BURST

A Bug Reporting System for OpenStep compliant systems

Analysis, Design and Implementation

Design and implementation 1999 by
Philippe C.D. Robert, robert@iam.unibe.ch

for

Prof. Dr. O. Nierstrasz, oscar@iam.unibe.ch
Software Composition Group (SCG)
Institute of Computer Science and Appl. Mathematics (IAM)
University of Berne, Switzerland

and

Uptime Object Factory Inc
Matthias Heubi, mh@uptime.ch
Technoparkstrasse 1
8000 Zürich

Contents

Chapter 1

Analyse

Ziel dieser Analyse ist es, die Anforderungen an ein Bug Reporting System zu betrachten und Wege aufzuzeigen, wie sich eine in 'vernünftigem Rahmen' annehmbare Lösung realisieren lässt.

Die hier gewonnenen Einsichten sollen ausserdem als Grundlage dazu dienen, ein Pflichtenheft für eine solche Softwarelösung zu erarbeiten, welche die darin geschilderten Erkenntnisse realisiert. Durch das Studium verschiedener, schon existierender Lösungen, wie etwa GNU gnats, Jitterbug, BugNeXT oder das Debian Bug Reporting System, wird versucht, eine möglichst allgemeingültige Sicht der Problemstellung zu erfassen und zu analysieren.

Da man die Kommunikation zwischen Entwickler und Kunde, und speziell das Bug Reporting als einen wichtigen Bestandteil des Software-Entwicklungs - und Wartungszykluses sehen muss, wird im ersten Abschnitt zuerst speziell auf dieses Thema eingegangen, bevor anschliessend das eigentliche Thema dieser Analyse im Vordergrund steht.

1.1 Softwareentwicklung im Allgemeinen

Wie Andreoli, Meunier und Pagani[1] einleuchtend darstellen, kann Software-Entwicklung, ähnlich dem Workflow Management, mit einer Black Box modelliert werden. Die Black Box kann nun wiederum mit dem sogenannten Input - Process - Output - Paradigma (IPO) beschrieben werden kann. Dieser Prozess ist dabei mehrschichtig, resp. hierarchisch zu verstehen. An oberster Stelle befindet sich ein einzelner Input, der einen Prozess, in unserem Fall die Software-Entwicklung, auslöst, welcher als Resultat oder Ouput das gewünschte Produkt liefert. Dieser so initiierte Prozess kann wiederum in

[1] Jean-Marc Andreoli, Jean-Luc Meunier, Daniele Pagani. Process Enactment and Coordination. In the Proc. of the European Workshop on Software Process Technology (EWSPT 1996) held in Nancy, 9 - 11 October 1996

verschiedene Subprozesse eingeteilt werden, die als kleinste Bestandteile aus sogenannten Aktivitäten bestehen.

Aktivitäten, Prozesse und Subprozesse sind somit in einem Netzwerk von Abhängigkeiten miteinander verbunden, welches den Informations - und Kontrollfluss des obersten Prozesses darstellt. Jede einzelne Aktivität befindet sich dabei in genau einem von drei möglichen Zuständen.

Inaktiv: Die Aktivität kann nicht ausgeführt werden, da die hierzu notwendigen Bedingungen nicht erfüllt sind.

Aktiv: Alle Bedingungen und Abhängigkeiten sind erfüllt, die Aktivität wird ausgeführt.

Vollendet: Die Aktivität wurde beendet und hat den gewünschten Output erzeugt. Die darauffolgenden Aktivitäten können somit gestartet, d.h. aktiv werden.

Man beachte hierbei, dass eine Aktivität ohne weiteres mehrere Folgeaktivitäten gleichzeitig auslösen kann. Wie man einfach daraus schliessen kann, wird der oberste, d.h. der 'eigentliche' Prozess am schnellsten durchgeführt, wenn die einzelnen Aktivitäten möglichst reibungslos in ihrer logischen Reihenfolge ausgeführt werden können. Dies bedingt das Vorhandensein einer guten Kommunikation innerhalb des Netzes, sodass inaktive Zustände selten auftreten.

1.2 Warum ein Bug Reporting System?

Testphasen gehören als fester Bestandteil zu jedem Zeitpunkt in den Prozess der Software-Entwicklung. Wenn man vom oben geschilderten Modell ausgeht, gilt es dabei, den Informationsfluss einerseits zwischen Tester(n) und Entwickler(n), und andererseits innerhalb eines Entwicklerteams möglichst effizient zu gestalten, damit inaktive Zustände vermieden werden können und das Verhältnis zwischen Informationsmenge und dem tatsächlich daraus resultierenden Informationsgewinn möglichst positiv ausfällt.

Da Software heutzutage beinahe immer im Team erstellt wird, ist es unerlässlich, gewisse Strukturen und Regeln bzgl. des Arbeitsprozesses aufzustellen, respektive einzuhalten, um sich nicht gegenseitig durch einen überdimensionierten Kontrollfluss-Overhead unnötig zu blockieren. Aus diesem Grund ist es sinnvoll, auch diesen Prozess zu standardisieren und ihm einen Rahmen zu geben, der allen beteiligten Mitarbeitern, seien es Tester, Maintainer oder Entwickler, ein einfaches aber wirkungsvolles System zur Verfügung stellt, möglichst effizient zusammen das gemeinsame Ziel anzusteuern, die optimale Software.

Ausserdem ist es mit einem Report System möglich, die verschiedenen Stadien der Software-Entwicklung jederzeit nachzuvollziehen, d.h. eine Art

Statistik des Arbeitsprozesses zu erstellen, welche wiederum Rückschlüsse auf die Effizienz des Entwicklungsprozesses zu ziehen erlaubt[2]. So erlangte Informationen können via Arbeitsprozessoptimierung zu einer allgemeinen Qualitätssteigerung beisteuern.

1.2.1 Testphasen

Wie schon erwähnt, finden Testphasen immerwährend im Gesamtverlauf der Entwicklung - und natürlich auch danach - statt.

Aus dieser Erkenntniss wird schnell klar, dass den verschiedenen Kommunikationskanälen eine spezielle und ausserordentlich wichtige Bedeutung zufällt. Gerade diese sind es, wo man sich Zeit und Ärger durch Optimierungen und Standardisierungen ersparen kann - denn es ist in der Praxis wohl nie so, dass der oben geschilderte Vorgang derart linear und reibungslos vonstatten geht.

Ein Bug Reporting System muss also einerseits dem Laien, respektive User die Möglichkeit geben, einen möglichst sinnvollen Bericht abliefern zu können, der andererseits dem Spezialisten die Chance gibt, die benötigten Informationen daraus extrahieren zu können. Ein Reporting System muss also eine gewisse Form aufweisen, die es allen beteiligten Personen erlaubt, ob Laie oder Spezialist, damit auch wirklich arbeiten können. Es bringt nichts, wenn man zwar ein solches System zur Verfügung hat, aber dauernd Nachfragen zu tätigen sind, um den Sinn einer Nachricht korrekt zu erfassen oder ergänzende Informationen zu erlangen! Dies stellt doch gewisse Anforderungen an die technischen Fähigkeiten und das Konzept der Benutzerführung eines solchen Systems!

1.2.2 Ist-Zustand

Der Informationsaustausch zwischen den einzelnen Partnern findet in den meisten Fällen 'unkoordiniert' statt, d.h. in mehr oder weniger zufälliger Art und Weise. Das bedeutet, es gibt keine speziell strukturierten Kanäle und Mechanismen für das Bug Reporting, die das oben geschilderte Netzwerk und dessen Informationsfluss optimieren. Folge davon sind Missverständnisse oder fehlende Informationen, die es bräuchte, um einen optimalen Arbeitsprozess zu erhalten. Dies ist v.a dann ein Problem, wenn Software Entwicklung verteilt stattfindet, z.B. mit mehreren unabhängigen Partnern, ev. sogar verteilt auf grössere Gebiete. Ausserdem sind die gesammelten Bug Reports, die jede Firma oder Organisation automatisch hat, oft nicht in einer Form gespeichert, die ein rasches und geziehltes Auffinden möglich macht. Der Verwaltungsaufwand, das sog. Bug Report Management ist also ev. nicht optimal gelöst.

[2] Jean-Luc Meunier. The Software Defect Reporting WorkFlow, XRCE

1.2.3 Soll-Zustand

Im Zuge einer Optimierung des Entwicklungsprozesses ist man daran interessiert, auch das übermitteln von Bug Reports oder Abweichungen von Verhaltensmustern gegenüber den Pflichtenheften schneller und effizienter zu gestalten. Man möchte in der Lage sein, zu jedem Zeitpunkt den gegenwärtigen Stand des Projektes erfassen zu können, etwaiges Fehlverhalten schnell und ohne Probleme direkt an die richtige Instanz weiterzuleiten und es sollte auch möglich sein, sich automatisch über bestimmte Systemzustände informieren zu lassen.

Ausser dem Informationsaustausch zwischen Kunde und Entwickler spielt auch derjenige innerhalb der Entwicklergruppe eine grosse Rolle: wer hat Bux X gefixt, respektive wer ist dafür zuständig, wo ist der Fix, in welcher Release ist der Fix eingebaut, etc. Dies sind alles Informationen, die meistens nur innerhalb der Entwicklergruppe von Bedeutung sind.

1.3 Mögliche Lösungsansätze

Im heutigen Technologiezeitalter existieren viele Möglichkeiten, die als Kommunikationskanäle benutzt werden können: direkte Kommunikation mit dem Datenbankserver (z.B. via EOF, ODBC, JDBC oder einer ähnlichen Technologie), die elektronische Post sowie das World Wide Web. Die Vorteile dieser Techniken sind, dass sie beinahe überall zur Verfügung stehen, und dass sie schnell und dennoch sehr einfach handzuhaben sind. Ein möglicher Lösungsansatz darf auf keinen Fall den Benutzer durch eine zu grosse Komplexität abschrecken!

Ferner stellt sich die Frage, wie und wo man die gesammelten Informationen ablegt. Die Daten können dabei auf zwei Arten verwaltet werden, zentral oder dezentral. Die erste Variante macht in diesem Fall mehr Sinn, da auf diese Weise eine Reduzierung der Kommunikationsflut realisiert werden kann und keine Verwirrung bzgl. dem Auffinden der gewünschten Daten entsteht. Kunden und Entwickler können so beide auf eine gemeinsame Datenbasis zugreifen, wobei nicht für beide Seiten die gleichen Informationen sichtbar sein müssen.

1.3.1 Datenbank vs. Dateisystem

Als Repository drängt sich meistens eine Datenbank mit standardisierter Schnittstelle, z.B. von Oracle oder Sybase, auf, da mit einer solcherart gewählten Lösung mit verschiedensten Technologien auf die Datensätze zugegriffen werden kann (z.B. via Web Frontend, Desktop Applikationen etc.), und sie bzgl. Geschwindigkeit und Komplexität gut skalierbar ist. Es darf nicht ausser acht gelassen werden, dass vom Grundprinzip her immer nach dem gleichen Schema vorgegangen werden kann, aber dass die Anforderun-

gen an ein zentrales Daten Repository für die Entwicklung eines grossen Projektes andere Dimensionen annimmt, als diejenigen einer Spezialsoftware mit einem sehr eingeschränkten Kunden - und Entwicklerkreis.

Wählt man den Weg, dass man die erfassten Reports direkt im Dateisystem anstatt in einer Datenbank speichert, gewinnt man dafür den Vorteil, dass Textoperationen meist einfacher, und ev. auch effizienter implementiert werden können (Stichwort Volltextsuche über alle Reports). Man kann so auch eventuell vorhandene, mÄchtige Tools wie *grep* einsetzen, um an gewünschte Daten zu gelangen.

1.3.2 Versenden von Bug Reports

Für das Versenden von Reports sollten Lösungen zur Verfügung stehen, die dem User eine Benutzerführung bieten, welche nicht im Voraus vom Gebrauch des Systems abschrecken. Zum Übertragen der Informationen findet am besten email Verwendung.

Obwohl, wie wir im nächsten Abschnitt sehen werden, eine automatische Verarbeitung der versendeten Reports nicht unbedingt das Ziel sein muss, sollten die Reports eine klare Gliederung aufweisen, um eine solche möglich zu lassen. Das kann am besten erfüllt werden, wenn ein vorgegebens GUI den Benutzer bei der Eingabe 'führt', z.B. mit einer HTML Form oder einer eigens dazu entwickelte Desktop Applikation. Beide Lsungen haben natürlich ihre Vor - und Nachteile.

Begeht man den Weg über das WWW, hat man den Vorteil, dass neue Schnittstellen (Benutzermasken) sehr schnell einem grossen Kunden - und Entwicklerkreis zur Verfügung gestellt werden können. Ausserdem bietet dieser Ansatz eine sehr gute Lösung, wenn dezentral Daten erfasst werden sollen - Kunden und Entwickler arbeiten bekanntlich selten am selben Ort. Andererseits ist das Internet nicht gerade für seine Geschwindigkeit bekannt, und auch auf der gestalterischen Seite des GUI (Look und v.a. Feel) ergeben sich zwangsläufig Einschränkungen, die man in Kauf nehmen muss. Dazu kommt noch, dass nicht jedermann gerne mit Browsern arbeiten und dass auch nicht überall genügend Web Zugriff wärend der Arbeit gewährleistet ist. Eine Desktoplösung hat auf der anderen Seite den Vorteil, dass ein optimales GUI erstellt werden kann, wobei auch die Geschwindigkeit (des Reporterfassens) nicht leidet. Leider ist es bei dieser Lösung schwieriger, neue Versionen allen beteiligten Parteien zukommen zu lassen. Im Gegenzug ist es einfacher, automatisch Informationen in einen Report einzufügen, die sich das System selber zusammenträgt. Das ist ein zentraler Punkt, da dem Benutzer soviel wie möglich an automatisierbarem Aufwand abgenommen werden soll! Dazu kommt, dass via Desktoplösung eine direkte Datenbankanbindung möglich ist, also kein Umweg über email gemacht werden muss.

Falls schon Ideen bzgl. einer Lösung des Problems vorhanden sind, sollen

diese logischerweise auch übermittelt werden können . Allgemein ist es sinnvoll, Beschreibungen in Erzählform zu verfassen, was die Struktur und Korrektheit des zeitlichen Ablaufs unterstützt. Ein weiteres Problem ist, dass ein und derselbe Bug mehrmals, durch verschiedene Reports beschrieben, in der Datenbank vorkommen kann. Wie kann man das nun am besten verhindern oder kenntzeichnen? Für die meisten FÄllte genügt es wohl, wenn eine hierzu ermächtigte Person von Zeit zu Zeit die Datenbank aktualisiert, d.h. Mehrfacheinträge, die das gleiche Fehlverhalten thematisieren, verbindet, resp. kombiniert. Parallel hierzu könnte ein Automatismus implementiert werden, der aufgrund von diversen Inhaltspunkten von Reports dem Administrator des Systems Vorschläge unterbreitet, welche Reports 'verdächtig' erscheinen. Natürlich können auch Benutzer des Systems diesen Part übernehmen. Erfolgen die Eintragungen der Reports in das System manuell, löst sich das Problem schon beinahe von alleine, v.a. bei kleineren Projekten, da die ausführende Person den Inhalt des Daten Repository wohl kennt.

1.4 Verarbeitung der Bug Reports

Um das Arbeiten mit den Reports zu erleichtern, gilt es diverse, zentrale Punkte zu beachten!

1.4.1 Form eines Bug Reports

Wenn die Reports automatisch vom System geparsed werden sollen, müssen die in ihnen enthaltenen Informationen eindeutig einer gewissen Kategorie zuortbar sein. Das wird am besten mit dem sog. *Key - Value Prinzip* realisiert, bei jedem Informationsblock wird also ein eindeutiges Schlüsselwort zur Identifizierung mitgesendet, was etwa wie folgt aussehen könnte:

Name: Mr. J. Oke

Report: Helloworld.App says 'goodbye world' instead of 'hello world'

Muss der gesendete Report nicht in vom Menschen leserlicher Form sein, kann dieses Prinzip z.B. auch mit sog. *Dictionaries* realisiert werden. Weiter ist es auch möglich, den Mailheader als Informationsträger zu verwenden. Das kann von Nutzen sein, wenn es darum geht, die ankommenden Emails korrekt zu klassifizieren und daraufhin gewisse Aktionen auszulösen, z.B. das automatische Versenden von Empfangsbestätigungen etc.

Um das Bug Reporting System nicht unnötig komplex werden zu lassen, ist es ausserdem sinnvoll, folgende Einschränkung zu akzeptieren: es soll pro email nur genau ein Bug Report versendet werden können, denn auch hier gilt das Prinzip **KISS** (Keep It Simple and Stupid).

1.4.2 Archivieren und Publizieren von Reports

Wie schon erwähnt, sollen die Reports alle zentral, ev. in einer Datenbank gespeichert werden. Dabei werden die einkommenden Emails entweder alle automatisch geparsed oder die Informationen des Reports werden von Hand in das System eingegeben. Ein Report wird so unter einer eindeutigen ID abgelegt. Nachträglich zu einem Eintrag hinzugefügte Kommentare und/oder Erweiterungen müssen unter derselben ID abgelegt werden. Das bedeutet, der Schlüssel zu einem Bug Report ist seine ID. Werden grosse Attachements dem Report hinzugefügt (z.B. *core files*), so macht es eventuell keinen Sinn, diese in der Datenbank zu speichern, sondern es sollte möglich sein, solche Dateien im (lokalen) Dateisystem abzulegen, und zwar an einem Ort, wo sie jederzeit wieder gefunden und einem bestimmten Report zugewiesen werden können.

Was muss also unbedingt in einem solchen Report stehen, und was wäre einfach nur nützlich daraus herauslesen zu können? Diese Frage allgemein zu beantworten ist nicht einfach, aber man kann sich dazu immer folgendes vor Augen halten: es gibt keine überdetaillierte Fehlerbeschreibungen! Kleinste Details können die Ursache von grössten Problemen sein. Was unbedingt immer enthalten sein muss, ist eine Kontaktperson (oft ist dies wohl diejenige, die den Bug entdeckt hat), um bei eventuell nötigen Nachfragen eine Anlaufsstelle zu haben. Weiter darin enthalten sein muss natürlich das vermeintliche Fehlverhalten, eine Beschreibung wie es dazu kam, und in welcher Umgebung dies geschah (Software Versionsnummern etc.).

Geht es darum, die in der Datenbank gespeicherten Informationen abzurufen, muss man eine Unterscheidung machen zwischen der Maintainer - und Entwicklerseite, und der Kundenseite. Die Sichtweisen dieser zwei Parteien sind grundsätzlich nicht gleich. Ist der Entwickler v.a. an technischen Informationen bzgl. eines Software Fehlverhaltens interessiert, will der Kunde eher wissen, ob ein Bug noch existiert, wie er ev. zu beheben oder zu umgehen ist. Auch macht es überhaupt keinen Sinn, interne (technische) Informationen der Allgemeinheit zukommen zu lassen! Es muss also eventuell sogar verschiedene Kategorien von Systembenutzern mit ihren eigenen Zugriffsmöglichkeiten geben.

Andererseits macht es sicher Sinn, Übersichtsinformationen allgemeiner Art über den Stand eines Projektes publik zu machen. Für diesen Zweck scheint das Internet (WWW) wie geschaffen (z.B. wenn jemand wissen möchte, ob Bug Nr. X von System Y noch existiert oder nicht), Informationstransparenz schafft Vertrauen und vermittelt Seriosität - natürlich nur, wenn die Bugliste nicht allzu gross ist...

1.5 Abschliessende Bemerkungen

Wie immer, wenn es um Kommunikation zwischen verschiedenen Partner geht, ist es wichtig, ein gesundes Mass an Balance zwischen technischen Lösungen und persönlichen Kontakten zu finden. Auch in diesem Fall kann ein solches Bug Reporting System nicht die Lösung für alle Probleme darstellen, aber es kann der Ansatz, resp. das Mittel für eine sinnvolle Arbeitsprozessgestaltung in Bezug auf das Testing und Bug Fixing sein. Es wird auch nicht immer eine vollautomatische Lösung die optimale sein! Wichtig ist, dass jede beteiligte Person das System auch wirklich benutzt, nur so werden seine Möglichkeiten voll ausgeschöpft werden können.

Resourcen im WWW

- Die Debian Linux Distribution: www.debian.org
- Die Free Software Foundation (GNU): www.fsf.org
- Das Jitterbug System: http://samba.anu.edu.au/jitterbug
- Problem Management Tools Summary: www.iac.honeywell.com/Pub/Tech/CM/PMTools.html
- Project Management Bug Tracking for Linux: http://linas.org/linux/pm.html

Chapter 2

Pflichtenheft

Dieses Kapitel beschreibt die Anforderungen an das Bug Reporting System BURST, welches im Rahmen eines Informatik Projektes (IP) an der Universität Bern vom Autor erstellt wird. Dieses Pflichtenheft stützt sich hauptsächlich auf die in Kapitel 1 erarbeitete Analyse und ausführliche Gespräche mit den Entwicklern von der Firma Uptime Inc[1].

2.1 Grundkonzept

BURST stellt eine Lösung zur halbautomatischen Verarbeitung von elektronisch übermittelten Bug Reports dar. Das System besteht aus einer Desktop Applikation, genannt *Manager.app*, einer SQL Datenbank zur Speicherung und Verwaltung der Reports, einer Mail Applikation *Reporter.app*, sowie eventuell zusätzlich einem Web Frontend. Ausserdem soll die Möglichkeit offen gelassen werden, zu einem späteren Zeitpunkt ein *deamon* zu implementieren, welcher die via Email übermittelten Reports direkt in die Datenbank schreibt.

Die Applikationen Manager.app und Reporter.app sollen komplett mit OpenStep implementiert werden, d.h. das System sollte auf allen z.Z. vorhandenen - und auch zukünftig vorhandenen - OpenStep Systemen eingesetzt werden knnen: Mac OS X, Mac OS X Server, WebObjects auf Windows und später ev. auch einmal GNUstep. Das System soll in englischer Sprache implementiert werden.

2.1.1 Manager Applikation

Die Desktop Application Manager.app dient einerseits dazu, neue Reports, welche dem System via Email zugekommen sind, manuell zu erfassen, und diese an das Repository zu übermitteln, sowie andererseits bestehende Reports aus der Datenbank zu lesen und weiterzuverarbeiten (d.h. Änderungen

[1] http://www.uptime.ch

von sog. Status Informationen vorzunehmen, Ergänzungen anzubringen, etc.). Die Verbindung zum Datenbankserver wird mit dem Enterprise Object Framework (EOF) realisiert.

Beim Erfassen von neuen Reports wird (automatisch) ein sogenannter *Maintainer* des Reports festgelegt, welcher die Zuständigkeit über den im Report beschriebenen Bug innehat. Dies kann diejenige Person sein, die den Report in das System einspeist, oder aber es soll auch die Möglichkeit bestehen, dass die Verantwortlichkeit delegiert werden kann. Eine nachträgliche Veränderung der Verantworlichkeit muss möglich sein. Auch wird dem Report eine Priorität und ev. eine optionale Deadline zugeordnet, welche die Abarbeitungsreihenfolge mitbestimmt.

2.1.2 Reporter Applikation

Die sog. Reporter.app soll dem Kunden ein Werkzeug geben, Reports in einem strukturierten Rahmen verfassen und versenden zu können. Die geführte Benutzerführung erleichtert dem Maintainer des Bug Reporting Systems u.a. auch die Übertragung von Reports in das zentrale Repository. Als zusätzliches Feature für Mach-basierte Systeme ermöglicht der Reporter eine automatische Einbindung von systemspezifischen Informationen in den Report, wie z.B. Frameworkversionen, Hardwareinformationen (verwendete Driver, RAM, CPU, ...), Betriebssystemversion, etc. Diese Option soll abgeschaltet werden können, wenn dies erwünscht, resp. sinnvoll scheint.

2.1.3 WWW und Helfer Tools

Zur Unterstützung des Systems wird es in einem zweiten Schritt einige Helferapplikationen geben, die gewisse Services übernehmen, dazu gehört das automatische Parsen von einkommenden Reports und Einfügen in die Datenbank. Desweiteren soll das System so designed sein, dass ein WWW Frontend zu einem späteren Zeitpunkt hinzugefügt werden kann!

2.1.4 Erfassung von Reports

Für jeden neu zu erstellenden Report muss eine eigene Eingabemaske zur Verfügung stehen, welche dem User schon zu Beginn sinnvolle Defaulteinstellungen bereitstellt. Auch darf es nur möglich sein, einen einzigen Report pro Eingabemaske zu erstellen, resp. genau 1 Fehlverhalten pro Report zu melden! Aus dem Report sollen folgende Informationen ersichtlich sein:

- Submitter Name
- Submitter Email
- Submitter Organisation

- Kategorie, resp. betroffenes Produkt. Es werden nur vorgegebene Möglichkeiten angeboten, welche eine gewisse logische Gliederung ermöglichen (z.B. Software Bug, Dokumentations Inkonsistenzen, Änderungswünsche etc.)
- Ausführliche Beschreibung des Fehlverhaltens, dazu gehört auch, wie man ein Fehlverhalten repliziert, was genau dazu geführt hat, eventuell vorhandene Fix-Vorschläge, angehängte Dateien (ev. erzeugt durch Compiler, Linker etc.) usw.
- Ausmass des Fehlverhaltens (unknown, App crash, System crash, etc.)
- Interner Report (Yes oder No, falls Yes, dann werden die Informationen des Reports nicht öffentlich, z.B. via Web Frontend, zugänglich gemacht)
- Allgemeine Informationen (welche Betriebssystem-Version, welche Software Version, CPU, Memory, verwendete Treiber etc.)

Viele dieser hier genannten, zu erfassenden Informationen sollten aus den Grundeinstellungen abrufbar sein und automatisch vom Programm korrekt eingefügt werden. Bei Punkten wo eine Auswahl zur Verfügung steht, muss unbedingt ein sinnvolles Defaultverhalten definiert sein. Eine eventuelle Erweiterung könnte darin bestehen, dass es zu einem späteren Zeitpunkt möglich sein wird, gewisse Informationen an einen Report anzuhängen, welche nicht im Repository gespeichert werden (z.B. grosse core files oder screenshots), sondern direkt an eine zu bestimmende Personen weitergeleitet werden, resp. an einer hierzu auserkorenen Stelle im Dateisystem gespeichert werden.

2.2 Bug Reports und Change Requests

Dem Entwickler zugesandte Reports sollen auf eine sinnvolle Art und Weise in das zentrale Repository eingegeben werden können, d.h. dies muss schnell und ohne grosse Komplikationen zu bewerkstelligen sein. Das Anbringen von Ergänzungen an schon existierende Reports, auch zu einem späteren Zeitpunkt, muss intuitiv handhabbar sein, das bedingt, dass das Auffinden und Darstellen von bestehenden Reports eine zentrale Rolle einnimmt.

2.2.1 Verarbeitung der Reports

Um die Reports weiter zu bearbeiten, resp. abzurufen, ist es von grosser Bedeutung, dass diese schnell und geziehlt gefunden werden können, sowie auf eine übersichtliche Art (z.B. eine frei wählbare Reihenfolge der Reports unter Verwendung von drag and drop) auf dem Schirm und/oder

auf Papier dargestellt werden. Die einfachste Lösung ist diejenige, wo anhand der eindeutigen ID auf einen Report zugegriffen wird. Es soll aber auch möglich sein, eine Suche nach bestimmten Keywords durchzuführen. Zusätzlich ist es wünschenswert, die Möglichkeit des Filterns zur Verfügung zu haben, welche es erlaubt, nur einen gewissen Teil des bestehenden Datensatzes anzeigen zu können (z.B. alle Reports über ein spezielles Produkt), dies erlaubt eine logische Gruppenbildung von Reports. So gefundene Reports können vom Anwender von BURST erweitert, resp. ergänzt werden, d.h. es muss möglich sein, schon bestehende Informationen abzuändern, Ergänzungen anzubringen oder verschiedene Reports zu einem einzigen zu 'verschmelzen'. Grundsätzlich ist es wichtig, dass bei allen solchen Operationen keine Daten verloren gehen. Neue Daten werden an den bestehenden Datensatz angehängt, es werden keine Daten gelöscht! Es muss zu jedem Zeitpunkt ersichtlich bleiben, wer wann was mit einem Report unternommen hat, z.B. etwa eine Statusänderung. Mit Hilfe von Referenzen soll es auch möglich sein, Reports untereinander zu gewissen Gruppen logisch zu verknüpfen. So ist es etwa möglich, Reports einer Testliste zuzuordnen, welche immer dann neu gesichtet werden muss, wenn eine bestimmte Softwarekomponente einer Drittfirma eine Releaseänderung erfährt. Es besteht auch die Möglichkeit, Termine und Abarbeitungsprioritäten für bestimmte Events (z.B. das Erstellen eines Bugfixes) für einzelne Bugs und ev. auch für ganze Referenzlisten festzulegen.

Mit der Manager Applikation muss es auch möglich sein, gewisse Werte, wie z.B. den Status eines Reports modifizieren zu können. So ist ein neuer Report automatisch in einem Anfangszustand *incoming*, der nach dem erstmaligen Sichten in *opened* oder *rejected* geändert wird.

Die Reports müssen selbstverständlich auch ausdruckbar sein, wobei entweder ein bestimmter Report oder eine Übersichtsliste aller Reports zu Papier gebracht werden kann.

2.3 Spätere Erweiterungen

Mit dem Manager soll es zu einem späteren Zeitpunkt auch möglich werden, bestimmte Reports verfolgen, resp. tracken zu können. Dies ist interessant, etwa wenn Produkte von Drittfirmen involviert sind (Frameworks, Libraries etc.), welche in bestimmten Versionen Bugfixs oder Workarounds erzwingen und in anderen (neueren oder älteren) nicht.

Da alle Transaktionen im System gespeichert werden, ist es möglich, sich gewisse statistische Informationen generieren zu lassen, z.B.

- die Anzahl der z.Z. gespeicherten Reports (global, bzgl. eines bestimmten Projektes etc.)
- die Anzahl der Reports, die sich in einem gewissen Status befinden

(z.B. noch nicht gefixt sind), einem bestimmten Owner zugeordent sind etc.

- die durchschnittliche Dauer, die ein Report eines bestimmten Projektes in einem gewissen state innehat
- die mittlere Dauer bis ein Bug gefixt ist

Ganz allgemein spielt die Abhängigkeit von Bugs und deren Fixes, resp. Workarounds eine grosse Rolle, die es auf eine geschickte Art und Weise zu verwalten gilt (ev. mit sog. Referenzen, welche Reports untereinander koppeln). Es muss aus dem System ersichtlich sein, welche Bugs welche Einflüsse auf ein Produkt (und damit verbundene Workarounds und Fixes) haben.

2.3.1 WWW Frontend

Das WWW Frontend wir mit dem WebObjects Framework (WOF) realisiert. Es bietet in erster Linie nur die Möglichkeit, Reports online im Internet, resp. Intranet zu betrachten und abzurufen, ist also als reine Informationsquelle gedacht. Mit diesem Frontend zum Repository soll es dem Kunden ermöglicht werden, sich eventuell unter Zuhilfenahme bestimmter Suchmechanismen die ihn interessierenden Informationen jederzeit abzurufen. Interne, sowie technische Informationen werden so natürlich nicht zur Verfügung stehen.

2.4 Prioritäten

Das Schwergewicht liegt bei diesem Projekt eindeutig auf den Desktop Applikationen Manager.app und Reporter.app. Das Web Frontend nimmt bewusst ein weniger grosses Gewicht ein, da die meisten Interaktionen mit dem System über die Manager Applikation vollzogen wird. Diese Vorgehensweise ist auf die Bedürfnisse der Entwickler von Uptime Inc[2] hin zugeschnitten.

2.5 Testing

Das Testen wird eingeteilt in verschiedene, logisch gruppierte Abläufe, welche top-down ausgeführt werden. Designprobleme und Implementationsfehler können auf diese Weise frühzeitig entdeckt und gefixt werden, was einer schnelleren Reaktionszeit gleichkommt. Inaktive Zustände[3] werden so auf ein Minimum reduziert:

[2] http://www.uptime.ch
[3] vgl. Kapitel 1

- Testen der individuellen Komponenten
- Testen von Komponenten, welche logisch zusammenhängend sind
- Testen des kompletten Systems mit Testdaten
- Testen des kompletten Systems mit echten Daten

Beim Testen wird immer auf logisches und programmiertechnisches Fehlverhalten geachtet. Desweiteren ist dieser Testablauf immer als iterativer Prozess zu verstehen, welcher in allen Projektphasen anzuwenden ist, denn getestet wird nicht nur am Schluss! Auch darf nicht davon abgesehen werden, in früheren Phasen getestete Abläufe und Mechanismen erneut zu testen, jeder neue Implementationsstand ist als *Snapshot* des kompletten Systems zu betrachten und somit auch komplett zu testen!

2.5.1 Beispiel: Verwalten von Reports

Als Beispiel eines Testverlaufs seien hier einige Punkte betreffend dem Erfassen und Verarbeiten eines Reports aufgeführt, die es zu beachten gilt:

- Korrektes Einfüllen von Defaultvalues und bekannten, vordefinierten Einträgen
- Korrektes Einbinden von automatisch erlangten Systeminformationen
- Korrektes Versenden der Rich-Text-Format basierten Reports via email
- Korrektes UI inkl. Drag and Drop Unterstützung
- Korrekte und dem Umfang entsprechende Implementation aller Features gemäss dem Pflichtenheft
- Korrektes Erfassung von neuen Reports]item Korrektes Erfassen von Änderungen an bestehenden Reports
- Korrektes Darstellung aller Reports sowie einzelner Detailansichten
- Korrekter Such - und Filtermechanismus

Diest stellt natürlich nur eine kleine Auswahl der Punkte dar, die es zu beachten gilt. Für das Testing sollten darum Testszenarios erstellt werden, die es strikt zu befolgen gilt.

Chapter 3

Design

In diesem Kapitel sollen das Design und die Spezifikationen von BURST beschrieben werden. Das Ziel ist dabei, einen möglichst guten Überblick über das System zu geben, für Details wird auf den Sourcecode und die in den Header Dateien enthaltenen Beschreibungen verwiesen.
Das Design weisst zudem bestimmt diverse Schwachpunkte auf, ebenfalls die Implementation. Verbesserungsvorschläge sind darum sehr willkommen!

3.1 Manager

Die BURST Manager Applikation stellt ein intelligentes Frontend für die verwendete Datenbank und die darin enthaltenen Datensätze dar. Das gesammte Design der Applikation stützt sich somit auf das verwendete Datenmodell, welches via dem *Enterprise Objects Framework* in eine objektorientierte Hierarchie abgebildet wird.

3.1.1 Übersicht

Alle Operationen und Funktionalitäten sind als sogenannte *Enterpise Objects* implementiert, welche die Tabellen in *echte* Objekte überführen. Diese Objekte, wie etwa das später beschrieben, zentrale BTReport Objekt, sind in dem eigens dafuer erstellten Framework *DBKit.framework* enthalten. Der Prefix *DB* wird dabei von allen darin enthaltenen Klassen verwendet.
Desweiteren existiert ein spezielles Framework *BTTablePrinter.framework*, welches das Drucken von Daten, die in sogenannten NSTableView Objekten dargestellt werden, ermöglicht. Somit lassen sich auf einfache Art und Weise Report Listen erstellen.
Die eigentliche Logik von Manager.app ist in einem Subprojekt der Applikation enthalten, den BTExtensions, genau so wie die eigens entwickelten Logging - und Suchengines BTLogging resp BTFinder.
Ich habe im Folgenden die vom System bereitgestellten Frameworks nicht

aufgeführt, vollständigkeitshalber erwähnte ich aber, dass grundsätzliche alle 3 EO Frameworks sowie natürlich der FoundationKit, der AppKit und das Messaging Framework verwendet werden.

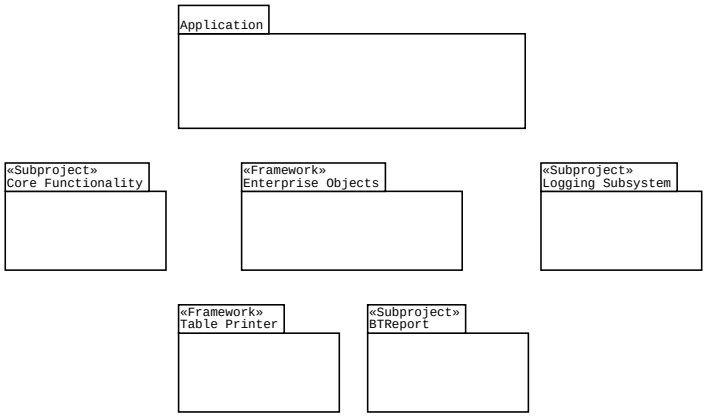

Figure 3.1: Komponentenübersicht Manager.app

3.1.2 Datenbankmodell

Das verwendete Modell gruppiert sich um die zentrale Entität *Report*. Von ihr aus erreicht man beinahe alle Informationen, die in der Datenbank gespeichert sind. Um eine genau Idee über das Zusammenpiel der Tabellen zu erhalten, verweise ich auf die Erklärung des DBKits. Folgende Entitäten sind vorhanden:

Report: Zentrale Tabelle, welche einen Bug Report repräsentiert. Folgende Attribute sind von Interesse:

- category (NSString)
- fixDate (NSCalendarDate, optional)
- fixDeadline (NSCalendarDate, optional)
- isPublic (NSNumber)
- reportNumber (NSNumber, optional)
- reportPriority (NSString)
- reportState (NSString)
- reportText (NSData)
- submissionDate (NSCalendarDate)
- title (NSString)

Desweiteren sind folgende Beziehungen implementiert:

- fixedVersion (optional)

- maintainer
- product
- reports
- submitter
- version

Ein Report hat also immer einen Submitter, einen Maintainer, ist einem Produkt zugeordnet, welches zwingend eine Version haben muss und ist charakterisiert durch Attribute wie 'category' oder 'reportState' etc. Die Beziehung 'reports' ist eine *many to many Relationship* zur Tabelle 'List'. Ein Report kann so in diversen Listen logisch gruppiert werden.

Product: Die Tabelle, welche die in der Datenbank verwalteten Produkte repräsentiert. Sie ist durch folgende Attribute charakterisiert:

- productName (NSString)
- productDescription (NSData)

Desweiteren sind folgende Beziehungen implementiert:

- reports (optional)
- versions

Ein Produkt hat also immer mindestens eine Version!

Version: Die Tabelle zur Verwaltung verschiedener Versionen von Produkten. Attribute:

- versionDescription (NSData)
- versionNumber (NSString)

Desweiteren ist folgende Beziehungen vorhanden:

- reports (optional)

Eine Version gehört also immer zu genau einem Produkt!

Submitter: Jeder Report hat genau einen Submitter, dieser stellt die Kontaktstelle für etwaige Fragen bzgl. des gemeldeten Reports dar. Er kennt die folgenden Attribute :

- company (NSString, optional)
- email (NSString, optional)
- fax (NSString, optional)
- fullName (NSString)

- phone (NSString, optional)
- responsibility (NSString, optional)

Weiter führt die *to many Relationship*

- reports (optional)

zu den von einem Submitter übermittelten Berichten.

Maintainer: Diese Tabelle stellt die angemeldeten Benutzer dar. Sie können sich im System einloggen und besitzen verschiedene Rechte bzgl. der möglichen Datenmanipulationen. Attribute:

- company (NSString, optional)
- email (NSString, optional)
- expire (NSNumber)
- fax (NSString, optional)
- fullName (NSString)
- login (NSString)
- password (NSString)
- phone (NSString, optional)
- responsibility (NSString, optional)
- rights (NSNumber)

Auch hier führt die *to many Relationship*

- reports (optional)

zu den von einem Maintainer verwalteten Berichten.

Customer: Diese Tabelle dient zur Verwaltung der Kunden von Produkten. Ein Kunde ist wie folgt mit Attributen charakterisiert:

- address (NSString, optional)
- comment (NSData, optional)
- country (NSString, optional)
- email (NSString, optional)
- fax (NSString, optional)
- name (NSString)
- phone (NSString, optional)
- state (NSString, optional)
- town (NSString, optional)

- www (NSString, optional)
- zip (NSString, optional)

Die *many to many Relationship*

- productsCustomers (optional)

verweist hier auf die von einem Kunden erstandenen Produkte.

List: Zur logischen Gruppierung von beliebigen Reports existieren Listen, sie haben folgende Attribute:

- listDescription (NSData, optional)
- name (NSString)

Die *many to many Relationship*

- reports (optional)

verweist hier auf die von den Listen referenzierten Reports.

LoggingTable: Zum Loggen der möglichen Transaktionen in BURST wird eine Tabelle verwendet, die folgende Informationen speichert:

- description (NSString)
- eventDate (NSCalendarDate)
- eventType (NSNumber)
- username (NSString)

Diese Tabelle steht für sich selbst.

ReportList: Hilfstabelle für die *many to many Relationship* zwischen 'Report' und 'List'.

ProductCustomer: Hilfstabelle für die *many to many Relationship* zwischen 'Product' und 'Customer'.

Das Datenbankmodell wird in genau einer EOModel Datei gespeichert und verwaltet, welche Bestandteil des sogenannten DBKits ist.

3.1.3 Subsysteme

BURST Manager wurde in diverse Subsysteme und Frameworks eingeteilt:

DBKit.framework: Enterprise Objects, welche mit dem EOModeller kreiert wurden und ev. vorhandene Categories zu diesen Klassen.

BTExtensions: Alle Klassen, die die eigentlichen Funktionalitäten von Mananger definieren und bereitstellen.

BTFinder: Klassen, die für Queries und Finding verwendet werden.

BTLoggin: Klassen, die für das Event Logging verwendet werden.

BTReport: Klassen, die für den direkten Import von versendeten Reports durch die Report.app benutzt werden.

BTTablePrinter.framework: Der sog. Table Printer zum Drucken von Objekten in NSTableViews via PostScript.

Darüber hinaus existieren noch diverse Klassen, welche eine übergeordnete Funktion haben und dementsprechend arrangiert sind.

3.1.4 Basisklassen

Folgende 3 Klassen sind im Toplevel Pfad von Manager gruppiert, da sie übergeordnete Kontrollerfunktionen wahrnehmen:

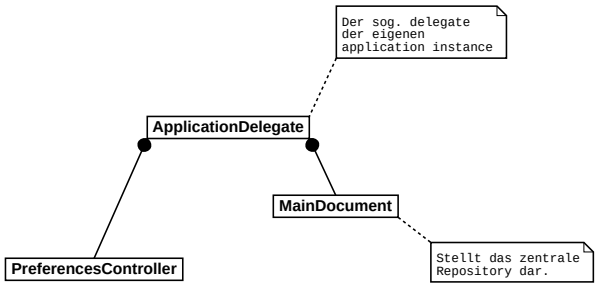

Figure 3.2: Toplevel Übersicht

ApplicationDelegate: Der Delegate der eigenen NSApplication, zuständig für das korrekte Initialisieren von Manager.app, dem Userlogin sowie dem korrekten Terminieren. Desweiteren implementiert sind Methoden für das Menuhandling, welche die entsprechenden Messages an die entsprechenden Objekte weiterleiten.

MainDocument: Das MainDocument stellt das eigentliche Datenrepository dar. Es ist zuständig für das Darstellen der Reports im Hauptfenster und instanziert diverse Objekte, wie etwa den Report Inspector oder Report Printer.

PreferencesController: Zuständig für die User Präferenzen, welche in der entsprechenden Tabelle des eingeloggten Benutzers gespeichert werden. Darunter fällt auch das Verwalten des Passwortes!

Weiter existieren zusätzlich noch 2 Header Dateien für globale Definitionen etc.:

BTDefines: Definiert die in der Applikation verwendeten Strings, sowie einige tags für die Datenbank (z.B. die verschiedenen Usergruppen).

local: Definiert die möglichen Exception Strings.

Zu einem späteren Zeitpunkt werden diese Dateien eventuell zu einer einzigen vereinigt.

3.1.5 Klassen im DBKit

Im DBKit sind alle Enterprise Object Klassen (sogenannte EOs) enthalten, welche vom EOModeller kreiert wurden, sowie mögliche, zugehörige Categories. Diese Klassen sind allesamt Subklassen von NSObject:

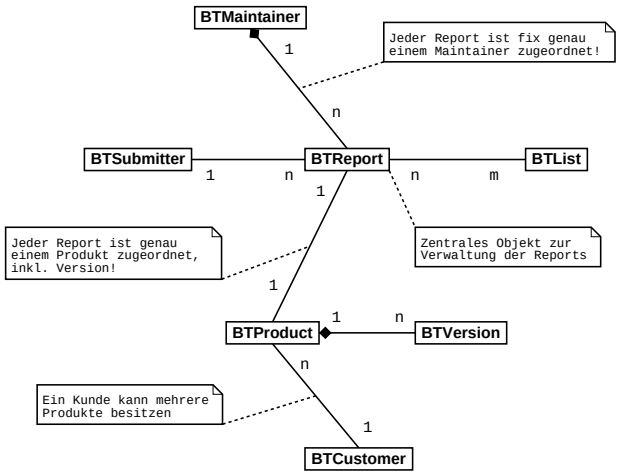

Figure 3.3: Übersicht DBKit.framework

DBCustomer: EO der Kundentabelle

DBList: EO der logischen Referenzlisten

DBLoggingTable: EO der Logging Tabelle

DBMaintainer: EO der Benutzertabelle

DBMaintainer+Extensions: Category zu DBMaintainer, z.Z. nicht verwendet.

DBProduct: EO der Produktetabelle

DBReport: EO der Reporttabelle

DBReport+Extension: Category zu DBReport, z.Z. nicht verwendet.

DBSubmitter: EO der Submittertabelle

DBVersion: EO zur Versionentabelle

Da in der aktuellen Version viele Features nicht implementiert oder aktiviert sind, ist nicht sehr viel erweiternder Code in diesen Klassen enthalten. Dies ändert sich in der nächsten Version, wenn z.B. komplexere Validierungen etc. eingeführt werden.

3.1.6 Klassen von BTExtensions

Im BTExtensions Subprojekt sind die meisten Klassen definiert, welche die eigentlichen Fähigkeiten der Applikation, bezogen auf das Reporthandling definieren.

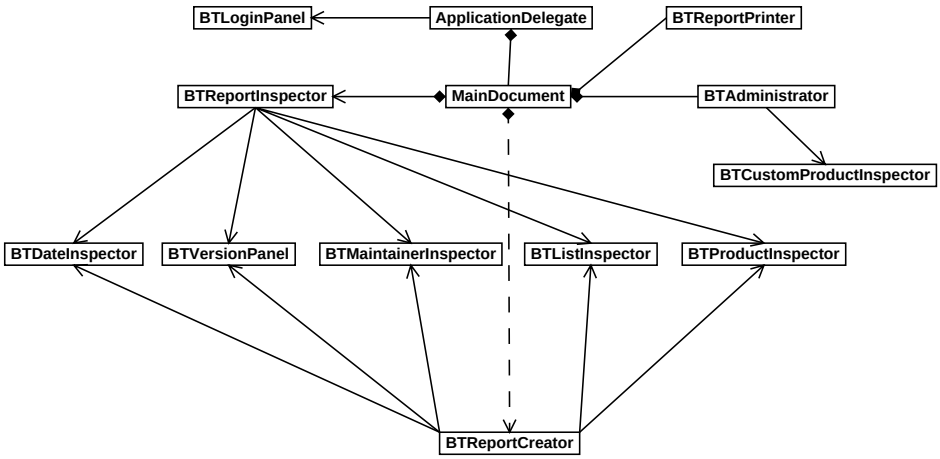

Figure 3.4: Übersicht BTExtensions

BTAbstractInspector: Eine abstrakte Klasse, definiert die Methoden für verschiedene Inspektoren, die als modale Panels implementiert sind. Die zu überschreibenden Methoden sind

- + (id)sharedInspector
- - (id)init

Zum Aufruf und Erfragen eines etwaigen Rückgabewertes sind die Methoden

- - (int)runModalWithObject:(id)anObject
- - (id)selectedObject

implementiert. Das übergebene Objekt wird momentan aber nicht verwendet, *nil* genügt also.

BTAdministrator: Eine zentrale Kontrollerklasse, welche die Administration der Maintainer, Submitter, Produkte und deren Versionen sowie der Listen koordiniert.

Diese Klasse wird später in mehrere, logisch sauberer gruppierte Klassen aufgeteilt werden!

BTCustomerProductInspector: Subklasse von BTAbstractInspector, verwaltet die Kunden und deren Produkte. Die Methoden

- - (int)runModalWithProducts:(NSArray *)someProducts
- - (NSArray *)selectedProducts

werden dabei für den Aufruf und das Erfragen der gewünschten Produkte verwendet, und nicht die im *parent* vorgesehene Standardmethoden!

BTListInspector: Subklasse von BTAbstractInspector, verwaltet die ReferenzListen, die zur logischen Gruppierung von Reports verwendet werden. Die Methoden

- - (int)runModalWithLists:(NSArray *)referencedLists
- - (NSArray *)selectedLists

werden dabei für den Aufruf und das Erfragen der gewünschten Produkte verwendet, und nicht die im *parent* vorgesehene Standardmethoden!

BTLoginPanel: Modales Loginpanel, wird vom ApplicationDelegate verwendet. Es kann aber auch für andere Appliaktionen verwendet werden, folgende Methoden sind dabei von Bedeutung:

- + (id)sharedPanel
- - (void)setResponse:(NSString *)string
- - (NSString *)response
- - (int)runModal
- - (NSDictionary *)loginDict

Im *loginDict* sind die Werte für den login name und das Passwort unter den Keys *@"LOGIN"* sowie *@"PASSWORD"* abgelegt.

BTMaintainerInspector: Subklasse von BTAbstractInspector, verwaltet die angemeldeten Benutzer des Systems. Die Methode

- - (int)runModalWithQualifier:(EOQualifier *)qualifier

wird für den Aufruf verwendet, und nicht die im *parent* vorgesehene Standardmethode! Der Qualifier wird dabei der verwendeten DisplayGroup übergeben.

BTProductInspector: Subklasse von BTAbstractInspector, verwaltet die Produkte, welche die Reports beschreiben. Die Methode

- - (id)versionOfSelectedObject

liefert zusätzlich die vom User selektierte Version.

BTReportCreator: Zentrale Kontrollerklasse zum Erstellen von neuen Reports mit der Manager Applikation. Die Methode

- - (void)createNewReport

öffnet und zentriert das hierzu benötigte Fenster. Ein so erstellter Report wird direkt in die DAtenbank gespeichert!

BTReportInspector: Der Inspektor, welcher die Details der einzelnen Reports darstellt. Er wird vom MainDocument instanziert und verwendet. Die Methode

- - (void)showInspector:(id)sender

öffnet das Inspektorfenster und stellt automatisch die korrekten Werte des in der verwendeten DisplayGroup selektierten Reports dar. Der Report Inspektor wird auch dazu verwendet, Werte des aktuellen Reports zu verändern, allenfalls vorhandene Abhängigkeiten werden dabei korrekt nachgeführt.

PRDateInspector: Ein Inspektor, welcher zur Auswahl eines beliebigen Datums mit Hilfe eines modalen Panels dient. Dieses wird mit den Methoden

- + (id)defaultDateInspector
- - (int)runModal
- - (NSCalendarDate *)selectedDate

aufgerufen und verwaltet. Der PRDateInspector verwendet dazu einen

Calendar: Kontroller, der die Grundfunktionalitäten eines Kalenders zur Verfügung stellt. Wird vom PRDateInspector verwendet, sollte also nicht direkt verwendet werden müssen!

Dies ist eine Portierung der frei verfügbaren NEXTSTEP Calendar.[hm] Klasse

BTVersionPanel: Panel zur Darstellung der Versionen eines bestimmten Produktes. Dieses ist das momentan selektierte Objekt der verwendeten DisplayGroup. Dieses wird mit den Methoden

- - (int)runModal
- - (NSCalendarDate *)selectedVersion

aufgerufen und verwaltet.

Dies ist kein shared Panel!

BTReportPrinter: Zuständig für das Drucken eines einzelnen, detaillierten Reports. Es wird dazu nur die Methode

- + (void)printReport:(DBReport *)aReport

benötigt. Es ist also keine spezielle Instanzierung nötig!

Dieses Objekt ist in der momentanen Version nicht funktionsfähig!

Da es galt, das angekündigte Releasedatum einzuhalten, sind viele der geplanten oder erwünschten Features der Manager Applikation in der momentanen Version nicht implementiert worden. Diese werden - falls positives Feedback dies verlangt - in einer nächsten Version nachgeliefert werden.

3.1.7 Klassen von BTFinder

Im BTFinder werden Klassen gruppiert, welche für das Auffinden von Reports in der der Datenbank dienen. Zur Zeit ist das nur der

BTSQLFinder: Klasse zum setzen eines SQL Qualifiers mithilfe eines shared Panels. Folgende Methoden werden dazu vernwedet:

- + (id)sharedFinder
- - (void)openQueryPanel

Wird der Query Button aktiviert, wird eine Delegate Message gesendet, die dem gesetzten Delegate den SQL String liefert:

- - (void)sqlFinder:(id)sender didSetSQLString:(NSString *)sqlString;

3.1.8 Klassen von BTLogging

Im BTLogging Subprojekt sind alle Klassen definiert, die für das Loggen von Transaktionen und Events verwendet werden:

BTLoggingController: Wird Logging verwendet, muss eine Instanz dieser Kontroller Klasse vorhanden sein. Diese ist zuständig für das öffnen und darstellen der vorhandenen Log Einträge, sowie dem Logging selber:

- + (id)sharedLogger
- - (void)showLogWindowWithRights:(NSNumber *)rights
- - (void)logMessage:(BTLogMessage *)aMsg
- - (void)deleteAllLogMessages:(id)sender

Die übergebenen *rights* definieren dabei, welche Möglichkeiten der aktuelle Benutzer hat (z.B. Löschen von Log Einträgen). Zur Zeit sind 3 verschiedene Usergruppen definiert - admin, user und guest - wobei nur Administratoren Log Einträge löschen können.

BTLoggingMailer: Klasse zum Versenden von Log Meldungen via Email. 2 Methoden sind dabei von Bedeutung:

- + (BTLoggingMailer *)sharedMailer
- - (void)mailMessage:(BTLogMessage *)msg withAdress:(NSString *)submitterAddress subject:(NSString *)subject

BTLogMessage: Klasse, welche die zu loggenden Meldungen darstellt. Eine solche Log Message hat immer eine Beschreibung und einen Typ:

- - (id)initWithMessage:(NSString *)msg type:(int)tp

Momentan verfügbare Typen sind in der Header Datei BTLogMessage.h definiert.

3.1.9 Klassen von BTReport

Das BTReport Subprojekt stellt die Klassen zur Verfügung, die zum automatischen Import von versendeten Reports benötigt werden.

BTReport: Definiert alle benötigten Attribute, die zum späteren Eintrag in die Datenbank benötigt werden. Wird ein Report mit der Reporter.app versendet, wird ei solcher Report generiert.

BTImportManager: Kontroller, welcher Objekte von Typ BTReport in die von der Applikation verwendeten Datenbank importiert. Dies geschieht mit den folgenden Methoden:

- + (BTImportManager *)sharedManager
- - (id)importBTReportWithMaintainer:(DBMaintainer *)aMaintainer

Diese Klasse ist z.Z. nicht vollständig implementiert, die Import Funktionalität ist nicht gegeben!

3.1.10 Klassen von BTTablePrinter

Der TablePrinter stellt die nach OpenStep portierte NEXTSTEP Palette zum Drucken von Tableviews dar. Das MainDocument instanziert einen TablePrinter, der für das Ausdrucken des Repository Inhaltes zuständig ist. Für eine genau Beschreibung der Arbeitsweise des TablePrinters sei auf die Kommentare in den Headerfiles hingewiesen. Der TablePrinter umfasst folgende Klassen:

- NSTableViewPrinter
- TablePrinter
- TablePrinterPanel

3.2 Reporter

BURST Reporter stellt eine Art BugNeXT clone dar, mit dem man Reports an ein BURST Repository senden kann. Die Idee ist, dass in einer nächsten Version ein automatisches Importing möglich sein wird.

3.2.1 Klassenstruktur

Die Reporter Applikation ist weit einfacher und kleiner als der Manager. Der Applikationskontroller *MBController* übernimmmt die Rolle des Delegate's der NSApplication und verwaltet die User Präferenzen.

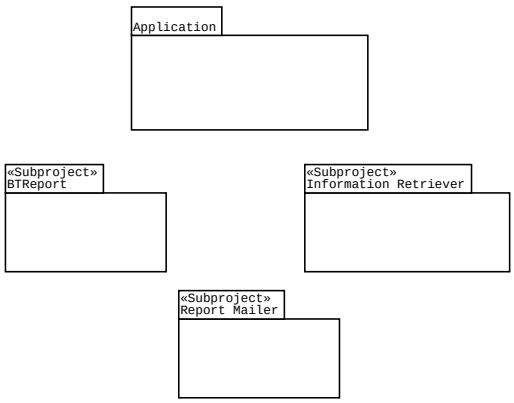

Figure 3.5: Übersicht Reporter.app

MBBrowser und *MBBrowserCellObject* sind der Delegate, resp. die Cell Objekte des Browser's, der im Hauptfenster zur Klassifizierung der Reports

verwendet wird. Sie werden vom MBController in dessen MBurst.nib Datei instanziert. Der Inhalt des Browsers wird durch die Datei *browser.description* bestimmt, welche folgendes Format aufweist:

Class: Produkt Name: Produktversion Specification: Beschreibung

Es können beliebig viele Zeilen in dieser Datei verwaltet werden, die Reihenfolge der Zeilen spielt keine Rolle.

Die zur Verfügung stehenden Zieladressen, an die ein Report gesendet werden kann, werden durch die Datei *destinations.list* definiert. Sie hat das Format einer Property List, z.B.

("someone@anywhere.net","soemwhereelse@nowhere.org")

Die Adressen erscheinen dann im dazu vorhergesehenen Popup Button im Composer Window.

Subprojekte

Reporter.app ist weiter in 3 Subprojekte eingeteilt:

BTReport: Dasselbe Subprojekt, das auch in der Manager Applikation verwendet wird, ausser das die Import Klasse weggelassen wurde.

InformationRetriever: Ein Subprojekt, implementiert nach dem sogenannten *Builder Pattern*, welches zur automatischen Informationsgewinnung über das verwendete System eingesetzt wird. Dabei muss eine Instanz der Klasse *InformationRetriever* instanziert werden. Dies geschieht mit der Methode

- - (id)initWithBuilder:(id< $ConcreteInformationBuilder$ >)bd

Es muss dabei ein konkreter *Builder* mitgegeben werden, welcher das *ConcreteInformationBuilder Protocoll* implementiert. Auf diese Weise können verschiedene Systeme einfach integriert werden, ohne das API ändern zu müssen! Die gewünschten Informationen über das Host System können dann mit den Methoden

- - (void)build
- - (NSString *)systemDependantInformation

gewonnen werden. Es können auch geziehlt nur bestimmte Informationen gewonnen werden, dazu sei auf die Header Datei *InformationRetriever.h* verwiesen.

Zur Zeit ist nur ein konkreter Builder für Mach 2.5 implementiert, der sogenannte *MachBuilder.[hm]*.

ReportMailer: Subprojekt, welches das Versenden von Reports via Email realisiert. Hierzu ist momentan eine Klasse vorhanden, *MBMailer.[hm]*, welche direkt via sendmail mails versendet. Dies geschieht mit folgenden Methoden:

- + (MBMailer *)defaultMailer
- - (BOOL)sendReport:(id)report from:(NSString*)sender to:(NSString*)recipient cc:(NSArray *)moreRecipients withSubject:(NSString *)subject andHeaders:(NSDictionary *)headers

In Zukunft wird statt einer direkten Ansprechung von sendmail das MailDelivery Framework von Apple verwendet, sodass auch SMTP möglich sein wird!

BURST

A Bug Reporting System for OpenStep compliant systems

Conclusion and final remarks

Written in June 2000 by
Philippe C.D. Robert, robert@iam.unibe.ch

for

Prof. Dr. O. Nierstrasz, oscar@iam.unibe.ch
Software Composition Group (SCG)
Institute of Computer Science and Appl. Mathematics (IAM)
University of Berne, Switzerland

Introduction

The bug reporting system BURST was designed and implemented as a *student project* for the Software Composition Group at the University of Berne in 1998/1999. The project was initiated by the company I worked at that time, Uptime Object Factory Inc and its CTO, M. Heubi.

Why a bug reporting system

To understand some of the problems I'll describe later on, one has to understand the situation in which BURST was realised. At the time the idea of the project came up for the first time, Uptime Inc was a small company, located in the *Technopark Zürich* as well as in the *Technopark Bern*, our core business was custom software development on NeXT's OPENSTEP, mainly for a few large companies.
On one side we were busy developing new custom applications, as well as doing ports from NEXTSTEP to OPENSTEP and maintaining other, existing applications. By then our customer base used to deploy on OPENSTEP Mach, Windows NT and Solaris systems.
Due to the fact that on one side we had only a few customers but quite a lot of projects going on, and that on the other side we had a very promising product in development we wanted to focus on, we decided that time has come where we needed a bug reporting system, which was meant for internal use only. So our goal was to create a system that would let us

- collect and maintain reports sent to us either by external sources, that is customers, or provided by us, the Uptime developers.
- optimise project management by having better control over all open issues.
- react faster on questions made by customers or our non technical staff.

We *never* had the intention to provide any information to the outside world.

What went poorly

First, BURST was designed to be used by smaller companies like Uptime Inc. It never was a goal to provide a web frontend, since our developers did not like that - we wanted to use a *real application*. The web as an application plattform was and still is a cripple... Looking back, I think this was a mistake, though. Not because I would like BURST to be used by big companies that use web based intranets or because the quality of web based applications is much higher by now, but because of the fact that more and more developers don't work at the same place anymore, who of course still

need to have access to a central repository. And this is what the web is all about. The result is that too much time was spent in optimising the UI, which in fact made a faster development cycle impossible.
Second, while it was planned to design and implement BURST within only a few months, this goal definitly failed, mainly due to 3 reasons:

Problem 1: Apple

Around 1998 Apple still planned to continue its crossplattform strategy with their new operating system called Rhapsody, which was planned to be sold on PowerPC and x86 machines. This was at the time I began working on BURST.
In the meantime Apple dropped its plans and focused on a PPC-only strategy, that is Mac OS X (Server). They made life for OPENSTEP based companies like us harder and harder by successively changing their plans without really communicating them. We - and a lot of other companies - lost faith in them and changed our business strategy completely, which meant we gave up custom development and focused on our awarded archiving and retrieving system *ARTS*. Therefore we shifted away from OPENSTEP towards NT and Intranet based application development (WebObjects) - while BURST still was an OPENSTEP desktop application.. I really have learned that it is a *Good Thing* not to be dependant on one vendor's technology only!

Problem 2: Timeframe

I began working on the project in 1998 as a side project of my work at Uptime. This never changed until I have finished the first version of BURST, which again meant I could not really focus on it. And it's unfortunately obvious that BURST had a lower priority than externally paid projects. This caused a major delay in the project's implementation stage.

Problem 3: The distance

Uptime Inc was and is located, as mentioned before, in Zürich and in Bern. While most of our programmers worked in Zürich, I began working in Bern more and more. While this had the advantage of traveling less, it also raised the problem of communication, one of the most important parts in a software development process. There was no daily discussion about the needs and 'nice to have's'anymore. So in one way I lost touch with my 'customers'. Fortunately much has been specified during the analysis of the problem already, therefore this situation could be handled more or less 'painless'.

What went right

First and most important, I succeeded in creating a bug reporting system that can be and is used by companies like Uptime Inc, Studiosendai.com and others. The software development award I have won at the MacWorld in January 2000 only seconds that.

Second, I believe we managed to focus on the most important parts of such a system, without losing *the big idea*. The design of the system allows a programmer to make changes in an easy way without breaking other parts.

Third, the OpenStep technology was the right technology to be used, despite of all the bad side effects mentioned above. It is a very powerful and robust environment which allows very fast development cycles (including UI prototyping). I couldn't have realised such a complexity and feature list using another API in such a timeframe, I suppose.

And last but but least, it was fun working on BURST!

Conclusion

So to say, while I definitly would focus on a web based UI solution these days, I think I would design most of the core functionality the same way as I have done. I would therefore concentrate myself more on a client/server based solution, which would offer a greater flexibility in realising different interfaces using standardised APIs to fit different needs.

Furthermore I have learned that one of the most important things in developping software for a *real customer* is to be faster than possible. The faster the single development cycles are, the faster one can react on different situations.
A product can be as good as it wants to be, if it comes late, it's late, if it misses the most urgently needed functionality, it misses its goal (luckily this did not happen to BURST). Therefore I would not only focus on rapid UI prototypes anymore but on rapid functionality prototypes. This perhaps also means using different working models, eg some kind of *extreme programming* or working more tightly together with another programmer (if possible). And while it seems that it is almost impossible to meet a deadline in computer business, it should be made sure that it *can* be done, at least! Setting up a realistic timeframe and feature list seems to be a question of 'realworld experience', though...

BURST

A Bug Reporting System for OpenStep compliant systems

Usermanual

Design and implementation 1999 by
Philippe C.D. Robert, robert@iam.unibe.ch

for

Prof. Dr. O. Nierstrasz, oscar@iam.unibe.ch
Software Composition Group (SCG)
Institute of Computer Science and Appl. Mathematics (IAM)
University of Berne, Switzerland

and

Uptime Object Factory Inc
Matthias Heubi, mh@uptime.ch
Technoparkstrasse 1
8000 Zürich

Contents

List of Figures

1 Introduction

This is the usermanual applying to BURST 0.9. Please be aware that this is the first public release of this package, it's more than likely that there are still a lot of nasty bugs around. I'd be glad to receive any comments or bug fixes that may then be included in the next release.

1.1 Requirements and license

In order to use BURST, you need to fulfill the following requirements:

- Mac OS X Server or Windows NT/98 running WebObjects 4.x
- A database providing an EOF 3.x adaptor, currently Oracle, FrontBase and OpenBase are supported.
- A lot of patience...

Be aware that BURST is an open project, it is not finished yet and won't be so for a long time. Everybody can contribute code, reports, better documentation or terrific ideas to make it more usable.

2 Installation

In order to use BURST you have to go through some initial installation stages, when you run the application for the first time.

2.1 Setting up the database

Of course you must have an appropriate database to run BURST Manager. It is assumed here that you know the EOModeler application. So there really should not be a lot of problems when setting up the database from within EOModeler. The following steps have to be done in order to set up a correct database layout:

1. First, create a database for BURST either by hand or with a database management application provided by your database.

2. Run EOModeler by double clicking *burst.eomodeld*, which can be found in the BTEOSubsystem subproject directory.

3. Choose *Generate SQL...*, found in the menu *Property*, and execute the automatically generated SQL script.

4. If there are no errors, you are ready to run BURST Manager!

If you don't use the same database as me - OpenBase 6 - then you must first convert the eomodel file. You can do this with the EOModeler application as well!

2.2 Installing the application

When Manager.app is launched for the first time, no users are provided but one administrator login with the following login and password:

login: admin

password: admin

It's better to change this password right after login! Once the administrator has logged in, the initial working environment should be set up in order to really use BURST, that means you must configure your products, their versions and customers as well as the BURST maintainers.

Figure 1: Manager Login Panel

3 Administration

This chapter describes how to administrate the working environment of BURST.

3.1 Adding a new User

To add a new user, open the maintainer administration panel and fill in the needed attributes of the new user. At least a unique login name is required. To store the new maintainer to the database, click *Add...* on the lower left side of the window. At this point the new authorised user has not yet a password of its own! This must be set after the first login!

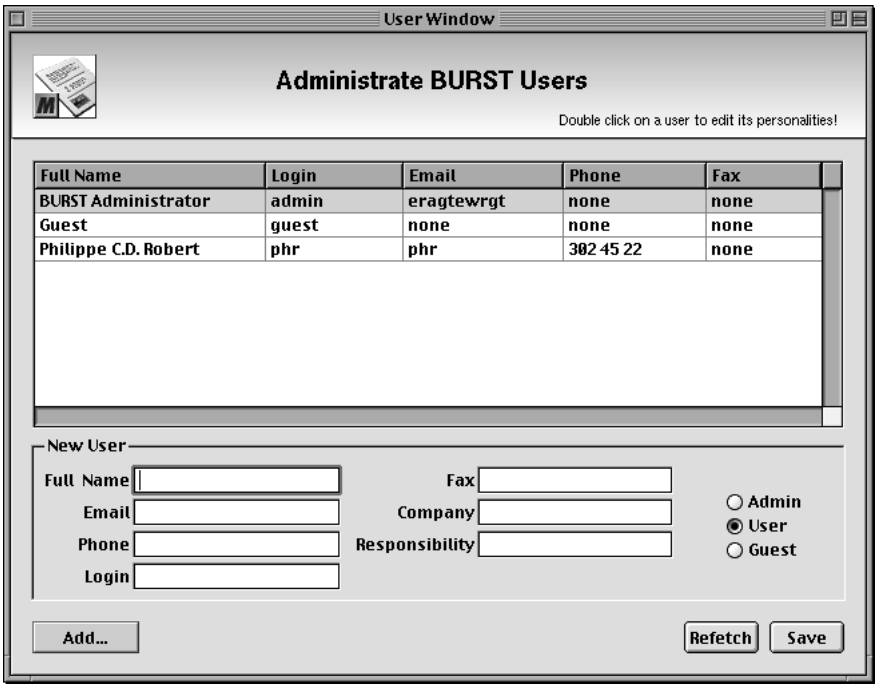

Figure 2: Manager Maintainer Window

When double clicking on a maintainer entry in the main tableview, a window containing more detailed information about the currently selected maintainer will appear.

Be aware that only administrators can add new maintainers or change their attributes!

3.1.1 Access Rights

A user can belong to one of three user groups, that is administrator, user or guest. *Guests* cannot maintain any reports nor can they administrate something, they are what the name says, just guests. *Users* can be maintainers of reports but they do have no access rights to administrate anything. *Administrators* can maintain reports as well as maintain the system.

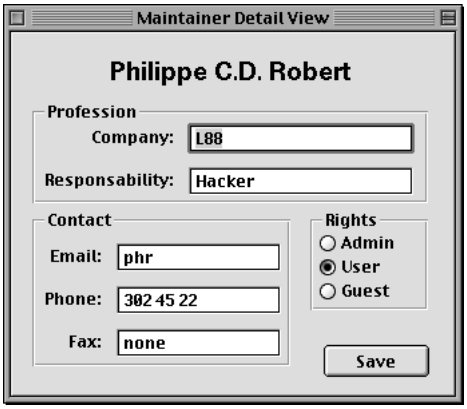

Figure 3: Manager Maintainer Detail Window

3.2 Adding a new product

To add a new product that will be tracked by the incoming reports, open the product window and add a new product by choosing New. Then you must add at least one version to this new created product! After this has been done you can fill in some comments describing the new product itself as well as every product version.

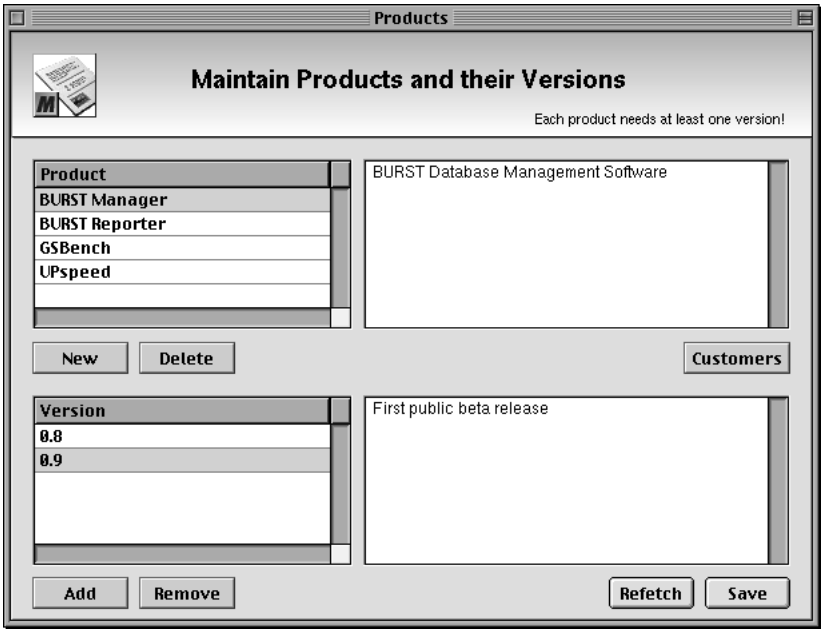

Figure 4: Manager Product Window

To remove a product and its versions, simply click on *Delete*. You shouldn't delete a product that owns any reports, first remove all those

reports!

Be aware that only administrators can add new products and versions!

3.2.1 Adding/Removing Product Versions

To add a new version or alter the attributes of an existing one, open the product window and click on *Add* on the lower left side of the window. A new version is then added to the currently selected product. You can now give it a name and a description.
To remove an existing version simply click on *Remove*. Be aware that each product must have at least one version!

Don't forget to save the newly made changes by clicking on 'Save'!

3.3 Adding/Removing Reference Lists

To add a new logical reference list, open the list window and click on *Add*. A new nameless list is then added. You now have to give it a name and add some comments.

Figure 5: Manager List Window

Don't forget to save the newly made changes by clicking on 'Save'!

3.4 Adding/Removing Customers

To add a new customer, simply open the customer window and click on *New Customer*. A new window then appears where you can fill in the needed

information. To save the customer permanently, click *OK* on the lower right side of this window.

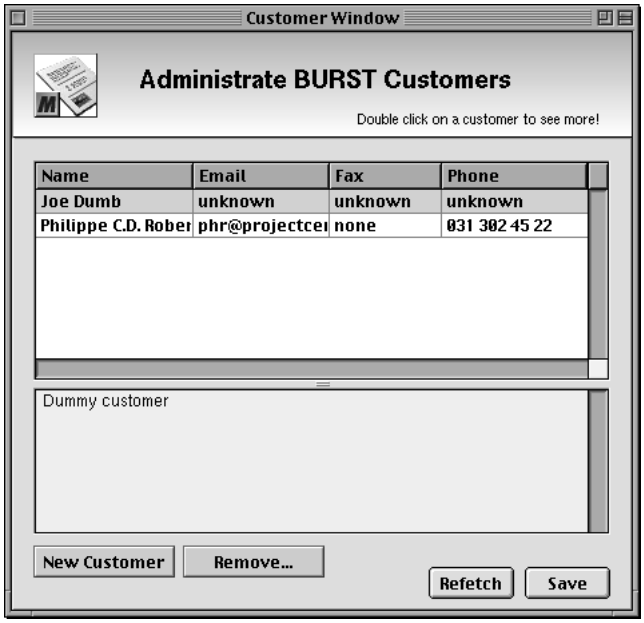

Figure 6: Manager Customer Window

Be aware that only administrators can add or remove customers!

3.4.1 Assigning Products to a Customer

To assign one or more products to a specific customer, open the customer window and double click on the selected customer entry. A new window appears where you click on *Edit* on the lower left side of the window. You now can add or remove products to/from the current customer.

3.5 Editing Personal Preferences

When double clicking on a maintainer entry in the maintainer window, a new window containing more detailed information about the selected maintainer should appear. At this place the maintainer's attributes can be altered at any time. Don't forget to save the newly made changes by clicking on *Save*! You also can change your personal attributes in the preferences panel.

Be aware that only administrators can add new maintainers or change their attributes!

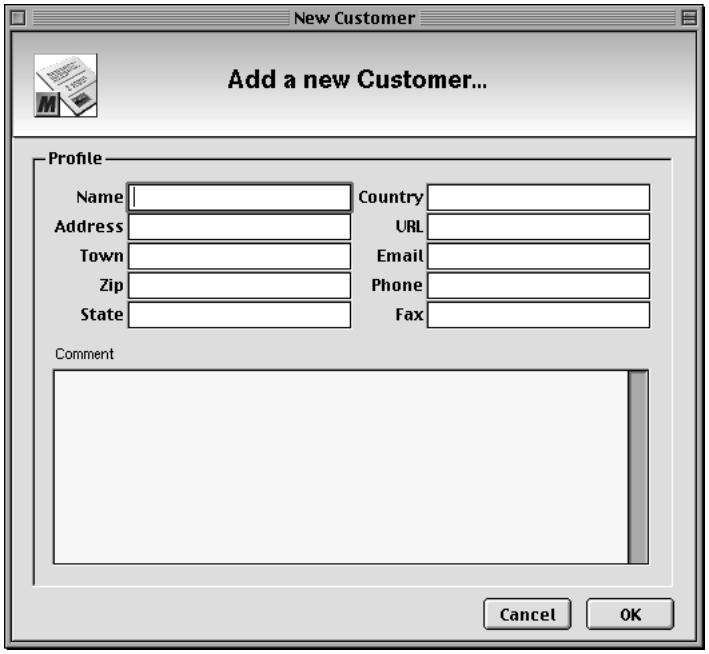

Figure 7: Adding a new customer

3.6 Changing the password

To change you personal password, open the preferences panel and select *Login Information* from the popup button. Next type in the old and the new password (twice!). After clicking on *Change*, the new password is stored in the database.

4 Working with BURST Manager

This chapter describes the most common tasks that need to be done when working with BURST Manager.

4.1 Creating new Reports

To create a new report from within Manager.app, press *Command + n* or click on *New* in the File menu. Doing so will open a new window to create a new report.
You now can type in all the needed information as well as drag and drop attachements to the new report. Further you must specify a maintainer for the report and of course the appropriate product. The deadline date and the report number are optional attributes.
If needed the edited report can be already assigned to one or more existing reference lists by clicking on *Lists...*. However this can also be done at any

later moment.
To add the report to the data repository simply click on *Save*. You now will see the new report appearing in the main repository window.

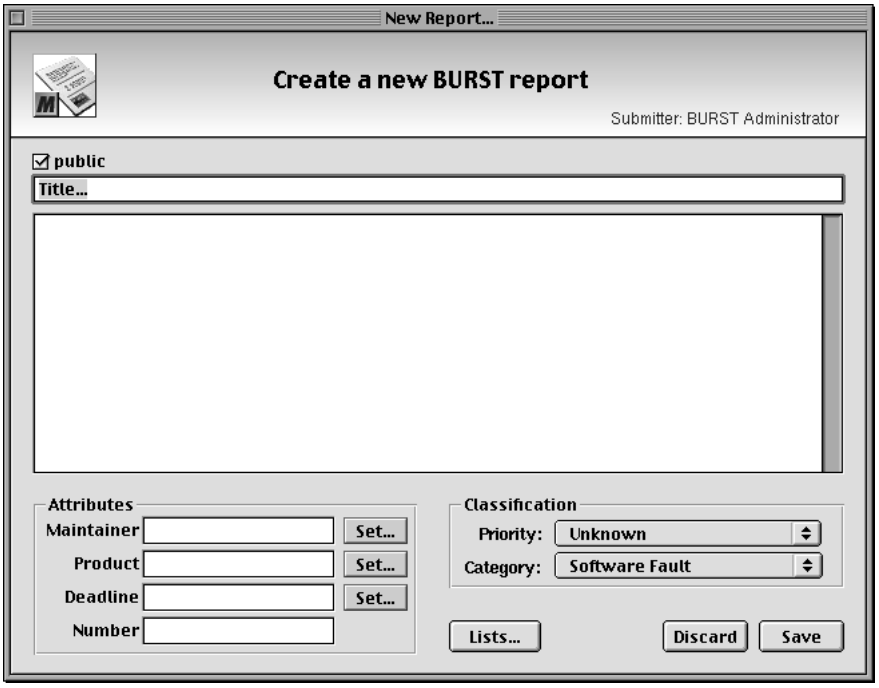

Figure 8: Creating a new report

When adding a report to the repository from within Manager.app, the current user is tracked as the submitter of the bug. You can't change this behaviour in this release!

4.2 Displaying Reports

For daily work it is highly recommended to have the ability to sort, qualify and group the existing reports. The user can achieve those needs on the main repository window. Additional information can then be displayed using the inspector window.

4.2.1 The Repository Window

The repository window is the main window where every report stored in the database can be viewed as an entry in the tableview and the associated description view.
To qualify the stored reports the user can specify which reports have to be displayed by choosing an appropriate criterium:

- reports belonging to a certain product

- reports belonging to a certain maintainer
- reports belonging that are referenced by a certain list

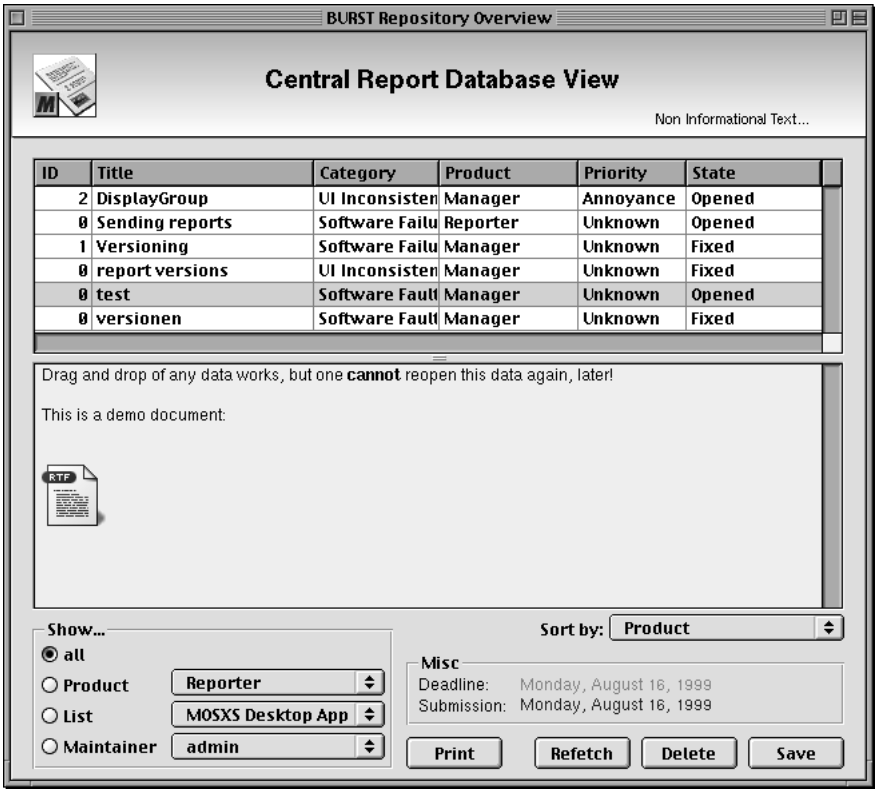

Figure 9: Main Repository Window

Additionally the displayed reports can be sorted by

- Category
- ID
- Product
- Priority
- State
- Submission
- Title
- Deadline

When double clicking on an report entry or choosing *Inspector* from the *Tools* menu, the inspector window appears displaying detailed information about the selected report.

The inspector always shows information about the currently selected report!

4.2.2 The Inspector Panel

The inspector window has 5 subviews, each one of them can be viewed by clicking on the appropriate popup button on the top of the window.

Figure 10: Manager Report Inspector

The 5 subviews are:

General: Information about the submission date, the optional deadline, the fixed version, the state of the report etc. are shown here. The user can made modifications here as well, i.e. changing the state of a report, its priority or the fixing date. When a report gets *Fixed*, an email can be sent to the submitter of the report to make a notification about the recent fix.

Lists: All reference lists that the selected report belongs to are shown here. By clicking on the *Edit* button, a list panel appears where the user can add or remove the current report from one or more lists.

Maintainer: The maintainer of the report is shown in this view. If somebody changes this attribute a panel appears where the user can send an email to notify the new maintainer about the recent changes.

Submitter: The maintainer of the report is shown in this view. This attribute can't be changed anymore!

Customers: All customers of the associated product are shown in this view.

4.2.3 SQL Queries

More experienced users have the chance to use SQL statements in order to determine which reports were displayed.

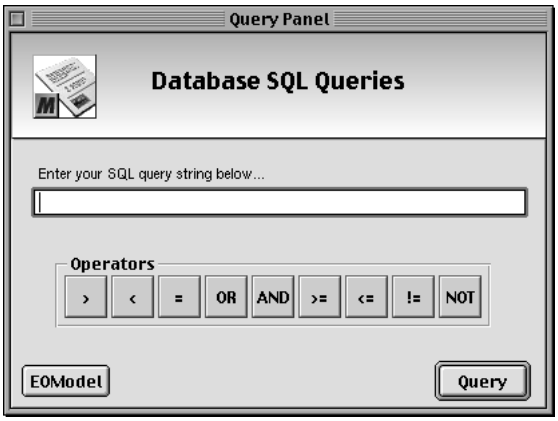

Figure 11: Manager SQL Query Window

To give an idea how such a statement may look like, here comes a simple example that selects all reports targeting product 'A' that were submitted by submitter 'B':

SQL string: product.productName='A' AND submitter.fullName='B'

To get an idea about the database layout, click on *EOModel* on the lower left side of the window.

4.3 Customers of a specific Product

Sometimes it is useful to know all the customers of a specific product. You can get those names in two ways!
Either you select a report of the wanted product and open the *Report Inspector*, there you select *Customers* from the popup button and a list containing all customers of the target product appears.
On the other hand you may also want to open the product window and select the target product. Then when clicking on the *Customers* button a window with a more detailed list of all customers will appear.

4.4 Printing reports

To print either a list of all currently displayed reports or just the selected one in a detailed manner, click on the *Print* button at the bottom of the main window.

When printing in list mode, all reports that are currently visible are printed!

4.5 Logging

Every transaction and/or event is logged to the database. So you always have control over every so made change to the database! To see those stored log entries, simply open the log window by choosing *Logging...* from the *Tools* menu.
To delete the currently stored log entries click on the *Delete* button on the log window. Of course, deleting is logged as well...

Be aware that only adminstrators can delete log entries!

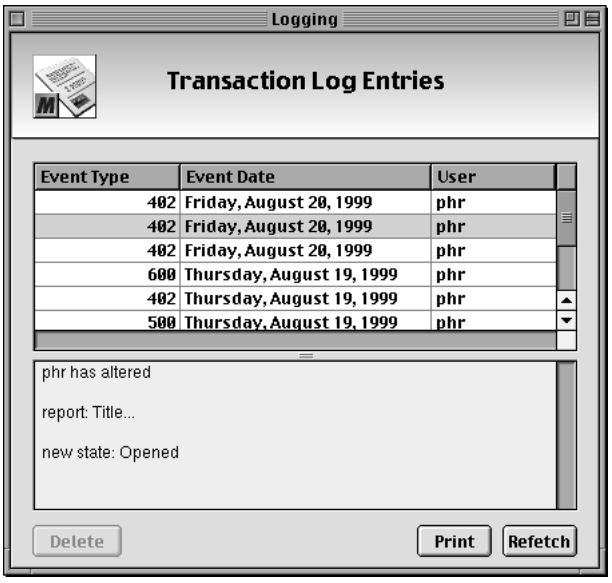

Figure 12: Manager Log Window

5 BURST Reporter

BURST Reporter is a tool to send bug reports to a central repository by using email as the transmission protocol (as published in the RFC 822).

5.1 Initial launch

Before start sending any reports, you should take a break and save your personalities using the preferences panel. This is not to track you down as other companies do, it's because developers should be able to contact the submitters of the reports, either to send you a notification about their fix or workaround of the described problem or to ask you more questions about the submitted misbehaviour.

It is highly recommended to customise your personalities, although not required!

5.1.1 Preferences Panel

Using the *Preferences* a user may type in his or her personalities. Additionally it can be selected which email format and which browser description file should be used. Note that it doesn't make much sense in general to use another browser description file as comes with the application.

If you have chosen to use the binary email format, you should be aware that by then only automatic imports into any BURST repositories can ever reproduce the correct information out of your reports!

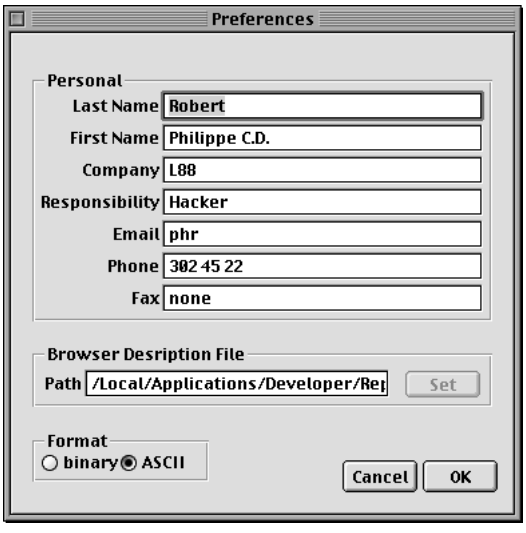

Figure 13: Reporter Preferences Panel

5.2 Composing Reports

The one and only purpose of Reporter.app is to compose reports and send them to a specific destination address. This is done by using the main *Composer* window.

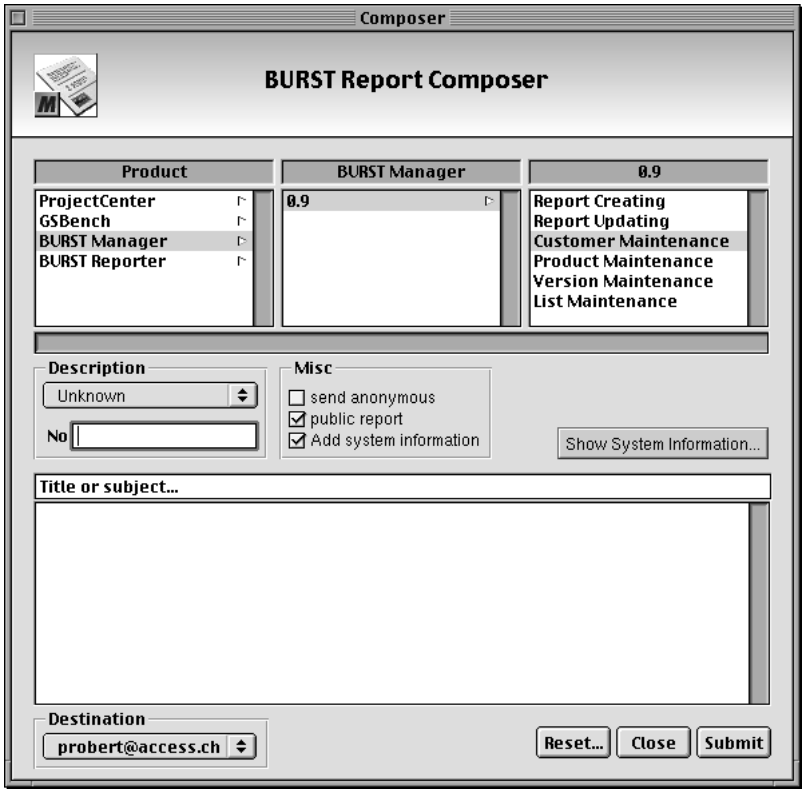

Figure 14: Reporter Composer Window

Users can categorise their reports using predefined attributes such as

- Unknown
- Suggestion
- Annoyance
- Avoidable/Unavoidable
- Performance
- App crash
- System crash
- Log out

- Security

The most important piece of information although is of course the report description itself. Attachements can be included as well as customised fonts and colours can be used to describe in best ways what exactly initiated the report. The target product, its version and a more detailed category of the report can be chosen using the browser on the top of the window. The report is finally sent to the specified destination address by clicking on *Submit* on the lower right corner of the *Composer* window.

You can send a report without submitting your personalities by activating send anonymous on the Composer *window.*

5.2.1 Additional Information

If needed you can include more information about your system into the report. To control which information is submitted click on *Show System Information....*

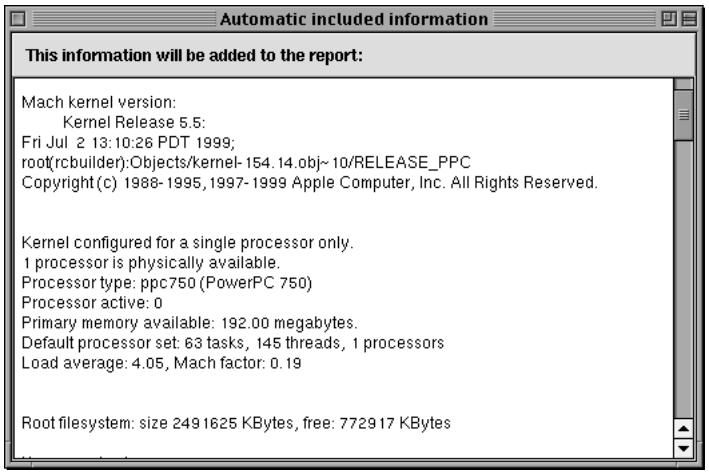

Figure 15: Additional Information

Attachements are only sent properly to the destination when sending in binary format, of course!

5.2.2 Storing Reports

You may want to save a report to the local disc for later use. You can do this be simply click on *Save As...* in the *Report* menu. You will be able to reuse those reports at any later moment by loading them from disc!

6 Outlook

6.1 BURST Core Features

In the next release, BURST will be enhanced in several directions:

- Better user experience
- Customer support management
- Project time tracking capabilities
- Additional web frontend, realised in *WebObjects*

Future versions of BURST won't be part of my work at the University of Berne, anymore.

6.2 Supported Plattforms

Due to changes in Apple's crossplattform strategies, BURST won't be available for WebObjects 4.5 on Microsoft Windows or any other systems than Apple's Mac OS X anymore. It may be possible though, that BURST will be ported to GNUstep, the free implementation of the OpenStep API.