigkeitsgraph einer kleinen Windows-Software. Rot eingefärbt sind Module, die von Wine zur Verfügung zu stellen wären. Der Graph stellt lediglich die Beziehung „benötigt“ zwischen Modulen dar und bezieht die in den Modulen enthaltenen APIs nicht mit in die Darstellung ein, um Übersichtlichkeit zu gewährleisten.

besondere der APIs der Entitäten `Module`, `Project`, und `Loader` läßt sich hieran veranschaulichen.

Initialisierung Ist eine Analyse instanziiert worden, so kann die Methode `analyze()` aufgerufen werden, um die Analyse der zugrundeliegenden Windows-Software mit der angegebenen Wine-Version anzustoßen. Das `Project`-Objekt, das analysiert werden soll, kann vorher vom Programmierer mit einer bestimmten Wine-Version initialisiert werden; es ist aber auch möglich, diese Aufgabe dem Analyse-Objekt zu überlassen. Somit beginnt die Methode `analyze()` wie folgt:

```
// reset Analysis
resetAnalysis();

// start time measuring
tm.start();

/* if Project is not already initialized
   and no initialization process is running,
   do initialisation ourselves */
f ((!p.isInitialized()) && (!p.isRunning()))
  p.initialize(WineVersion);

/* in any case, wait until initialisation
   has finished */
while (!p.isInitialized()) {
  Thread.yield();
  }
```

Mit der Zeile `tm.start()` wird auf eine Hilfsklasse des Systems zugegriffen, den so genannten `TimeMeasurer`. Dieser dient als eine Art Stoppuhr, um die Dauer der Analyse zu messen, und wird mit `start()` bzw. `stop()` in Gang gesetzt und angehalten.

Iteration über Module Die Analyse beginnt nun als nächsten Schritt eine Iteration über alle Module im Projekt. Jedes Modul wird daraufhin untersucht, ob sich seine Importe vom Loader auflösen lassen. Ist dies nicht der Fall, so wird diese gefundenen Problemstelle im Analyse-Objekt gespeichert, um über die API des Systems abrufbar zu sein.

Zunächst werden einige Variablen intialisiert:

```
// Iterator over all modules (DLLs) of the project
```

```
Iterator mI = p.getModules();

// the module currently under analysis
Module m = null;

/* module loaded by loader in order to
   satisfy import of m */
Module loadedModule = null;

// the import to be satisfied by the Loader
ImportedFunction f = null;

// the project's loader instance
Loader l = p.getLoader();
```

Bei Beginn des Iteration über alle Module im System wird zunächst bei jedem gerade betrachteten Modul überprüft, ob seine Initialisierung abgeschlossen ist:

```
// start iteration
while (mI.hasNext()) {

  // currently analyzed module
  m = (Module) mI.next();

  // increment module counter of Analysis
  totalModules++;

  // if m is not yet fully initialized, wait
  while (!m.isInitialized()) {
    Thread.yield();
    }
```

Iteration über Importe Bei jedem Modul, das die Analyse auf diese Weise unter die Lupe nimmt, wird jeder einzelnen Import dieses Moduls betrachtet und versucht, diesen mittels Loader aufzulösen. Gleichzeitig findet eine Zählung der Importe und Exporte der Module statt:

```
// count current module's imports and exports
totalImports += m.imports.size();
totalExports += m.exports.size();
```

Der nun folgende Block versucht, jeden einzelnen Import des gerade betrachteten Moduls mittels des Loaders aufzulösen:

```
// get imports of current module
Iterator i = m.getImports();

// iterate over imports of current module
while (i.hasNext()) {

  // set f to current ImportedFunction
  f = (ImportedFunction) i.next();

  /* ask loader to load module that should
     satisfy the import f */
  loadedModule = l.getImportedModule(m, f.fromModule);

  /* if loadedModule == null, the loader didn't
     find a corresponding module, i.e. it's
     missing */
  if (loadedModule == null) {

    // record which module is missing
    addMissingModule(m, f);

  /* else, module was found. now make sure
     it has finished initialisation */
  } else {
    while (!loadedModule.isInitialized()) Thread.yield();

    /* is this a Wine module? if yes, store that information.
    if (loadedModule.path.equalsIgnoreCase(l.getVirtualWinePath()))
        totalWineModules.put(loadedModule,"nothing");

    /* does the loaded module provide the
       function we need, i.e. the export
       that corresponds to the currently
       analyzed ImportedFunction f ? */
    ExportedFunction ef = loadedModule.hasFunction(f, l);
    if (ef != null) {

      /* Function was found. Now look into its
         implementation state and count it as
         "missing" if the state is below the accept-
```

```
          able minimum, or the state is unknown */
        if (( ef.getImplStatus() >= analysisDepth ) ||
            ( ef.getImplStatus() == Analysis.IMPL_STATE_UNKNOWN)) {
          /* we found the function, but its impl. state
             doesn't suffice */
          addMissingFunction(ef, f.moduleName, loadedModule);
          addIncompleteModule(loadedModule);
        }

        // store WINE API, even if it's sufficiently implemented
        else
          if (loadedModule.path.equalsIgnoreCase(l.getVirtualWinePath()))
           totalWineAPIs.put(ef,"nothing");
      }

      // required function not found <=> ef==null
      else {

        // record the missing function
        addMissingFunction(f, f.moduleName, loadedModule);

        // record the incomplete module
        addIncompleteModule(loadedModule);

      }
    }
  }
```

Abschließend wird die Zeitmessung der Analyse mittels `TimeMeasurer` angehalten und der Status des Analyse-Objekts auf initialized gesetzt. Dem mit der Analyse assoziierten `ProjectMeta`-Objekt werden einige Informationen bekannt gemacht, die die Analyse gewonnen hat. Darunter fallen beispielsweise die Information, ob und wieviele Problemstellen gefunden wurden.

Schließlich wird versucht, einige einfache Probleme in der Konfiguration von Wine für die analysierte Windows-Software automatisch zu lösen, und die entsprechende Konfiguration als `Configuration`-Objekt zurückgegeben. Die Arbeitsweise der Methode `solveMissing()`, die diesen Konfigurationsvorschlag erstellt, ist in Abschnitt 6.7.3 erläutert.

```
// stop time measuring
tm.stop();
```

```
// this Analyis object is now initialized
initialized = true;

/* inform the ProjectMeta object about some
   information we have acquired during analyze() */
updateProjectMeta();

// try to find solutions for missing modules
Configuration c  = solveMissing();

return c;
```

6.7.2 Loader-Vorgehen

Um ein genaueres Verständnis dessen zu erlangen, wie das `Loader`-Objekt versucht, Modulimporte aufzulösen, muß man sich zunächst vor Augen führen, welche Aufgaben dabei genau zu erledigen sind. Anhand der Loader-Methode `getImportedModule()`, die als Parameter ein Modul m und einen seiner Importe f als `ImportedFunction` erhält, soll dies erläutert werden.

Verzeichnis des importierenden Moduls Beim Auflösen eines Modulimports wird zunächst in dem Verzeichnis in dem das importierende Modul abgelegt ist nach einem Modul gesucht, das den Import bereitstellen kann:

```
// get current configuration from Project
Configuration c = p.getConfiguration();

// reference to module being loaded
Module mod = null;

/* try to load from same path as importing
   module */
if (mod == null) {
  mod = getByPath(m.path.toLowerCase(),
                  ImportedModule.toLowerCase());
  }
```

Konfigurierte Suchpfade Zu den standardmäßig gesetzten Suchpfaden (Verzeichnis des importierenden Moduls sowie Windows-Systemverzeichnisse) lassen sich in Wine und unter Windows zusätzliche

konfigurieren. Diese werden im System vom `Configuration`-Objekt verwaltet und abgesucht, wenn im Verzeichnis des importierenden Moduls kein passendes Modul geladen werden konnte:

```
/* Look into additional search paths defined
   by the current Configuration */
if ((mod == null) && (c != null)) {

  // get search paths from config
  Vector v = c.getPaths();

  // if paths are set, iterate over them
  if (v != null) {
    Iterator i = v.iterator();

    // Iteration over the paths
    while (i.hasNext()) {

      // try to load module from current search path
      mod = getByPath((String)i.next(),
                      ImportedModule.toLowerCase());
      if (mod != null) break;
    }
  }
}
```

Configuration-Objekt befragen Falls das gewünschte Modul nicht geladen werden konnte wird auf die gerade aktuelle Wine-Konfiguration zugegriffen um abzufragen, ob der Import durch ein von der Konfiguration hinzugefügtes Modul bereitgestellt wird. Mittels Konfiguration hinzugefügte Module werden behandelt wie Module, die ins Windows-Systemverzeichnis kopiert werden, wobei ein solcher Kopiervorgang ein eventuell dort vorhandenes namensgleiches Modul überschreiben würde.

```
/* try to load module by its name from
   the Configuration, if a Configuration
   exists */
if ((mod == null) && (c != null)) {
mod = c.getAddedModule(ImportedModule.toLowerCase());
}
```

Windows-Systemverzeichnisse Falls in den vorangegangenen Schritten kein zum Import passendes exportierendes Modul geladen werden konnte, werden die Windows-Systemverzeichnisse durchsucht.

```
// Look in Windows/System and Windows/System32 directories
if (mod == null) {
  mod = getByPath(winpath, ImportedModule.toLowerCase());
  }
if (mod == null) {
  mod = getByPath(win32path, ImportedModule.toLowerCase());
  }
```

Durch Konfiguration entfernte Module Ebenso wie es möglich ist mittels Konfiguration bestimmte Module einem Projekt hinzuzufügen, so ist es auch möglich Module zu entfernen. Dies kommt einem Löschen aus dem Dateisystem gleich. Sinnvoll ist dies insbesondere bei Modulen, die von Wine zur Verfügung gestellt werden müssen, weil ihre Windows-Äquivalente mit Wine nicht kompatibel sind. Eine eindeutige Identifizierung von entfernten Modulen erfolgt über ihren Hashwert.

```
/* If the module has been removed by the Configuration,
   we may not load it from the file system (even if it
   exists there), so just in case we did just so,
   set it back to null and ask Wine to provide it: */
if ((c != null) && (mod != null) && (c.isBuiltIn(mod.hash))) {
  mod = null;
  }
```

Wine Falls das gewünschte Modul nicht geladen werden konnte bzw. durch die Konfiguration ein Laden verboten wurde, wird als letzter Schritt überprüft, ob ein passendes Wine-Modul vorhanden ist. Wine-Module werden vom System behandelt wie gewöhnliche PE-Module, die in einem speziellen Pfad – dem virtualWinePath – abgelegt sind.

```
// look if Wine can provide the module
if (mod == null) {
  mod = getByPath(virtualWinePath,
                  ImportedModule.toLowerCase());
  }

// return Module instance (or null if nothing found)
return mod;
```

6.7.3 Konfigurationsvorschlag

Ist eine erste Analyse einer Windows-Software mit einer bestimmten Wine-Version abgeschlossen und alle Informationen somit über die API des Systems und insbesondere des Analyse-Objekts verfügbar, so kann das Analyse-Objekt einen ersten Vorschlag einer günstigen Wine-Konfiguration machen. Dieses leistet die Methode `solveMissing()`.

Die zugrundeliegende Idee ist die, daß von der Analyse möglicherweise einige Module deswegen als fehlend gemeldet werden, weil sie nicht in den Standard-Suchpfaden vorhanden sind. Würde man aber die Suchpfade für Wine in der Konfiguration geeignet setzen, würden diese Module aufgefunden. `solveMissing()` iteriert also über alle als fehlend gemeldeten Module der Analyse und durchsucht das Projekt darauf hin, ob namensgleiche Module vorhanden sind. Wird ein solches Modul gefunden, so wird sein Pfad einer Liste von Suchpfaden hinzugefügt. Aus diesen Informationen wird dann ein `Configuration`-Objekt instanziiert, auf dessen Basis man die Konfiguration manuell verfeinern kann.

Bei vielen Projekten erwies sich in der Praxis diese Funktionalität als nützlich, da insbesondere bei umfangreicherer Software meist zumindest eine kleine Anzahl von Problemen so automatisch gelöst werden kann.

Kapitel 7

Fallbeispiel Prosoz/S-Kommunalsoftware

Um einen Eindruck von Arbeitsweise und Leistungsumfang des Systems zu vermitteln soll in diesem Kapitel als Beispiel eine Untersuchung einer Windows-Anwendung hinsichtlich ihrer Kompatibilität mit Wine durchgeführt werden. Es sei darauf hingewiesen, daß diese Art von Analyse lediglich eine von mehreren möglichen Anwendungen des Systems darstellt. Weitere Anwendungsmöglichkeiten werden im Abschnitt 8.1 vorgestellt.

Als Beispiel wurde eine vergleichsweise komplexe Applikation gewählt, nämlich das Sozialamtsprogramm Prosoz/S der Firma Prosoz. Bei Prosoz/S handelt es sich um ein Programm zur Abwicklung von Abläufen im Sozialamt, insbesondere der Berechnung von Sozialhilfebeiträgen und der Verwaltung der Antragsteller[1]. Es lag für die Analyse in der Version 7.1 vor.

Für die Analyse wurde auf die zum Zeitpunkt der Anfertigung der Arbeit aktuellste Wine-Version zurückgegriffen, die als Download über die Homepage des Projekts zu beziehen ist.

7.1 Vorbereitungen

Bevor sich mit dem System der Abgleich von Wine-Implementierungsstand und API-Anforderungsprofil der Software durchführen läßt, sind einige vorbereitende Schritte zu erledigen. Es wird im Folgenden davon ausgegangen, daß das Programmsystem mitsamt seiner Datenbank frisch installiert auf einem Rechner zur Benutzung vorliegt. Es muß daher noch die Wine-Version, gegen die getestet werden soll, installiert und ihr Implementierungsstand in die Datenbank eingetragen werden. Zudem muß die Windows-Software

[1] Siehe auch `http://www.prosoz.de/sozial/prosozs/index.htm`, zuletzt besucht am 21.10.2004.

installiert werden, um mittels PE Header-Analyse ihr API-Profil in der Datenbank des Systems für den späteren Abgleich abzulegen[2].

7.1.1 Wine-Installation

Die Installation einer Wine-Version auf dem System, auf dem die Analyse durchgeführt werden soll ist aus mehreren Gründen erforderlich. Zum einen wird das im Wine-Projekt enthaltene Tool winedump für die Durchführung der PE Header Analyse benötigt. Es dient dazu, Import- und Export-Tabelle der PE Module zu parsen und den Inhalt dieser Tabellen in Textform auszugeben. Dieser Output wird von der PE Header-Analyselogik verarbeitet und die gewonnene Informationen geeignet in der Datenbank abgelegt.

Des weiteren muß die zu analysierende Windows-Software unter Windows oder unter Wine installiert werden. In der Regel ist es aus Gründen der Einfachheit vorzuziehen, die Installation unter Wine durchzuführen; eine Installation unter Windows und ein nachträgliches Kopieren der Programmdateien und der Registry ist aber grundsätzlich auch möglich. Für das gewählte Beispiel wurde unter Wine installiert.

Download und Installation Der Wine-Quelltext wurde von der Homepage des Projekts heruntergeladen und mit dem Kommando

```
tar -xzf Wine-20041019.tar.gz
```

in das Verzeichnis wine-20041019 entpackt. Kompiliert und in das Zielverzeichnis `/opt/wine/wine-20041019` wird der Quelltext mittels der Befehle

```
cd wine-20041019
./configure --prefix=/opt/wine/wine-20041019
make depend
make
su -c "make install"
```

Um auf diese Wine-Version und ihre Bibliotheken und Module zugreifen zu können müßten verschiedene Umgebungsvariablen gesetzt werden. Konkret wären dies mindestens

- LD_LIBRARY_PATH. Mit dieser Umgebungsvariable wird dem Linker und Loader des Linux-Betriebssystems bekannt gemacht, in welchen Pfaden außer den fest in /etc/ld.so.conf konfigurierten er nach dynamischen Bibliotheken wie z.B. den Wine-Modulen suchen soll.

[2]Die Installation der Windows-Software muß nicht zwingend unter Wine erfolgen. Man kann hierfür auch auf eine unter Windows erstellte Installation zurückgreifen, bei der man lediglich die durch den Installationsprozeß hinzugekommenen oder veränderten Module analysiert. Auf diese Weise kann Wine-kompatible Software, die mit einem Wine-inkompatiblen Installationsprogramm ausgeliefert wird, analysiert und unter Wine betrieben werden.

- PATH. Wine besteht aus zahlreichen Programmen, die alle im Suchpfad des Systems liegen müssen, um gefunden zu werden.

winestart Häufig will man z.B. für Testzwecke mehrere Wine-Versionen gleichzeitig auf einem System installiert haben, ohne daß diese gegenseitig in Konflikt geraten. Zudem möchte man dann leicht zwischen der Benutzung der einzelenen Versionen umschalten können, weshalb eine statische Konfiguration obiger Variablen in Konfigurationsdateien nicht in Frage kommt.

Abhilfe schafft hier das Tool winestart, das vom Autor dieser Arbeit als Hilfsprogramm für den genannten Einsatzzweck entwickelt wurde. Es setzt obige und andere von Wine berücksichtigte Umgebungsvariablen so weit wie möglich automatisch und unterstützt zudem ein Logging des Debug-Outputs von Wine in Dateien, an die Standardausgabe, sowie beides gleichzeitig. Das Programm ist Freie Software unter der GPL-Lizenz und kann im Internet bezogen werden[3]. Im Folgenden wird winestart für den Aufruf von Programmen mit Wine benutzt.

7.1.2 Software-Installation

Für die Installation von Prosoz/S unter Wine werden vorab zwei Komponenten benötigt:

1. MSI. Der Microsoft Installer wird vom Installationsprogramm benötigt und liegt in der passenden Version auf der Installations-CD im Verzeichnis RTS/msi/9x vor. Bei der Installationsdatei handelt es sich um ein in ausführbarer Form vorliegendes CAB-Archiv, das mit dem Tool cabextract unter Linux folgendermaßen ins Windows-Systemverzeichnis entpackt werden kann:

   ```
   cd $HOME/.wine/fake_windows/Windows/System
   cabextract /mnt/cdrom/RTS/msi/9x/instmsia.exe
   ```

2. DCOM98[4]. Die Implementierung von DCOM in Wine ist nicht hinreichend vollständig, so daß hier auf echte Windows-DLLs zurückgegriffen werden muß. Wie bei MSI ist auch der DCOM-Installer ein ausführbares CAB-Archiv und wird folgendermaßen entpackt:

   ```
   cd $HOME/.wine/fake_windows/Windows/System
   cabextract $HOME/downloads/dcom98.exe
   ```

[3] Homepage von winestart: `http://www.david-guembel.de/winestart.html`, zuletzt besucht am 21.10.2004.

[4] Für die Installation wurde Wine so konfiguriert, daß es Windows 98 simuliert. Diese virtuelle Windows-Version gilt in der Community allgemein als die mit dem größten Kompatibilitätsgrad.

Sind diese beiden Komponenten installiert, so kann die Installation der Prosoz/S-Software mit Hilfe von winestart unter der Benutzung der Wine-Version 20041019 wie folgt gestartet werden:

```
cd /mnt/dvd/
winestart --wine-version=20041019 --program=Setup.exe
```

Der Installer kopiert wie unter Windows alle Programmdateien auf die Festplatte und installiert zusätzlich die für die Ausführung der Software nötigen COBOL-Runtime-Module. Für alle Optionen während der Installation wurde die Standardeinstellung gewählt, so daß Prosoz/S auf Laufwerk C: im Verzeichnis PROSOZ und die COBOL-Module im Verzeichnis FJCOBOL installiert werden. Der Installer verlangt nach Beendigung der Installation einen Reboot des Windows-Systems. Dies ist unter Wine mit dem Kommando wineboot zu erreichen:

```
winestart --wine-version=20041019 --program=wineboot
```

7.2 Analyse

Nachdem nun die nötigen vorbereitenden Schritte für eine Untersuchung der Kompatibilität mit Wine durchgeführt wurden, soll im Folgenden eine solche Analyse von Prosoz/S durchgeführt werden. Diese besteht aus drei Teilen:

1. Wine-Abbild. Zunächst wird mittels der in Abschnitt 6.3 erläuterten Wine-Quelltextanalyse alle von Wine zur Verfügung gestellten Funktionen und ihr Implementierungsstand ermittelt und in der Datenbank eingetragen.

2. Software-Abbild. Mittels der in Abschnitt 6.2 erläuterten PE Header-Analyse werden Importe und Exporte eines jeden Moduls der Windows Software – hier Prosoz/S – ermitelt und in der Datenbank eingetragen. Die Gesamtheit aller Module, die zu Prosoz/S gehören, werden dabei zu einem Projekt gruppiert.

3. API-Abgleich. Schließlich lassen sich API-Profil von Prosoz/S und Implementierungsstand von Wine 20041019 mittels einer Analyse im System vergleichen und die Ergebnisse ausgeben und interpretieren.

7.2.1 Wine-Abbild mittels Wine-Quelltextanalyse

Um vom System alle von Wine zur Verfügung gestellten APIs mit ihrem Implementierungsstand ermitteln und in die Datenbank eintragen zu lassen, muß das Kommandozeilenprogramm gdb.pl mit entsprechenden Para-

metern[5] aufgerufen werden. gdb.pl ist die Kommandozeilenschnittstelle des Systems zur PE Header-Analyse und zur Wine-Quelltextanalyse. Es wird der entpackte Quelltextbaum der zu importierenden Wine-Version benötigt. Die allgemeine Syntax von gdb.pl zur Wine-Quelltextanalyse lautet wie folgt:

```
gdb.pl --import-wine --wine-dir=[dir] \
   --wine-version=[version]
```

Im vorliegenden Beispiel startet man die Quelltextanalyse von Wine Version 20041019 folgendermaßen:

```
gdb.pl --import-wine --wine-dir=$HOME/downloads/wine-20041019/ \
   --wine-version=20041019
```

Nachdem dieser Programmaufruf beendet ist, steht die Wine-Version im System für Analysen zur Verfügung.

7.2.2 Software-Abbild mittels PE Header-Analyse

Eine Ermittelung von allen zu Prosoz/S gehörigen bzw. bei der Installation installierten Module mit ihren Importen und Exporten läßt sich ebenfalls mit dem Programm gdb.pl vornehmen. Angegeben werden muß hier der Projektname unter dem die Module des Software in der Datenbank gruppiert werden sollen. Zudem wird der Pfad des Verzeichnisses benötigt, in das die Software installiert wurde, sowie der Pfad des Windows-Systemverzeichnisses. Gegebenenfalls muß auch noch der Pfad des 32-Bit-Systemverzeichnisses (i.d.R. Windows/System32/) angegeben werden. Allgemein ist die Syntax für den Aufruf von gdb.pl zur PE Header-Analyse wie folgt:

```
gdb.pl --create-project \
   --project-name=[name] --project-dir=[dir] \
   [--project-windir=[dir] --project-windir32=[dir]]
```

In unserem Beispiel würde der Aufruf lauten:

```
gdb.pl --create-project \
   --project-name="Prosoz/S 7.1" \
   --project-dir="$HOME/.wine/fake_windows/" \
   --project-windir="$HOME/.wine/fake_windows/Windows/System/"
```

Nach Beendigung dieses Programmaufrufs sind alle zu Prosoz/S gehörigen Module in der Datenbank mitsamt ihrer Importe und Exporte eingetragen und als Projekt „Prosoz/S 7.“ abrufbar.

[5] Es wird an dieser Stelle davon ausgegangen, daß die zu importierende Wine-Version in den ENUM-Feldern der Modul-, Import- und Export-Tabellen der Datenbank vermerkt ist. Ist dies nicht der Fall, so muß diese Eintragung manuell per SQL-Statement einmalig erfolgen.

7.2.3 API-Abgleich

Der eigentlich Abgleich des API-Profils von Prosoz/S und Wine-Implementierungsstand erfolgt im in Java programmierten Teil des Systems. Für die Durchführung der Analyse wird auf das von Stefan Munz programmierte Tool *sysiphus* zurückgegriffen. Dieses benutzt die API des Systems für die Durchführung von Analysen und bietet dabei eine umfassendere graphische Oberfläche als die zu Test- und Forschungszwecken programmierte GUI des Systems.

Loader-Vorinitialisierung Um Analysen zu beschleunigen wurde im System die Möglichkeit geschaffen, Loader-Objekte mit allen zu einer bestimmten Wine-Version gehörenden Modulen vorinitialisiert in der Datenbank abzuspeichern und bei Bedarf wieder in den Speicher zu laden. Dieses Vorgehen ist zwar optional, spart aber im Betrieb ein durchaus erhebliches Maß an Zeit, weswegen dieser Schritt zunöchst mit folgendem Kommando veranlaßt werden soll:

```
java -Xmx192M ganymede.Ganymede -init-loader 20041019
```

Mit dem Parameter -Xmx192M wird der Java VM die Möglichkeit eingeräumt, bis zu 192 MB RAM für sich zu reservieren, was für den Betrieb des Systems erforderlich ist. Zudem müssen für obigen Aufruf noch folgende JAR-Archive und Verzeichnisse in den Classpath eingebunden sein:

- y.jar. Dies ist ein Archiv, das die API zum Graphenzeichner yed der Firma yworks zur Verfügung stellt. Das System enthält Codefragmente, die die Basis einer Darstellung von Analysen als Graph mittels yed in einer späteren Weiterentwicklung bilden.

- mysql-connector-java-3.0.9-stable-bin.jar. Dieses Archiv enthält den JDBC-Konnektor für die verwendete MySQL-Datenbank.

- jfreereport. In diesem Verzeichnis finden sich die kompilierten Klassen des freien Frameworks JFreeReport zur Report-Generierung.

- junit.jar und jfcunit.jar. Sysiphus benötigt JUnit für einige JUnit-basierte Tests.

- poi-2.0-RC1-20031102.jar, pixie-0.8.1.jar, itext-1.0.0.jar, bsh-1.2b6.jar, gnujaxp.jar. Diese Pakete werden von JFreeReport für verschiedene seiner Funktionalitäten benötigt.

Das System meldet eine korrekte Abspeicherung eines vorinitialisierten Loaders mit der folgenden Ausgabe:

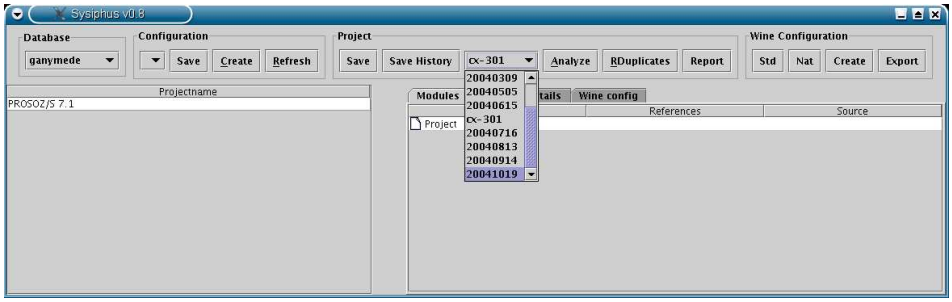


Abbildung 7.1: GUI des Programmes sysiphus. Links eine Liste alle im System vorhandener Projekte und Datenbanken. Durch Auswahl der Wine-Version im Bereich „Project“ und den Klick auf ein Projekt in der Liste links wird die Analyse des gewählten Projekts mit der gewählten Wine-Version gestartet.

```
Ganymede 0.1.0
.. creating empty Loader instance
.. setting Wine modules
.. waiting for Wine modules to finish initialize()
.. saving pre-init. loader to db
..saving returned true
```

sysiphus Der Start des in Java programmierten *sysiphus* wird mit folgendem Kommando veranlaßt:

```
java -Xmx192M install.Sysiphus
```

Danach baut sich die in Abbildung 7.1 dargestellte GUI auf. Durch Auswahl der gewünschten Wine-Version und einen Klick auf den Projektnamen der zu analysierenden Software wird eine Analyse gestartet. Dabei werden eventuell in der Datenbank als Objekte bereits abgespeicherte Analyse-Ergebnisse automatisch berücksichtigt und verwendet. Nimmt man – so wie in diesem Beispiel – die Analyse zum ersten Mal vor, so liegen solche Analyse-Objekte in der Datenbank noch nicht vor.

Analyse-Ergebnisse Nach Beendigung der Analyse zeigt *sysiphus* in mehreren Karteireitern verschiedene Informationen über das Projekt und das Ergebnis der Analyse. Im einzelnen sind dies

- Modulliste. Im Karteireiter „Modules“ werden alle im Projekt vorhandenen Module sowie die gerade aktive Konfiguration aufgelistet. Abbildung 7.2 zeigt diese Liste für Prosoz/S.

- Diagnose. Der Karteireiter „Report“ stellt alle von der Analyse ermittelten Problemstellen dar. Dabei werden die problembehafteten Module in fehlende und unvollständige eingeteilt. Fehlende Module

sind solche, die der Loader in der gesetzten Konfiguration nicht laden konnte, z.B. weil ein Suchpfad nicht korrekt gesetzt war oder weil kein passendes Modul vorhanden war. Unvollständige Module sind Module, die zwar aufgefunden und geladen werden konnten, jedoch nicht alle benötigten Funktionen exportieren. Typischerweise sind dies Wine-Module mit unvollständiger oder fehlender Implementierung einiger APIs. Abbildung 7.3 zeigt die gefundenen Problemstellen nach der Analyse von Prosoz/S.

- Details. Dieser Karteireiter enthält weitere Meta-Informationen über die Software, die in der Datenbank gespeichert sind. Diese Informationen lassen sich editieren und über die API des Systems in die Datenbank schreiben.

- Wine-Konfiguraton. *sysiphus* dient dazu, aus den Analyseergebnissen die bestmögliche Wine-Loaderkonfiguration zu erstellen. Diese wird auf Knopfdruck im Karteireiter „Wine-Config“ angezeigt

Es empfiehlt sich, diese fertige Analyse in der Datenbank abzuspeichern, damit sie später von dort wieder geladen werden kann. Dies bietet einen erheblichen Geschwindigkeitsvorteil gegenüber der vollständigen Analyse. Insbesondere bei solch umfangreichen Softwareprodukten wie Prosoz/S, bei dem mit 240 MB Installationsgröße auf der Festplatte und gut tausend Modulen mit Sicherheit einige hunderttausend Modulabhängigkeiten zu untersuchen sind, ist der Performancevorteil enorm[6]. Das Abspeichern erfolgt über einen Klick auf den „Save“-Button im Bereich „Project“.

Konfigurationen Nach Beendigung der Analyse von Prosoz/S erstellt das System automatisch einen Vorschlag für eine Verbesserung der Konfiguration durch das Setzen geeigneter Suchpfade. Für ein Projekt kann eine beliebige Zahl von Konfigurationen angelegt und in der Datenbank gespeichert werden. Probehalber wird der Konfigurationsvorschlag in die neue Konfiguration „Testkonfiguration“ übernommen. Abbildung 7.4 zeigt den entsprechenden Dialog in *sysiphus*.

sysiphus unterstützt in der vorliegenden Version noch nicht das Hinzufügen oder Entfernen von Modulen aus dem Projekt. Hinzufügen bzw. entfernen ließen sich Module modOne und modTwo unter der Benutzung der API des Systems mit folgendem beispielhaften Programmcode, der anschließend die Konfiguration auch gleich in der Datenbank speichert:

```
// c is a Configuration
// modOne and modTwo are of type Module
```

[6]Eine vollständige Analyse von Prosoz/S auf einem Pentium IV mit 2 GHz dauert ca. acht Minuten, das Laden einer gespeicherten Analyse in den Speicher unter zehn Sekunden.

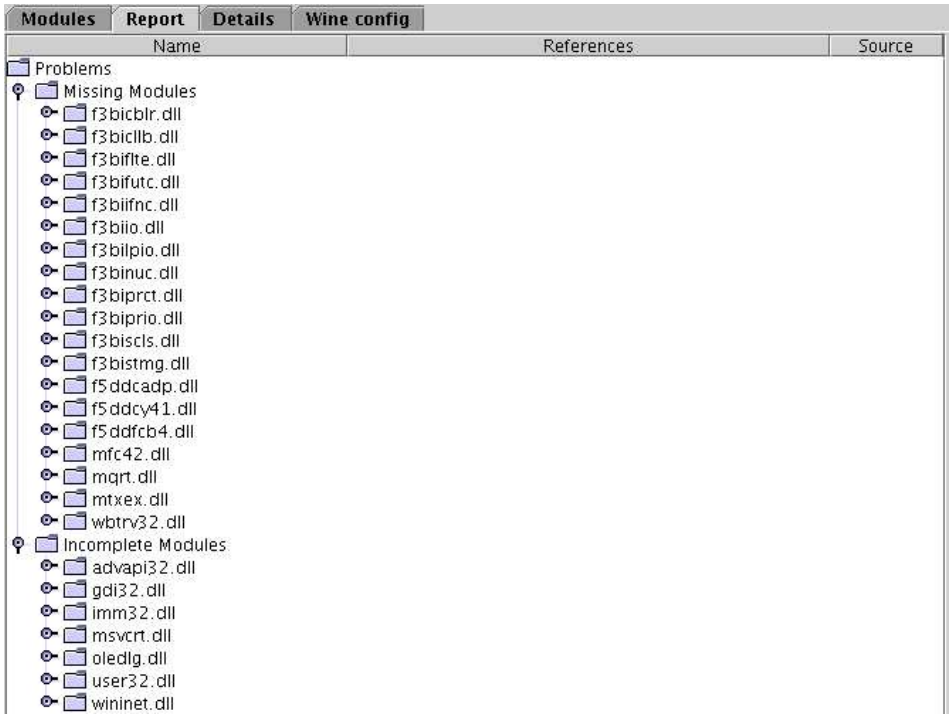


Abbildung 7.2: Screenshot von sysiphus nach Beendigung einer Analyse von Prosoz/S mit Wine Version 20041019.

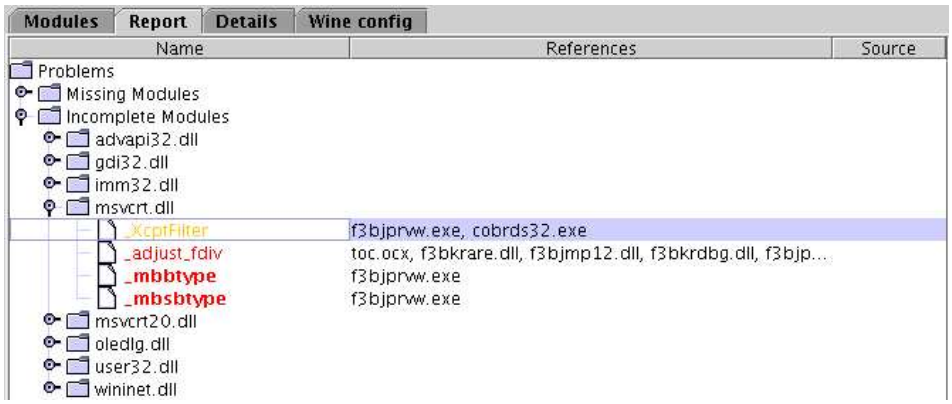


Abbildung 7.3: Screenshot von sysiphus nach Beendigung einer Analyse von Prosoz/S mit Wine Version 20041019. Angezeigt werden alle gefundenen Problemstellen. Es erfolgt eine Gruppierung nach Modulen und Art des gefundenen Problems.

Abbildung 7.4: Dialog zum Anlegen einer Konfiguration in sysiphus. Es können Suchpfade per Konfiguration gesetzt werden. Das Hinzufügen und Entfernen von Modulen mittels Konfigurationen, das über die API des Systems möglich ist, wird von der GUI noch nicht unterstützt.

```
// add modOne to Project
c.addModule(modOne);

// remove modTwo from Project
c.removeModule(modTwo);

// save Configuration into DB
c.saveToDatabase();
```

In *sysiphus* kann die neu erstellte Konfiguration durch einen Klick auf „Save“ in der Datenbank abgelegt werden und taucht dann auch in der Liste der Konfigurationen als Auswahlmöglichkeit auf.

Verbesserung des Ergebnisses Es empfiehlt sich, durch eine Re-Analyse des Projekts mit der neuen Konfiguration zu testen um zu sehen, ob sich die Analyseergebnisse dadurch verbessern. Durch Auswahl der Konfiguration in der Konfigurationsliste, Klicken auf „Refresh“ und anschließend „Analyze“ wird diese Re-Analyse angestoßen. Wie in Abbildung 7.5 ersichtlich, hat sich so die Zahl der problembehafteten Module deutlich reduziert.

7.3 Interpretation der Ergebnisse

Fehlende Module Die Interpretation der gewonnenen Ergebnisse würde man sinvollerweise Modul für Modul durchführen, wobei den als fehlend

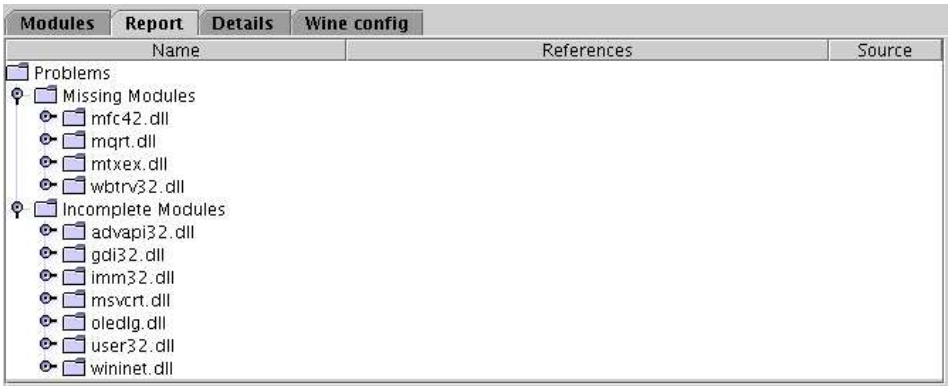


Abbildung 7.5: Anzeige der Problemstellen in Prosoz/S nach Setzen von Suchpfaden. Die Pfade wurden von der Analyse als Konfigurationsvorschlag ausgegeben. Die Zahl der Problemstellen hat sich dadurch sichtbar verringert.

gemeldeten Modulen eine besondere Rolle zukommt. In einer Installation unter Windows, die produktiv eingesetzt wird, dürfen keine Module fehlen, es sei denn das Installationsprogramm hat fälschlicherweise benötigte Module nicht installiert. Daher sind als fehlende gemeldete Module meistens solche, die unentgeltlich z.B. von Microsoft zum Download bereitgesetellt werden bzw. sich auf der Installations-CD des Programmes finden und von dort kopiert werden können:

- mfc42.dll. Dieses Modul wird mit Prosoz/S auf der Installations-CD ausgeliefert und kann einfach ins Windows-Systemverzeichnis kopiert werden.

- mqrt.dll. Hierbei handelt es sich um die Microsoft Message Queue, die als Bestandteil des Personal Web Servers unter Windows verfügbar ist. Dieses Modul findet sich z.B. auf einer Windows 98-CD im Verzeichnis add-ons/pws, wo auch eine weitere als fehlend gemeldete DLL (mtxex.dll) liegt.

- wbtrv32.dll. Diese Datei ist Bestandteil des BTrieve Client Engine, die vom Hersteller Pervasive[7] heruntergeladen werden kann.

Aus technischer Sicht sind diese Problemstellen damit als gelöst zu betrachten. Es bleibt jedoch zu bemerken, daß selbstverständlich die für die einzelnen Dateien geltenden Lizenzbestimmungen zu beachten sind. Während mfc42.dll noch mit der Prosoz/S-Software ausgeliefert wurde und damit unproblematisch sein dürfte, wird für die Module, die Bestandteil des Personal Web Servers sind wohl eine Windows-Lizenz benötigt.

[7]Homepage von Pervasive: `http://www.pervasive.com/`, zuletzt besucht am 2.11.2004.

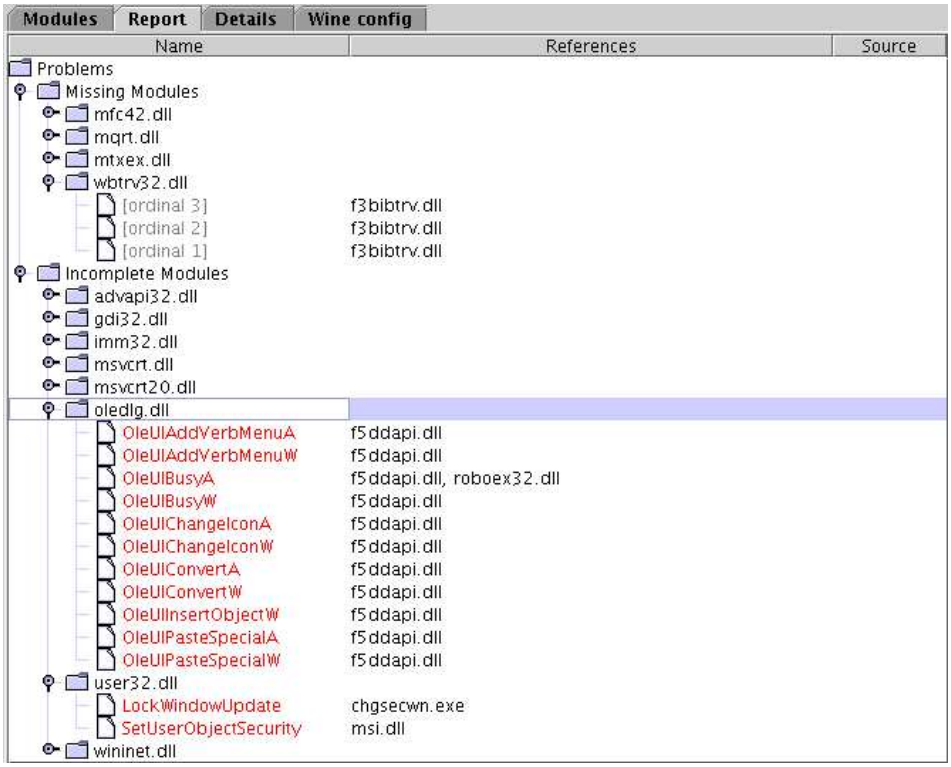


Abbildung 7.6: Detaillierte Anzeige der Problemstellen in Prosoz/S. Für jedes Modul werden die problematischen APIs angezeigt und aufgelistet, von welchen Modulen diese Problemstellen referenziert werden.

Unvollständige Module Bei den als unvollständig gemeldeten Modulen liegt die Problematik etwas anders: Hier sind Implementierungslücken in Wine die Ursache für die gemeldeten Problemstellen. Teilweise lassen sich diese aber auch mittels Windows-Modulen lösen:

- advapi32.dll. Die Detailansicht in *sysiphus* zeigt, daß die Funktionen dieser DLL vornehmlich vom Microsoft Installer benutzt werden. Einige Exporte werden aber auch von Modulen der mit installierten COBOL Runtime Engine benötigt. advapi32.dll kann nicht ohne Weiteres nativ verwendet werden.

- gdi32.dll. Dieses Modul kann nicht nativ verwendet werden, man ist also auf die Wine-Implementierung angewiesen. Es werden insgesamt 18 APIs aus gdi32.dll benötigt, davon allerdings nur drei von mehr als einem Modul.

- imm32.dll. Aus diesem Modul werden zahlreiche unvollständige APIs benötigt. Es sollte und kann nativ verwendet werden.

- msvcrt.dll. Dieses Modul ist ebenfalls auf der Prosoz/S Installations-CD vorhanden und kann von dort einfach ins Windows-Systemverzeichnis der Installation kopiert werden. Es kann dann nativ verwendet werden. Die einzige in msvcrt20.dll verwendete unvollständige API wird auf eine API in msvcrt.dll geforwardet, so daß sich dieses Problem durch die native Verwendung von msvcrt.dll lösen läßt.

- oledlg.dll. Auch hier kann die native Version verwendet werden, um die Lücken im Implementierungsstand von Wine zu überbrücken.

- user32.dll. Diese DLL kann nicht nativ verwendet werden, es muß also auf jeden Fall auf ihre Nachimplementierung zurückgegriffen werden. Es werden jedoch nur zwei unvollständige APIs aus diesem Modul benötigt, und jede wird lediglich ein Mal referenziert, d.h. von einem anderen Modul importiert.

- wininet.dll. Diese DLL kann nativ verwendet werden.

Welche DLLs nativ verwendbar sind bzw. wann für ein Modul zwingend auf die Wine-Implementierung zurückgegriffen werden muß ist zum Teil in der Wine-Dokumentation[8] aufgeführt. Wird man dort nicht fündig, läßt sich auf Erfahrungsberichte anderer Benutzer aus dem Internet oder der Applikations-Datenbank des Wine-Projekts[9] oder auf eigene Erfahrungen zurückgreifen.

[8] Wine-Dokumentation zur Konfiguration des Loaders (DllOverrides-Abschnitt in der Konfigurationsdatei): `http://www.winehq.com/site/docs/wine-user/config-dll`, zuletzt besucht am 2.11.2004.

[9] Wine Application Database: `http://appdb.winehq.org/`, zuletzt besucht am 2.11.2004.

Abbildung 7.7: Erstellung einer optimalen Wine-Konfiguration von Wine mittels sysiphus auf der Basis der ermittelten Analyseergebnisse.

Wine-Konfiguration Aus technischer Sicht fällt das Analyseergebnis für Prosoz/S recht positiv aus, wenn man davon ausgeht, daß Windows 98-DLLs verwendet werden können um die Implementierungslücken von Wine zu füllen. Die Zahl der Problemstellen ist überschaubar, und die nicht durch native Verwendung von Modulen auflösbaren Probleme werden nur von wenigen Modulen benötigt. Auffällig ist, daß viele problembehaftete APIs vom Microsoft Installer, jedoch nicht von Prosoz/S selbst verwendet werden. Die meisten nicht durch Verwendung von Windows-Modulen lösbaren Probleme liegen in advapi32.dll und gdi32.dll.

Mittels der Implementation des in dieser Arbeit vorgestellten Verfahrens und dem darauf aufsetzenden GUI-Programm *sysiphus* kann innerhalb von Minuten eine Wine-Konfiguration erstellt werden, die so für einen Test unter Produktivbedingungen eingesetzt werden kann. Abbildung 7.7 zeigt die Erstellung der Wine-Konfiguration.

7.4 Übernahme in Testbetrieb

Es ist für gewöhnlich angebracht, die Applikation unter Wine unter Bedingungen, die einem Produktivbetrieb möglichst nahe kommen, zu testen. Insbesondere ist dies sinnvoll, weil außer der von der Analyse erkannten Klasse von Problemen auch anders geartete auftreten könnten. Dazu zählen beispielsweise Fehler in der Windows-Applikation oder auch in Wine möglicherweise nicht korrekt implementierte APIs[10]. Prosoz/S soll hier jedoch lediglich als ein Beispiel für die Durchführung einer Applikations-Analyse

[10]Wie bereits im Abschnitt 4.1 angesprochen sind z.B. manche APIs von Microsoft fehlerhaft dokumentiert. Gelegentlich sind auch die Implementierungen unter Windows mit subtilen Fehlern behaftet, die unter Wine in der Regel ebenfalls nachgebildet werden müssen.

mittels des Systems dienen. Daher wurde hier auf einen Nutzertest verzichtet und lediglich verifiziert, daß sich alle installierten Programme mit der gewählten Konfiguration fehlerfrei starten lassen.

Um die Prosoz/S-Applikation in einen solchen Testbetrieb zu übernehmen, wären die folgenden Schritte zu unternehmen:

- Module. Diejenigen Module, bei denen nicht die Wine-Nachimplementierung verwendet werden soll, bzw. die als fehlend gemeldeten Module, die aus dem Internet und anderen Quellen bezogen wurden, müssen ins Windows-Systemverzeichnis kopiert werden. Konkret sind das in diesem Fall also imm32.dll, msvcrt.dll, oledlg.dll, mfc42.dll, msqrt.dll, mtxex.dll, und wbtrv32.dll.

- Konfiguration. Man kopiert die von *sysiphus* automatisch erstellte Loader-Konfiguration in die Wine-Konfigurationsdatei. Außerdem fügt man den Suchpfad, den der vom System nach der Analyse automatisch erstellte Konfigurationsvorschlag enthielt (siehe Abbildung 7.4), dem Wine-Loadersuchpfad hinzu. Dies geschieht mittels dem in Wine enthaltenen Registry Editor regedit. Der Suchpfad ist im Schlüssel

  `HKEY_CURRENT_CONFIG\Environment\PATH`

  gespeichert[11].

Laut auf der Installations-CD vorhandenen Installationsanleitung müssen vor dem Betrieb der Prosoz/S-Software noch Komponenten im System registriert werden. Dies erledigt die Datei SWINREG.BAT, die mit installiert wurde und folgenden Inhalt hat:

```
regsvr32 pswinocx.dll
regsvr32 toc.ocx
```

Wine beherrscht das Ausführen von Batchdateien nicht. Es müssen also noch einmalig folgende Kommandos aufgerufen werden, um diese Komponenten zu registrieren:

```
cd /home/guembel/.wine/fake_windows/Windows/System
winestart --wine-version=20041019 \
      --program="regsvr32 PSWINOCX.dll"
winestart --wine-version=20041019 \
      --program="regsvr32 Toc.ocx"
```

Nun kann das Programm mit folgendem Kommando gestartet und dann getestet werden.

[11] Der Pfad muß in „Windows-Notation“ hinzugefügt werden, d.h. als Pfad mit Laufwerksbuchstabe. Mit der üblichen Standard-Konfiguration von Wine wäre der Pfad also C:\FJCOBOL\COBOL.

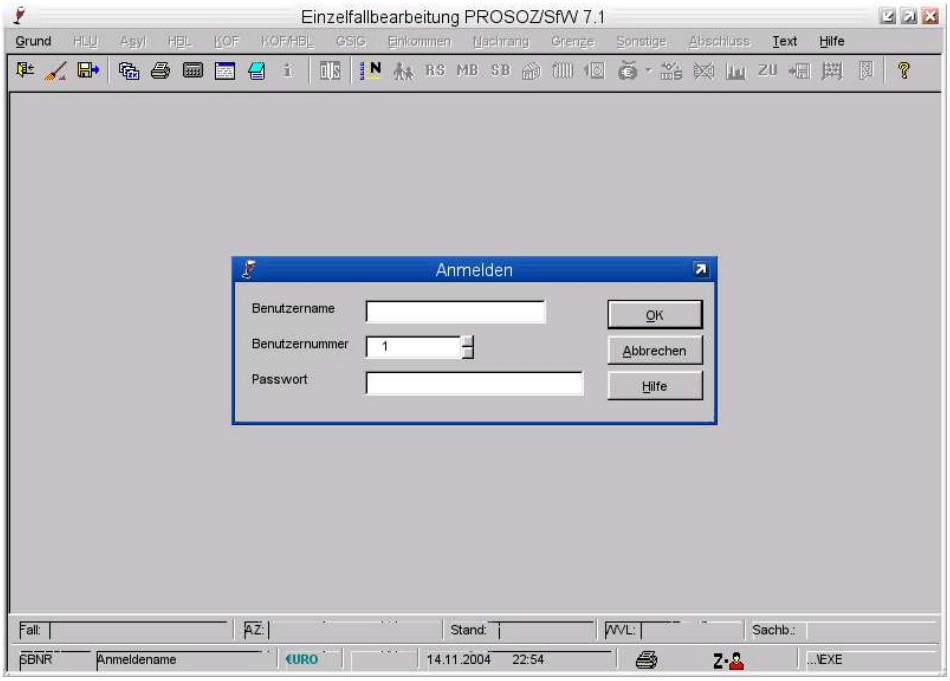


Abbildung 7.8: Anmeldedialog von Prosoz/S unter Linux/Wine.

```
cd /home/guembel/.wine/fake_windows/PROSOZSW/EXE
winestart --wine-version=20041019 --program="./PROSOZ.exe"
```

Abbildung 7.8 zeigt den sich dann aufbauenden Login-Bildschirm von Prosoz/S unter Linux/Wine.

Kapitel 8

Diskussion und Ausblick

Nachdem in den vorangegangenen Kapiteln das entwickelte Verfahren und eine beispielhafte Anwendung vorgestellt wurden, stellt sich nun die Frage nach weiteren Anwendungsmöglichkeiten. Es soll im Folgenden zudem beleuchtet werden, in welchem Umfang und in welcher Weise sich eine Migration von Windows-Applikationen auf Linux/Wine durch das Verfahren vereinfachen läßt.

Verfolgt man das in der Arbeit zugrunde gelegte Szenario einer fiktiven Fachanwendungs-Migration der Stadt Böblingen weiter, so ergeben sich eine Reihe von offenen Fragen und Problemstellungen. Es wird daher im Folgenden eine übersichtsartige Darstellung dieser Punkte erfolgen, die bei einer realen Migration zunächst einer Klärung bedürften.

Neben diesem Ausblick sollen im letzten Teil dieses Kapitels sinnvolle technische Erweiterungen des entwickelten Systems angesprochen werden.

8.1 Anwendungsmöglichkeiten

Die denkbaren Anwendungen des entwickelten Verfahrens lassen sich aus technischer Sicht grob in zwei Kategorien unterteilen[1]:

1. „Black Box“-Migration. Hierunter fällt eine Migration einer Fachanwendung, bei der der Quelltext der Anwendung nicht zur Verfügung steht. Dies ist z.B dann der Fall, wenn eine Anwendung ohne Unterstützung durch den Softwarehersteller migriert werden muß. Ebenso in diese Kategorie fällt eine Untersuchung, ob eine Migration mit Linux/Wine überhaupt machbar ist oder wegen mangelnder Kompatibilität von Wine und Fachanwendung verworfen werden muß.

2. Portierung. Steht der Quelltext der Applikation zur Verfügung, so bietet es sich an, diesen so zu modifizieren, daß z.B. in Wine nicht im-

[1] Auf die spezifische Anwendung des Verfahrens auf kommunale Software geht [Gümbel 2004] genauer ein.

plementierte Windows-Betriebssystemaufrufe durch Unix-Äquivalente ersetzt werden. Mit Hilfe der winelib läßt sich so u.U. leicht und schnell eine Linux-Version der Software erstellen.

8.1.1 „Black Box“-Migration

In Kapitel 7 wurde die Anwendung des Verfahrens anhand der Analyse einer kommunalen Fachsoftware (Prosoz/S) vorgeführt. Eine solche „Black Box“-Analyse von Fachanwendungen ist immer dann von Bedeutung, wenn Programme ohne Unterstützung durch den Hersteller – der über den Quelltext der Applikation verfügt – auf ihre Ausführbarkeit unter Linux/Wine untersucht werden sollen.

Ebenfalls ähnlich vorgehen würde man, wenn man die grundsätzliche technische Kompatibilität einer Software mit Linux/Wine bewerten müßte, ohne daß ihr Quelltext für diese Analyse vorhanden ist.

Projektablauf Ein typischer Ablauf solcher Projekte würde sich in drei Phasen gliedern, die Abbilding 8.1 schematisch veranschaulicht:

1. Konfiguration und Test. In dieser ersten Phase würde ein mit Wine vertrauter Experte die Software bzw. Wine konfigurieren und nach technischen Gesichtspunkten testen. Auf Basis dieser Testergebnisse würde die Konfiguration verbessert und erneut getestet, bis das Programm dem Experten als stabil genug für einen Funktionstest durch den Benutzer erscheint.

2. Nutzertest. Das Programm würde anschließend in einer Testumgebung installiert und durch den Endbenutzer getestet. Diese Phase muß durch den Experten betreut werden, um eventuell auftretende technische Fehlfunktionen falls möglichst zu beseitigen. Ein Nutzertest ist bei Fachapplikationen unerläßlich, da ein Techniker die korrekte Funktionsweise einer Fachsoftware z.B. zur Sozialhilfeberechnung nur bedingt einschätzen kann.

3. Produktivbetrieb. Sind auch die Nutzertests zufriedenstellend verlaufen, kann das Programm in den Produktivbetrieb übernommen werden.

Verbesserung Der genannte Projektablauf bleibt auch bei einem Einsatz des Verfahrens grundsätzlich bestehen. Es ergeben sich jedoch in folgende Verbesserungen:

1. Die Konfigurations- und Testphase läßt sich durch die Analyse der Applikation mit dem Verfahren erheblich verkürzen. Eine optimale Konfiguration läßt sich weitgehend automatisch erstellen und kann

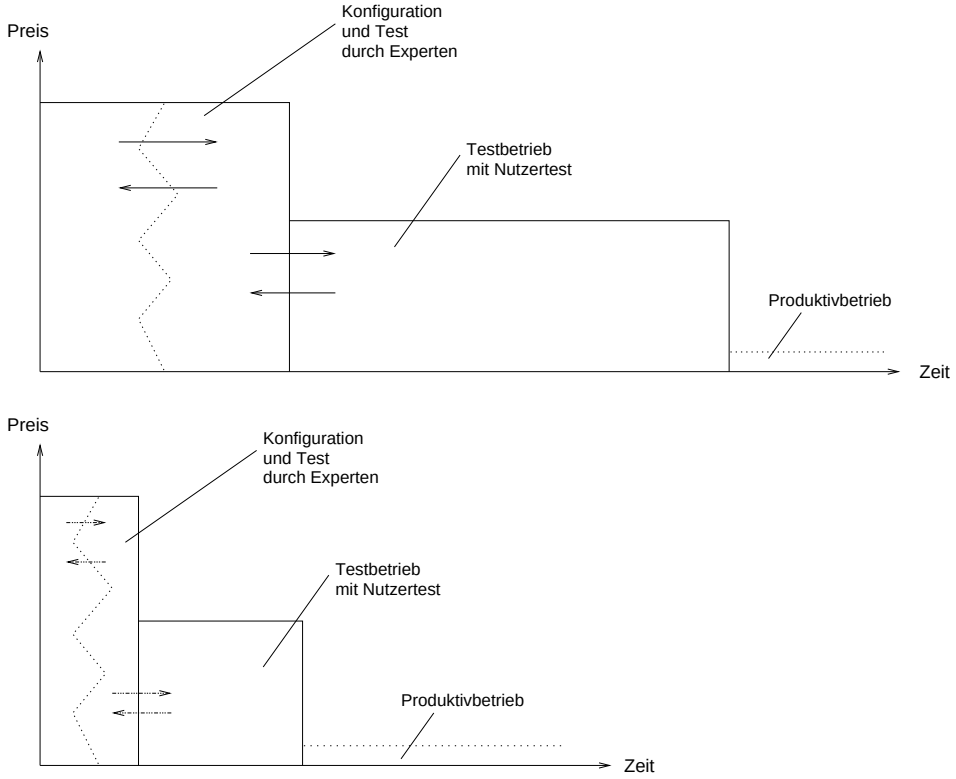


Abbildung 8.1: Projektablauf einer „Black Box"-Migration, bei der der Quelltext der Applikation nicht zur Verfügung steht. Gegenübergestellt wird schematisch der Ablauf mit (unten) und ohne (oben) Unterstützung durch das vorgestellte Verfahren.

vom Experten sehr schnell getestet werden. Eine Überprüfung hinsichtlich der grundsätzlichen technischen Kompatibilität der Software mit Linux/Wine, die im Wesentlichen mit der Durchführung der ersten Projektphase gleichzusetzen ist, kann deutlich effizienter durchgeführt werden.

2. Der Zeitraum des Tests durch den Endbenutzer kann deutlich verkürzt werden, da die vom Experten erstellte Konfiguration in der Regel kaum verbessert werden muß, und damit in der Nutzertest-Phase aus technischer Sicht selten weitere Fehler auftreten. Es kann vom Nutzer zudem gezielt auf gewisse potentielle Schwachstellen getestet werden, weil diese durch das Analyseergebnis dem Experten weitgehend bekannt sind.

3. Eine Überprüfung hinsichtlich der grundsätzlichen technischen Kompatibilität der Software mit Linux/Wine, die im Wesentlichen mit der Durchführung der ersten Projektphase gleichzusetzen ist, kann deutlich effizienter durchgeführt werden.

4. Wird eine Applikation vom Experten für kompatibel befunden, so ist das Risiko eines Abbruchs des Projekts in der Nutzertestphase zudem bei Benutzung des Verfahrens geringer, da durch gezielteres Testen und im vorhinein optimierte Wine-Konfiguration eventuelle kritische Fehler schneller gefunden werden können.

5. Bewertet der Experte die Applikation für nicht migrierfähig, so kann ihr Profil dennoch in der Datenbank verbleiben und zukünftig jeweils mit der neuesten Wine-Release analysiert werden. Ergeben sich Verbesserungen, so kann gezielt erneut getestet werden, ob die vorhandenen Probleme beseitigt sind.

Praxistest Da das Szenario einer Linux-Migration der Stadt Böblingen, das dieser Arbeit zugrunde liegt jedoch nur fiktiv war, konnte kein umfassender Testbetrieb mit kommunaler Software durchgeführt werden. Aus einem Migrationsprojekt an der Fakultät für Chemie der Universität Tübingen sind dennoch einige Erfahrungswerte vorhanden. Die dort im Einsatz befindlichen Chemie-Fachanwendungen wurden dabei mit Hilfe des hier vorgestellten Verfahrens migriert[2]. Tabelle 8.1 zeigt eine Liste der Applikationen.

Der Projektablauf der Migration folgte dem erläuterten Schema, wobei Applikationen die in der ersten Phase für nicht hinreichend kompatibel befunden wurden ausgeschlossen wurden. Die als migrierbar bewerteten Fachanwendungen wurden auf zwei Testrechnern installiert und dort von

[2]Die hier präsentierten Daten wurden freundlicherweise vom Projektleiter der Migration, Herrn Stefan Munz, zur Verfügung gestellt.

Applikation	Bewertung (Phase eins)	Benutzertest (Phase zwei)
Atoms5	migrierbar	bestanden
C-Design	nicht migrierbar	–
ChemWindows	nicht migrierbar	–
Chemdesk	nicht migrierbar	–
Chempute	migrierbar	bestanden
Diamond	fehlerbehaftet	–
ISCD	(migrierbar)	–
IsisDraw	nicht migrierbar	–
Micrografx	nicht migrierbar	–
Office XP	migrierbar	bestanden
PCW	migrierbar	bestanden
WINGX	migrierbar	bestanden
WinPlotr	migrierbar	bestanden

Tabelle 8.1: Liste der für die Migration vorgesehenen Chemie-Fachapplikationen. Die Applikation ISCD wurde wegen auslaufender Lizenz von der Migration ausgeschlossen.

den Endbenutzern in Augenschein genommen. Von den insgesamt 13 Applikationen wurden sieben in der ersten Phase als migrierbar eingestuft. Nach Ausschluß einer dieser Anwendungen aus lizenzrechtlichen Gründen wurde in Phase zwei ein Benutzertest mit den verbleibenden sechs durchgeführt.

Alle diese Anwendungen haben den Nutzertest bestanden und werden zukünftig im Computerpool in den Produktivbetrieb übernommen. Als Fazit des Projektes läßt sich Folgendes festhalten:

- Alle in Phase eins als migrierbar bewerteten Applikationen haben den Nutzertest bestanden.
- 50% der Applikationen[3] ließen sich ohne Modifikation des Quelltextes migrieren („Black Box“-Migration). Das Programm *Diamond* hat bei ansonsten guter Funktionalität einen Darstellungsfehler, der bereits in Phase eins entdeckt wurde.
- Aus technischer Sicht interessant sind die Daten, die in Tabelle 8.2 aufgelistet sind[4]: Die Zahl der benötigten APIs schwankt zwischen den Fachanwendungen stark, von 273 bis 1539.
- Selbst Applikationen wie Microsoft Office XP, die mit 1539 APIs aus 23 verschiedenen Wine-Modulen das anspruchsvollste Programm im

[3]Die Applikation ISCD wurde wegen Lizenzproblemen von der Migration ausgeschlossen und wurde deswegen hier nicht mitgezählt.

[4]Die Applikation PCW ist offenbar statisch gelinkt bzw. importiert alle benötigten Funktionen über expliziten Import.

Applikation	Wine-Module	APIs	Problembehaftet
Atoms5	*8*	*526*	*0/9/1*
C-Design	6	273	0/1/1
ChemWindows	4	301	0/2/0
Chemdesk	12	727	1/10/3
Chempute	*7*	*304*	*0/1/0*
Diamond	22	1184	11/43/7
IsisDraw	11	730	0/9/3
Micrografx	22	1381	11/51/8
Office XP	*23*	*1539*	*30/72/8*
PCW	*(0)*	*(0)*	*(0/0/0)*
WINGX	*8*	*1032*	*6/21/2*
WinPlotr	*23*	*1232*	*11/36/7*

Tabelle 8.2: Technische Übersicht der für die Migration vorgesehenen Chemie-Fachapplikationen. Kursiv markiert sind für migrierfähig befundene Programme. Die Spalte „Problembehaftet“ enthält durch Schrägstriche getrennt die Werte für die Zahl der benutzten Wine-APIs, die als Spec-Stubs/Stubs/Semi-Stubs unter Wine-20041019 erkannt wurden. In lediglich einem Fall konnte für eine Wine-API kein Implementierungsstand ermittelt werden.

Test war, konnte erfolgreich migriert werden. Gleichzeitig ergaben sich aber auch bei kleineren Programmen Fehler, die einen Ausschluß von der Migration notwendig machten. Dies ist unter Anderem darauf zurückzuführen, daß die gefundenen Problemstellen im Falle von Office gut durch die Verwendung von Windows-Modulen geschlossen werden konnten, während dies bei anderen Applikationen nicht der Fall war.

8.1.2 Portierung

Andere Möglichkeiten bieten sich jedoch, wenn der Quelltext der Applikation zusätzlich zum Kompilat zur Verfügung steht. Dies ist z.B. dann der Fall, wenn ein Softwarehersteller möglichst effizient und kostengünstig eine Linux-Version seines Produktes anbieten will. In diesem Fall ergeben sich folgende Verbesserungen durch das Verfahren:

- Da das Verfahren auch eine Zuordnung der problembehafteten API-Aufrufe zu dem Modul das diese Aufrufe benötigt beinhaltet, kann mit Hilfe von Analyseergebnissen gezielt Quelltext modifiziert werden. Für eine Portierung mittels winelib könnten beispielsweise diese Aufrufe durch Unix-Äquivalente ersetzt werden.

- Durch eine Analyse einer Teilmenge von Programmmodulen können beispielsweise Teilsysteme einzeln analysiert und portiert werden.

- Eine gezielte Nachimplementierung von problembehafteten APIs in Wine ist möglich.

Es ist ebenfalls anzumerken, daß auch bei einer Portierung eine sinnvolle Strategie mit Hilfe des Analyseergebnisses im Vorhinein leicht festzulegen ist. So können Windows-Applikationen sehr effizient nach Linux portiert werden. Unter Umständen ist es sogar möglich, eine einzige Codebasis für Windows- und Linux-Version zu pflegen. Dies ist insbesondere bei Versionsupgrades der Software hilfreich.

8.2 Fachanwendungsmigration in Böblingen

Angesichts von erfolgreichen Migrationsprojekten in Behörden wie beispielsweise den bayerischen Vermessungsämtern[5] oder der Stadt Schwäbisch Hall, sowie zahlreichen laufenden Migrationsprojekten wie dem der Stadt München[6], des Bundesamtes für Sicherheit in der Informationtechnik (BSI)[7] und des Bundeskartellamts[8] scheint es kaum strittig, daß eine Linux-Migration sowohl von Server- als auch Clientseite in der Verwaltung technisch möglich ist.

Auch die Resultate des genannten Migrationsprojekts an der Fakultät für Chemie der Universität Tübingen, bei dem bei Verwendung des in dieser Arbeit vorgestellten Verfahrens immerhin 50% der Fachapplikationen unmodifiziert unter Wine stabil betrieben werden konnten erscheinen hier sehr ermutigend. Es stellen sich dennoch eine Reihe von technischen und administrativen Fragen.

Klassifikation der Applikationen Die Fachanwendungen der Stadt lassen sich anhand der Hersteller und der Migrierbarkeit im Wesentlichen in die folgenden Kategorien einteilen:

1. Kooperativer Hersteller. Hier ist der Hersteller bereit, eine Linux-basierte oder zumindest plattformübergreifende Version anzubieten bzw. hat dergleichen bereits in Entwicklung.

2. „Böblinger Weg“. Bei migrierfähigen Applikationen, bei denen der Hersteller nicht bereit ist, eine Linux-Version herauszubringen und zu

[5] Hier erfolgte eine Migration von 3000 Bildschirmarbeitsplätzen in der Vermessungsverwaltung. Quelle `http://www.golem.de/0306/25910.html`, zuletzt besucht am 6.12.2004.

[6] Migrationsprojekt „Limux“ der Stadt München: `http://www.muenchen.de/Rathaus/dir/limux/projekt/89257/index.html`, zuletzt besucht am 7.12.2004.

[7] Projekt BSIMig: `http://www.kbst.bund.de/OSS-Kompetenzzentrum/Open-Source-Projekte-,276.304817/Migration-der-IT-Infrastruktur.htm,,,128,,,,,0,10,kbs_zur_einzelsicht.htm`, zuletzt besucht am 6.12.2004.

[8] Migration Bundeskartellamt: `http://www.kbst.bund.de/OSS-Kompetenzzentrum/Open-Source-Projekte-,276.304829/Migration-Bundeskartellamt.htm,,,128,,,,,0,10,kbs_zur_einzelsicht.htm`, zuletzt besucht am 6.12.2004.

pflegen kann die Stadt Böblingen eigenständig die Windows-Version unmodifiziert unter Wine betreiben. Es stellen sich jedoch sofort eine Reihe weiterer Fragen, die im Folgenden diskutiert werden sollen.

3. Unkooperativer Hersteller. In diesem Fall ist die Applikation ohne Modifikation nicht migrierfähig und der Hersteller nicht bereit, diese Modifikationen vorzunehmen und eine unter Linux lauffähige Version zu erstellen.

4. Unbekannter Hersteller. Es ist zu erwarten, daß unter den kommunalen Fachanwendungen auch Legacy-Applikationen zu finden sind, bei denen der Hersteller insolvent ist bzw. der Quelltext aus anderen Gründen nicht mehr verfügbar.

Untersuchung und Konsolidierung Es wäre angeraten, zunächst die gesamten Fachapplikationen in Böblingen zu untersuchen und festzustellen, ob ein Äquivalent als Open Source Software zur Verfügung stehen würde. Dieses wäre beispielsweise bei der eingesetzten Office-Suite von Microsoft und bei einigen Hilfsprogrammen wie z.B. WinZip der Fall.

Zudem wäre zu untersuchen, in welchem Umfang Programme durch andere Lösungen ersetzt werden könnten bzw. aus technischer Sicht vereinheitlicht werden könnten. Eine durchaus erhebliche Anzahl an kommunalen Fachapplikationen sind aus technischer Sicht wenig anspruchsvoll und könnten funktional identisch beispielsweise unter SAP betrieben oder mit vertretbarem Aufwand nachprogrammiert werden. Es erscheint zudem angesichts der Anzahl der eingesetzten Fachanwendungen durchaus wahrscheinlich, daß ein gewisses Potential zur Verringerung dieser Anzahl existiert.

Für die verbleibenden Anwendungen sollte nach einer Klassifikation gemäß oben genanntem Schema insbesondere für die ohne Herstellerunterstützung migrierfähigen Anwendungen („Böblinger Weg“) geklärt werden, inwiefern bestehende Wartungs-, Upgrade- und Supportverträge durch einen Betrieb der Anwendung unter Linux möglicherweise hinfällig werden. Für eine Übernahme von unveränderten Applikationen in den Produktiv werden formalisierte Testszenarien benötigt, die auch bei Upgrades der Software eingesetzt werden können. Es ist wünschenswert, diese Tests weitestgehend automatisiert durchführen zu können. Der Betrieb unter Wine ohne Unterstützung durch den Hersteller sollte aber höchstens als Zwischenlösung betrachtet werden.

Zusätzliche Maßnahmen Es verblieben nach Untersuchung und Konsolidierung der Fachapplikations-Landschaft mit Sicherheit einige Programme, bei denen keine Migrierfähigkeit ohne Herstellerunterstützung gegeben ist. Um dieses Problem mittelfristig anzugehen, wären es sinnvoll, sich mit anderen laufenden oder abgeschlossenen Migrationprojekten wie denen der

Städte München und Schwäbisch Hall zu koordinieren, da sich die zu meisternden Aufgaben vielfach stark überschneiden werden.

Es wäre ebenfalls sinnvoll, wenn sich migrationswillige bzw. migrierene Gemeinden zusammenschließen, um gemeinsam eine bessere Verhandlungsposition gegenüber Herstellern kommunaler Software zu erreichen. Da sich die eingesetzten Fachapplikationen zumindest landesweit nicht allzu stark unterscheiden dürften, wäre es zudem möglich, gemeinsam mehr Applikationen migrierbar zu machen, in dem man gezielt besonders stark genutzte APIs in Wine nachprogrammieren läßt. Eine solche gezielte Nachprogrammierung ist auf Basis der Analyseergebnisse des in dieser Arbeit vorgestellten Systems problemlos möglich.

8.3 Ausblick

Unter Berücksichtigung der in diesem Kapitel diskutierten Aspekte erscheint über die Zielsetzung dieser Arbeit hinausgehend zukünftig eine Reihe an Fragen interessant.

Tests Trotz der Informationen, die durch Analyseergebnisse mittels des Verfahrens zur Verfügung stehen, ist es nicht angeraten, auf eine Testphase durch den Endbenutzer zu verzichten. Auch ist aus einer Liste der unvollständig implementierten APIs nicht immer im Vorhinein ableitbar, wie genau die Probleme im Betrieb der Applikation aussehen werden. Hierfür existieren – ohne Anspruch auf Vollständigkeit – mehrere Gründe:

- Fehler in der Software. Die Windows-Software kann auf eine Weise fehlerhaft sein, die unter Wine häufiger zu Problemen führt. Denkbar wäre z.B. ein einziger API-Aufruf, der unter Windows selten und unter Wine oft fehlschlägt, was zu einem Programmabsturz führt. Auch möglich sind subtile Fehler in nebenläufiger Programmausführung, die unter Wine z.B. wegen leicht anderem Timing Deadlocks o.Ä. produzieren[9].

- Fehler in Wine. APIs, die keine der Indikatoren für unvollständige oder fehlerhafte Implementierung enthalten, werden von der Analyse nicht entdeckt.

- Hacks. Aufgrund der großen Funktionsvielfalt der Win32-API gibt es oft mehrere Wege, die gleiche Aufgabe zu programmieren. Manche Software ermittelt z.B. Systeminformationen somit über Umwege, die

[9] Probleme der genannten Art treten in der Realität leider häufiger auf. Der Autor gestattet sich an dieser Stelle die Anmerkung, daß derartige Symptome als Hinweis auf ein gewisses Optimierungspotential des Herstellers bei der Qualitätssicherung gewertet werden könnten.

schlecht dokumentiert sind und deswegen in Wine nicht oder unvollständig nachgebildet sind. Beispielsweise versuchen einige Spiele über verschiedenste Arten herauszufinden, ob sie unter einem Debugger ausgeführt werden, um so die Entwicklung von Spiele-Cracks zu erschweren.

Es wäre daher für die Zukunft wünschenswert, typische Fallstricke und Probleme bei der Ausführung unter Wine mittels Test-Suiten automatisch zu überprüfen. Es exisitieren bereits die Anfänge eines Test-Frameworks *CXtest*[10], das unter der GPL-Lizenz zur Verfügung steht. CXtest ermöglicht die Erstellung automatisierter Tests verschiedener Teile einer Software, von ihrer Installation bis hin zu GUI-Tests.

Wartung In der vorliegenden Arbeit wurde von einer fiktiven Migration der Stadt Böblingen auf Linux ausgegangen. Es wäre konsequent, diesen Gedanken weiter zu treiben und für weitere Entwicklungen von einer fiktiven Kommune auszugehen, die eine größere Anzahl an Fachanwendungen unter Linux/Wine betreibt. Eine effiziente Lösung der hierbei häufig auftretenden technischen Probleme wäre ein lohnenswertes Unterfangen. Darunter fällt unter anderem ein Mechanismus zum Nutzertest, der eventuell weitmöglichst automatisiert werden sollte. Auch ein sinnvolles Vorgehen für Upgrades der Software müßte hier angegangen werden

Portierungen Es ist zu erwarten, daß sich winelib-basierte Portierungen mit Hilfe des entwickelten Verfahrens sehr effizient und mit vergleichsweise wenig Aufwand durchführen lassen. Sinnvoll wäre es, anhand von tatsächlichen Portierungsprojekten Vorgehensweisen zu entwickeln, um diese Projekte möglichst problemlos durchführen zu können. Die Lösung üblicher Probleme bei Portierungen könnte durch die Entwicklung von verschiedenen Tools weiter unterstützt werden, die z.B. ihre Informationen aus den Analyseergebnissen des Systems mittels seiner API abfragen und den Quelltext der Software dann entsprechend auf manuell umzuprogrammierende Stellen untersuchen. Auch eine Art Leitfaden beispielsweise für die Umstellung von verschiedenen Applikationstypen (GUI-basiert, Client/Server-basiert o.Ä.) wäre ein denkbarer Ansatz.

Verbesserungen des Verfahrens Interessant im Hinblick auf künftige Weiterentwicklungen und Anwendungen des Verfahrens selbst erscheinen zum jetzigen Zeitpunkt mehrere Aspekte:

- Automatisierung. Die Vorgänge im System könnten noch weiter automatisiert werden. Beispielsweise könnten die erstellte Testkonfigurati-

[10] Homepage von CXtest: `http://www.cxtest.org/`, zuletzt besucht am 1.12.2004.

on automatisch auf einen Datenträger übertragen und so ausgeliefert werden.

- Heuristik. Das System könnte soweit ausgebaut werden, daß es mittels Iteration und in der Datenbank gespeicherten Profilen von Windows-DLLs automatisch optimierte Konfigurationen ermittelt.

- Schnittmengenbildung. Es wäre interessant, mit Hilfe des Systems Schnittmengen zwischen zwei oder mehreren Applikationen hinsichtlich ihrer Problemstellen zu bilden. So könnten ganz besonders nützliche APIs entdeckt und gezielt nachimplementiert werden. Dieses Vorgehen könnte sich als gleichermaßen nützlich für Softwarehersteller wie für das Wine-Projekt erweisen.

- Überwachung. Die ermittelten API-Profile der Applikationen sind in der Datenbank des Systems gespeichert. Selbst wenn ein Testbetrieb als für nicht im gegebenen Kostenrahmen durchführbar angesehen wurde, wäre es mit dem entwicklenten System mittels eines Skriptes möglich, Applikationen in regelmäßigen Abständen automatisiert wieder mit dem aktuellen Wine-Implementierungsstand zu vergleichen. Verbessert sich so die Kompatibilität mit Wine im Laufe der Zeit, könnte so der Testbetrieb später wieder in Angriff genommen werden.

Da das Verfahren und seine Implementierung sich im kommerziellen Einsatz befinden, sind einige dieser Verbesserungen bereits in Planung oder in Angriff genommen worden.

Anhang A

Tabellendefinitionen

Tabelle A.1: Die Modultabelle speichert Informationen über PE-Softwaremodule in binärer Form und über Wine-Module, die Windows-Module nachimplementieren. Bei Wine-Modulen wird der Hashwert des spec-Files als Hashwert in der Datenbank gespeichert, während bei PE-Modulen der Hashwert der Moduldatei verwendet wird.

Feld	Typ	Null erlaubt	Key	Default
hash	varchar(72)		Primary	
md5	varchar(32)			
sha1	varchar(40)			
name	varchar(83)		Primary	
path	mediumtext	yes		
size	int(11)	yes		
image_base	int(11)	yes		
comment	text	yes		
ablob	mediumblob	yes		
type	enum		Primary	NOT_WINE
wine_version	enum		Primary	none
win_version	enum		Primary	none

Tabelle A.2: Die Config-Metatabelle nimmt Konfigurationen in textuell repräsentierter Form auf. Diese sind spezifisch für ein bestimmtes Projekt (pname) und sind über Projektname und Konfigurationsnamen (cname) eindeutig identifizierbar.

Feld	Typ	Null erlaubt	Key	Default
pname	varchar(255)		Primary	
cname	varchar(100)		Primary	
paths	text			
content	text			
comment	varchar(100)			

Tabelle A.3: In der Import-Tabelle des Systems werden analog zu den Import-Tabellen von PE-Modulen Informationen über importierte Funktionen von Modulen gespeichert.

Feld	Typ	Null erlaubt	Key	Default
hash	varchar(72)			
md5	varchar(32)		Primary	
sha1	varchar(40)		Primary	
name	varchar(80)		Primary	
imported_from	varchar(83)		Primary	
function	varchar(255)		Primary	
function_demangled	mediumtext	yes		
return_type	mediumtext	yes		
calling_convention	enum	yes		
parameter	mediumtext	yes		
ordinal	int(11)		Primary	0
by_ordinal	tinyint(1)	yes		0
entry	int(11)		Primary	0

Tabelle A.4: In der Export-Tabelle des Systems werden analog zu den Export-Tabellen von PE-Modulen Informationen über exportierte Funktionen von Modulen gespeichert.

Feld	Typ	Null erlaubt	Key	Default
hash	varchar(72)		Primary	
md5	varchar(32)			
sha1	varchar(40)			
name	varchar(80)		Primary	
function	varchar(117)			
function_demangled	varchar(250)		Primary	
return_type	mediumtype	yes		
calling_convention	enum	yes		
paramter	varchar(220)			
ordinal	int(11)		Primary	0
entry	int(11)			0
forwards_to	varchar(200)			null
semi_stub_count	int(11)	yes		0
stub_count	int(11)	yes		0
fixme_count	int(11)	yes		0
wine_version	enum		Primary	none

Tabelle A.5: Die Projekt-Tabelle dient dazu, Module zu gruppieren und Informationen (z.B. Pfadangaben) zu Modulen zu speichern, die spezifisch für ein Projekt sind. Eine solche Gruppe von Modulen wird vom Project-Objekt des Systems für die Analysen verwaltet.

Feld	Typ	Null erlaubt	Key	Default
win_version	enum			none
pname	varchar(255)		Primary	
hash	varchar(72)		Primary	
md5	varchar(32)			
sha1	varchar(40)			
path_md5	varchar(32)		Primary	
name	varchar(83)			
path	mediumtext			

Tabelle A.6: Die Projekt-Metatabelle nimmt Informationen auf, die mit einem Projekt assoziiert sind, jedoch nicht direkt mit den in Projekt gruppierten Modulen verbunden sind. Dazu zählen beispielsweise Installationsprotokolle, projekt-globale Pfadeinstellungen, oder Informationen über Version und Hersteller der analysierten Software.

Feld	Typ	Null erlaubt	Key	Default
pname	varchar(255)	no	Primary	
win_path	mediumtext	yes		
win32_path	mediumtext	yes		
graph	mediumblob	yes		
graph_object	mediumblob	yes		
created	varchar(20)	yes		
modified	varchar(20)	yes		
missing_modules_reference_count	int(11)	yes		0
missing_modules_count	int(11)	yes		0
missing_modules	mediumtext	yes		
incomplete_modules_reference_count	int(11)	yes		0
incomplete_modules_count	int(11)	yes		0
incomplete_modules	mediumtext	yes		
missing_functions_reference_count	int(11)	yes		0
missing_functions_count	int(11)	yes		0
missing_functions	mediumtext	yes		
works_with_wine_versions	mediumtext	yes		
works_with_wine	tinyint(1)	yes		0
analysis_object	longblob	yes		
changes	text	yes		
software_name	text	yes		
software_version	text	yes		
software_language	text	yes		
software_vendor	text	yes		
software_description	text	yes		
software_needs	text	yes		
software_components	text	yes		
software_complexity	text	yes		
software_license_info	text	yes		
protocol	text	yes		
graph_is_dirty	tinyint(1)	yes		0
graph_object_is_dirty	tinyint(1)	yes		0

Tabelle A.7: Die Wine-Metatabelle dient zum Verwalten und Speichern von für jeweils eine bestimmte Wine-Version vorinitialisierten Loader-Objekten, die aus der Datenbank direkt in den Speicher geladen werden können.

Feld	Typ	Null erlaubt	Key	Default
pname	varchar(100)		Primary	
loader	mediumblob	Yes		
is_wine	tinyint(1)			0

Anhang B

Abkürzungsverzeichnis

API ..... Application Programming Interface. Eine API ist die Schnittstelle, die ein Betriebssystem oder auch ein anderes Softwaresystem anderen Programmen zur Verfügung stellt. Im Gegensatz zu einem ABI (Application Binary Interface) definiert ein API nur die Verwendung der Schnittstellen, nicht aber deren Realisierung. Die meisten Betriebssysteme bieten sehr vielfältige APIs an. Neben den Zugriffen auf die Hardware wie Festplatte oder Grafikkarte wird dem Programmierer auch das Erstellen von Komponenten der graphischen Benutzeroberfläche erleichert.

FLOSS ..... Freie und Open Source Software. Während *Open Source* im Wesentlichen das Offenliegen von Quellcode bezeichnet, sind bei Freier Software eine Reihe von weiteren Bedingungen zu erfüllen, z.B. das Recht auf Modifikation des Quelltextes (siehe *GPL*). Im Englischen bedeutet das Wort „free“ sowohl „frei“ als auch „gratis“, weshalb der Buchstabe L (wie in „Liberty“) dem Akronym hinzugefügt wurde, um klar zu machen, daß es sich bei Freier Software um ein grundlegend anderes Konzept handelt, das mit dem Preis des Programmes nur peripher etwas zu tun hat. Weitere Informationen unter `http://de.wikipedia.org/wiki/FLOSS`.

GNU ..... GNU's not Unix. Das GNU-Projekt wurde 1983 von Richard Stallman mit dem Ziel gegründet, ein vollständig freies Betriebssystem, das GNU System, zu entwickeln.

Daraus entstand 1985 die Free Software Foundation, eine gemeinnützige Organisation, die einen logistischen, juristischen und finanziellen Rahmen für das GNU-Projekt schaffen sollte. In Linux-Systemen sind typischerweise alle üblichen Unix-Tools durch unter der GPL lizensierte Implementierungen des GNU-Projektes ersetzt. Weitere Informationen unter `http://de.wikipedia.org/wiki/GNU`.

GPL ..... GNU General Public License. Die GPL ist eine Lizenz für sog. Freie Software. Darunter versteht man Software, deren Lizenzbestimmungen dem Nutzer und den Programmierern bestimmte Freiheiten einräumen. Zu den wichtigsten dieser Freiheiten gehört die Offenlegung des Quelltextes und das Recht jedermanns, daran beliebige Modifikationen vornehmen zu können. Diese Veränderungen müssen ihrerseits wieder unter den gleichen Bedingungen verfügbar sein.

JDBC ..... Java Database Connectivity ist eine API der Java-Plattform, die eine einheitliche Schnittstelle zu Datenbanken verschiedener Hersteller bietet und speziell auf relationale Datenbanken ausgerichtet ist.

KDE ..... K Desktop Environment. KDE ist eine moderne grafische Arbeitsumgebung für Unix-Computer. Sie ist die voreingestellte graphische Oberfläche unter den meisten großen Linux-Distributionen. KDE ist vollständig Open Source, die überwiegende Mehrheit des Codes ist sogar Freie Software.

LGPL ..... GNU Lesser General Public License. Die LGPL ist eine etwas restriktivere Variante der GPL. Im Wesentlichen erlaubt sie es, gegen als Freie Software vorliegende Bibliotheken zu linken, ohne daß die dagegen gelikten Programme selbst unter den Bedingungen der GPL verfügbar sein müßten.

ODBC ..... ODBC steht für Open DataBase Connectivity und ist ein Datenbanktreiber, bietet also eine API die einem Programmierer erlaubt, seine Anwendung unabhängig vom verwendeten Datenbank-Server (der Datenbank) zu entwickeln.

SQL ..... Structured Query Language. Hierbei handelt es sich um eine Abfragesprache für relationale Datenbanken.

WINE ..... Das Programm Wine, ein rekursives Akronym für „Wine Is Not an Emulator“, ermöglicht es Programme, die für das Betriebssystem Windows geschrieben wurden, auf der grafischen Oberfläche von Unix zu starten. WINE ist Freie Software unter der LGPL-Lizenz. Wine kann ohne vorhandene Windows-Installation verwendet werden, da jedoch einige Bibliotheken noch unvollständig sind, bietet Wine die Möglichkeit an, DLLs und die Registry einer vorhandenen Windows-Version zu verwenden, um so die Kompatibilität zu Windows-Applikationen verbessern.

Literaturverzeichnis

[BMI 2003] Migrationsleitfaden - Leitfaden für die Migration der Basissoftwarekomponenten auf Server- und Arbeitsplatz-Systemen, Version 1.0, Juli 2003, Schriftenreihe der KBSt, Band 57

[PECOFF 1999] Microsoft Corp., Microsoft Portable Executable and Common Object File Format Specification, Februar 1999, verfügbar unter `http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx`, zuletzt besucht am 2.11.2004

[Inside PE 2002] Pietrek, M., An In-Depth Look into the Win32 Portable Executable File Format, 2002, MSDN Magazine, verfügbar unter `http://msdn.microsoft.com/msdnmag/issues/02/02/PE/default.aspx` und `http://msdn.microsoft.com/msdnmag/issues/02/03/PE2/default.aspx`, zuletzt besucht am 2.11.2004

[iX 2004] Wessner, M., Mit oder ohne Aufsatz - Microsoft Windows 2003 vs. Citrix Metaframe XPe, 2004, iX Magazin für professionelle Informationstechnik, 2/2004, S. 78-82

[Klaeren 2004] Klaeren, H., Skriptum Softwaretechnik (Entwurf), Tübingen, 2004, verfügbar unter `http://www-pu.informatik.uni-tuebingen.de/users/klaeren/sw.pdf`, zuletzt besucht am 7.12.2004

[Kryptologie 2003] Hauck, P., Kryptologie und Datensicherheit, Skriptum zur Vorlesung im Wintersemster 2002/2003, Tübingen, 2003

[Linux Magazin 2004] Bablok, B., Zahlen zeigen, 2004, LinuxMagazin, 6/2004, S. 118-122

[Handelsblatt 2002] Linux auf Arbeitsplätzen durchsetzen, Handelsblatt, 30. Juni 2002, verfügbar unter `http://www.handelsblatt.com/hbiwwwangebot/fn/relhbi/sfn/buildhbi/cn/GoArt!200104,201197,543060/SH/0/depot/0/`, zuletzt besucht am 7.12.2004

[IT Manager's Journal] Preimesberger, C., Four out of four experts agree: Linux lowers TCO, IT Manager's Journal, Juni 2004, abrufbar

unter http://management.itmanagersjournal.com/print.pl?sid=04/06/04/2114222, zuletzt besucht am 7.12.2004

[Gümbel 2004] Gümbel, D., Effiziente Migration von Fachanwendungen auf das Linux Betriebssystem, September 2004, Vortrag beim Städtetag Baden-Württemberg (AK IuK), auf beiligender CD abrufbar