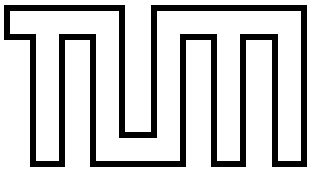


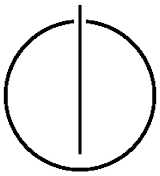
FAKULTÄT FÜR INFORMATIK

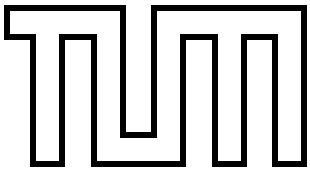
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatik

Unterstützung von Performance-Optimierungen durch Analyse von gesampleten Mainframe-Verbrauchs-Messungen.

Stefan Laner





FAKULTÄT FÜR INFORMATIK

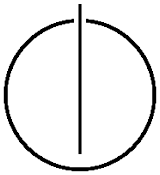
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatik

Unterstützung von Performance-Optimierungen durch Analyse von gesampleten Mainframe-Verbrauchs-Messungen.

Supporting performance optimizations by analyzing sample based consumption measurements on mainframe systems

Autor:	Stefan Laner
Aufgabensteller:	Prof. Dr. Florian Matthes
Betreuer:	M.Sc. Matheus Hauder
Datum:	22. März 2013



Ich versichere, dass ich diese Master's Thesis selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 22. März 2013 Stefan Laner

Danksagung

An dieser Stelle möchte ich mich für die Unterstützung der Firma itestra GmbH bedanken, die mir die Einblicke in das Thema „Performance-Optimierung am Mainframe“ überhaupt erst ermöglicht hat.
Zudem bedanke ich mich bei der Firma Compuware, die mir freundlicherweise die Genehmigung zur Verwendung der Grafik in Kapitel 5.2 erteilt hat.

Zusammenfassung

Großunternehmen und Behörden betreiben auch heute noch viele Anwendungen auf IBM-Mainframe-Systemen. Mit alternder, an aktuelle Anforderungen unzureichend angepasster Software und stetig wachsendem Datenvolumen treten häufig Performance-Probleme auf. Unzureichende Performance hat auf Grund nutzungsbasierter Lizenzmodelle, die im Mainframe-Bereich eingesetzt werden, einen direkten Einfluss auf die Betriebskosten einer Anwendung. Eine nachhaltige Senkung dieser Kosten kann durch eine gezielte Performance-Optimierung kritischer Code-Pfade erreicht werden. Dazu müssen kritische Programmteile identifiziert werden, deren Optimierung einen attraktiven ROI bietet.
Zur Identifikation solcher Programmteile werden Informationen darüber benötigt, welcher Programmteil wieviel CPU-Nutzung verursacht. Dazu werden Programmausführungen mit speziellen Werkzeugen beobachtet. Die Ergebnisse einer solchen Messung sind umfangreiche, meist textbasierte Reports, welche in der Regel manuell ausgewertet werden müssen. Diese Auswertung ist auf Grund des Umfanges der Reports sehr zeitaufwändig. Einige Performance-Probleme spiegeln sich in den Messdaten in Form bestimmter Muster wider (Anti-Pattern), deren manuelle Suche jedoch ebenfalls sehr aufwändig ist und für jedes solche Anti-Pattern einzeln erfolgen muss. Zudem liegen inhaltlich zusammengehörige Informationen teilweise verteilt an verschiedenen Stellen innerhalb solcher Reports vor und müssen zur Analyse zunächst zusammengebracht werden.
Im Rahmen dieser Arbeit wird ein Analyse-Werkzeug entwickelt, mit dem solche Reports eingelesen und die darin enthaltenen Daten in aufbereiteter Form dargestellt werden können. Das Werkzeug vereinfacht die Navigation innerhalb dieser Daten, ermöglicht eine schnelle Identifikation von Programmteilen mit besonders hohem CPU-Bedarf und verbindet zusammengehörige Informationen innerhalb der Messdaten. Eine automatische Suche nach einer Liste von definierten Anti-Pattern erspart einem Performance-Experten das aufwändige Durchsuchen der Textreports nach jedem einzelnen dieser Anti-Pattern.
Mit einer Umfrage an sechs solche Experten wird schließlich der Nutzen einer Softwareunterstützung für diesen Analyseprozess evaluiert.

Abstract

Even today large-scale enterprises and public agencies still run many applications on IBM Mainframe systems. Aging software which is not adapted sufficiently to current requirements and constantly growing data volumes frequently leads to performance issues. Because of usage-based licencing fees, which are commonly used in Mainframe environments, insufficient performance has a direct impact on an application's operational costs. A sustainable reduction of those costs is achievable by targeted performance optimization of critical code paths. Therefore critical program parts, whose optimization offers an attractive ROI, need to be identified.

To identify those program parts information about how much CPU usage is caused by each program part is necessary, therefore program executions are observed with special software tools. The outputs of such measurements are extensive, usually textual, reports which usually have to be analyzed manually. Those analyses are very time consuming due to the reports' large extent. Some performance problems are indicated within such measurement data by certain patterns (or anti-patterns). Manually searching for these is very time consuming, because each anti-pattern has to be looked up separately. Moreover related information may be spread over the whole report and has to be collected before being analyzed.

Within the scope of this Master's Thesis an analysis tool is developed, which is able to parse such reports and present the collected information in an appropriate way. This tool eases navigation within those data, allows for quick identification of program parts causing high CPU demand and joins related information from different parts of the report. The tool automatically looks for each defined anti-pattern, sparing performance experts laborious look-ups for each anti-pattern within those textual reports.

A survey answered by six such experts evaluates the benefit of such tool support for the performance analysis process.

Inhaltsverzeichnis

Überblick über diese Arbeit

I: Performance-Messungen am Mainframe

KAPITEL 1: EINFÜHRUNG

Dieses Kapitel gibt eine Einführung in das Thema dieser Master's Thesis. Die Relevanz des Themas wird kurz erläutert, ebenso die Problemstellung, die mit dieser Arbeit behandelt wird. Zudem wird der Rahmen der Arbeit abgegrenzt.

KAPITEL 2: ÜBERBLICK

Dieses Kapitel beschreibt allgemeine Grundlagen zum Mainframe und stellt statische und dynamische Verfahren zur Performance-Analyse vor.

KAPITEL 3: VORHANDENE ARBEITEN

Dieses Kapitel beschreibt Gemeinsamkeiten und Unterschiede vorhandener Arbeiten in den Bereichen Profiling, Performance-Analyse und Performance Anti-Pattern im Vergleich zu dieser Arbeit.

II: Konzepte zur Performance-Analyse

KAPITEL 4: ANFORDERUNGEN

Dieses Kapitel beschreibt Anforderungen an ein Werkzeug zur Unterstützung von Performance-Analysen. Die Anforderungen werden getrennt in Analyse-Anforderungen und Usability-Anforderungen.

KAPITEL 5: ANALYSE VON STROBE-REPORTS

Dieses Kapitel beschreibt den Aufbau von Strobe-Reports, sowie den vom Hersteller von Strobe vorgeschlagenen Analyseprozess. Zudem wird ein Datenmodell vorgestellt, welches für die Speicherung von Performance-Daten entwickelt wurde.

KAPITEL 6: ANTI-PATTERN

Dieses Kapitel beschreibt detailliert vier Anti-Pattern, die negativen Einfluss auf die Performance von Anwendungen auf Mainframe-Systemen haben.

III: Werkzeug zur Performance-Analyse

IV: Zusammenfassung

Teil I.

Performance-Messungen am Mainframe

1. Einführung

Die Verbreitung von IBM Mainframe Systemen im Umfeld von Großunternehmen und Behörden ist entgegen früherer Prognosen nach wie vor sehr bedeutend [1]. So nutzen zum Beispiel laut einem Artikel von Bob Thomas [20] 90% der Fortune 1000, also der 1000 umsatzstärksten US-Unternehmen, Mainframes. Die Bedeutung der Mainframes zeigt sich auch darin, dass IBM noch immer in deren Weiterentwicklung investiert und erst vergangenes Jahr eine neue Version angekündigt hat [3].
Während der Betriebszeit einer Software ändern sich meist auch die an sie gestellten Anforderungen. Dies können beispielsweise neue Funktionen oder wachsende Datenmengen sein. Wird die Software an diese geänderten Rahmenbedingungen nicht oder nur unzureichend angepasst, treten häufig Performance-Probleme auf. Diese Beobachtung beschrieb Parnas [14] bereits 1994 als „altern“ von Software. Für den Betrieb einer IBM Mainframe-Umgebung fallen laufende Software-Lizenzgebühren an, welche dem Betreiber von IBM in Abhängigkeit der CPU-Nutzung in Rechnung gestellt werden. Der CPU-Bedarf einer Anwendung hat somit einen unmittelbaren Einfluss auf die Höhe dieser Lizenzkosten – eine Senkung des CPU-Bedarfs bewirkt entsprechend eine Senkung der Lizenzkosten.
Eine Möglichkeit zur Senkung der von solch problematischen Anwendungen verursachten Lizenzkosten wäre es, diese Anwendungen abzuschalten und gegen effizientere Neuentwicklungen auszutauschen, ggf. in Verbindung mit einer Migration auf andere Plattformen. Allerdings treten Performance-Probleme, welche sehr hohe Lizenzgebühren verursachen meist an zentralen Punkten innerhalb einer Applikationslandschaft auf, welche zur Erfüllung der Geschäftstätigkeit der jeweiligen Unternehmen notwendig sind. Das Risiko, dass ein Problem bei der Ersetzung der Software den Geschäftsbetrieb behindert oder gar zum erliegen bringt ist deshalb sehr hoch. Zudem besitzen diese Softwaresysteme, die oftmals Umfänge in der Größenordnung mehrerer Millionen LoC haben, auf Grund der darin eingeflossenen Entwicklung einen Gegenwert, der bei einer Neuentwicklung erneut aufzubringen wäre. Eine Wartung und Optimierung socher „Altsysteme“ ist deshalb meistens der einzige sinnvolle Ansatz.
Um solche Optimierungen durchzuführen, müssen performance-kritische Programmteile innerhalb der zu optimierenden Anwendung identifiziert werden, deren Optimierung einen möglichst attraktiven ROI bietet. Dazu werden Informationen darüber benötigt, wie sich der CPU-Bedarf auf die einzelnen Bestandteile der Anwendung aufteilt. Um diese Informationen zu ermitteln, werden spezielle Werkzeuge eingesetzt, welche die tatsächliche Ausführung eines Programms „beobachten“ und daraus umfangreiche Reports erzeugen. Diese Reports sind in der Regel Textdateien – derzeit gängige Praxis für deren Analyse ist eine manuelle Auswertung. Diese manuelle Auswertung ist sehr zeitintensiv, besonders auf Grund des großen Umfangs der erzeugten Reports. Zudem müssen Performance-Experten häufig Informationen, welche für die Analyse zusammengefasst werden müssen, an verschiedenen Stellen innerhalb der Reports suchen und manuell zusammenfassen. Eine systematische Suche nach bestimmten Mustern innerhalb der Daten eines solchen

Reports, welche Hinweise auf gewisse definierte Performance-Probleme liefern können, muss ebenfalls manuell und deshalb unter hohem Zeiteinsatz erfolgen. Eine genauere Beschreibung dieser Problematik erfolgt im Abschnitt 1.2.

1.1. Relevanz

Die gesamten Lizenzkosten für den Betrieb einer IBM-Mainframe-Umgebung belaufen sich bei Großunternehmen oftmals auf einige hundert Millionen Euro pro Jahr. Einzelne Anwendungen verursachen dabei bereits Kosten im Umfang mehrerer Millionen Euro jährlich. Eine Optimierung einzelner Anwendungen bietet somit bereits Potential für attraktive Kosteneinsparungen. Da diese Kosteneinsparungen über die gesamte Betriebszeit der Anwendung hinweg stattfinden, sind solche Optimierungsmaßnahmen auch langfristig nützlich für die betreibenden Unternehmen.
Eine Umfrage von Computerworld [4] nennt ineffizienten Code in Applikationen als eine der Hauptursachen für gestiegenen Ressourcenbedarf und bestätigt damit die weite Verbreitung potentiell optimierungsfähiger Anwendungen.
Der grundlegende Ansatz dieser Arbeit - die Überführung von Performance-Daten in ein Meta-Modell und dessen Auswertung und Visualisierung zur Identifikation performancekritischer Programmteile - lassen sich auch ausserhalb des Mainframe-Umfelds einsetzen. Nicht nur beim Mainframe fallen Lizenzgebühren in Abhängigkeit der CPU-Nutzung an – viele Hersteller von Unternehmenssoftware lizenzieren ebenfalls nutzungsabhängig. Verringerter CPU-Bedarf führt jedoch auch auf anderen Plattformen zu Kosteneinsparungen, z. B. durch verringerten Energieverbrauch für Betrieb und Kühlung oder dadurch, dass bei cloud-basierten Rechenzentren weniger Ressourcen gebucht werden müssen.

1.2. Problembeschreibung

Nicht optimal implementierte Anwendungen benötigen viel Prozessorleistung und erzeugen dadurch vermeidbar hohe Kosten auf einem Mainframe oder benötigen sehr lange Zeit für ihre Ausführung. Um solche Anwendungen zu optimieren, müssen zunächst die kritischen Stellen bzgl. CPU- oder Wartezeit identifiziert werden. Dazu werden (meist produktive) Ausführungen der zu optimierenden Anwendung mit einem Sampling-basierten Verfahren (siehe Abschnitt 2.2.2) gemessen. Die Ergebnisse von solchen Messungen sind sehr umfangreiche, meist textbasierte Reports. Performance-Experten müssen aus diesen Reports ableiten, welches die performance-kritischen Stellen sind.
Dieser Schritt ist jedoch sehr zeitintensiv, da die Reports eine große Menge an Informationen liefern, welche zunächst gefiltert und aufbereitet werden müssen. Oftmals sind Informationen, welche für eine Analyse gemeinsam betrachtet werden müssen (z. B. ein Aufruf eines Systemmoduls und dessen Ursprung) über verschiedene Stellen innerhalb eines Reports verteilt. Diese Stellen müssen für eine Analyse dann einzeln gesucht und die jeweiligen Informationen kombiniert (z. B. aggregiert) werden. Innerhalb der Daten solcher Reports spiegeln sich Performance-Probleme häufig durch das Auftreten bestimmter Muster (Anti-Pattern) wider, die jedoch ebenfalls manuell gesucht werden müssen. Dazu müssen für jedes solche Anti-Pattern einzeln die notwendigen Informationen aus dem Report gesammelt und teilweise auch aufbereitet werden.

Derzeit fehlt eine Werkzeugunterstützung, damit Performance-Experten an Hand der Daten solcher Reports schnell Optimierungspotentiale erkennen und einschätzen können. Die Aufbereitung der Performance-Daten und eine Suche nach einer definierten Liste von Anti-Pattern könnte ein derartiges Werkzeug automatisch vornehmen.

1.3. Schwerpunkt der Arbeit

Schwerpunkt dieser Arbeit ist es, ein Analyse-Werkzeug zu entwickeln, welches wie im Abschnitt 1.2 beschrieben, Performance-Experten eine schnelle Erkennung performancekritischer Teile einer Anwendung ermöglicht. Dazu sollen die Daten des Messwerkzeuges „Strobe“ eingelesen werden. Das Analyse-Werkzeug soll auf Basis dieser Daten für eine Analyse notwendige zusammenfassende Darstellungen erzeugen und automatisch, mittels der Anwendung definierter Regeln, Anti-Pattern in den Daten auffinden. Dabei richtet sich der Fokus dieser Arbeit auf eine Analyse der CPU-Nutzung, nicht der Laufzeiten der gemessenen Programme.

1.4. Methodik und Überblick

Im Teil I werden zunächst Grundlagen über Mainframes erläutert und ein Überblick über Performance-Analyseverfahren gegeben. Zudem beschreibt das Kapitel 3 vorhandene Arbeiten im Umfeld von Profiling und performance-relevanter Anti-Pattern und diskutiert Schnittmengen und Unterschiede mit der vorliegenden Arbeit.
Der Teil II beschreibt zunächst eher abstrakt die Konzepte zur Analyse von Performance-Reports. Dazu werden im Kapitel 4 zunächst die Anforderungen an ein Programm zur Unterstützung von Performance-Experten aufgeführt. Die Anforderungen wurden in Gesprächen mit Performance-Experten erfasst. Das Kapitel 5 beschreibt den Aufbau von Strobe-Reports und den von dessen Hersteller Compuware vorgeschlagenen Arbeitsablauf zur Analyse dieser Reports. Zudem wird ein Datenmodell vorgestellt, welches entwickelt wurde, um die in Textform vorliegenden Informationen aufzunehmen und leichter analysierbar zu speichern. Der konzeptionelle Teil endet mit einer Beschreibung ausgewählter, performance-relevanter Anti-Pattern und erklärt, wie diese aus Profilingdaten auf Basis des unter 5.3 beschreibenen Datenmodells automatisiert gefunden werden können.
Im Teil III werden anschließend Aufbau und Implementierungsdetails des entwickelten Software-Werkzeugs beschrieben. Dieser Teil gliedert sich in 3 Kapitel. Zunächst wird der Parser zum Einlesen der Strobe-Reports und zur Überführung in das Datenmodell behandelt (Kapitel 7), in Kapitel 8 die Implementierung des Analyse-Werkzeugs selbst. Im Kapitel 9 wird dann das Werkzeug mittels Experteninterviews evaluiert.
Der Teil IV fasst schließlich die Ergebnisse dieser Arbeit zusammen und gibt einen Ausblick auf darauf aufbauende Arbeiten.

2. Überblick

Das folgende Kapitel beschreibt zunächst das Umfeld für die betrachteten Performance-Optimierungen. Dazu werden einige Grundlagen und Begriffe zum Mainframe sowie die beiden verschiedenen Ansätze zur Erfassung von Performance-Daten - statische und dynamische Analyseverfahren - erläutert.

2.1. Mainframe

Der Begriff „Mainframe“ wurde ursprünglich verwendet, um Großrechner im Unternehmensbereich, welche viele Nutzer bedienen, von Computern für einzelne Nutzer zu unterscheiden. Heute wird der Begriff hauptsächlich für IBM „System z“ Systeme verwendet - so auch im Rahmen dieser Arbeit. Zentrale Eigenschaften moderner Mainframes sind:

- Multi-User Fähigkeit
- Hochverfügbarkeit
- Hohe Durchsatzraten
- Hohe Sicherheit

Zur Gewährleistung einer sehr hohen Verfügbarkeit kommen verschieden Formen von Redundanzen zum Einsatz, wie z. B. RAIM („redundant array of independent memory“) [9].
Abbildung 2.1 zeigt eine typische Architektur eines Mainframes. Die Grundlage bildet das Betriebssystem IBM z/OS, welches in der aktuellen Version (Stand Januar 2013) ein 64bit Betriebssystem ist. Es erlaubt die Aufteilung eines Mainframes in bis zu 15 logische Partitionen (LPARs), in denen jeweils getrennte Betriebssysteme installiert werden können. z/OS bietet darüber hinaus mit z/VM einen Hypervisor zum Betrieb von virtuellen Maschinen. Damit können beispielsweise Linux- oder Windows-Server auf einem Mainframe konsolidiert werden. z/OS verfügt wie jedes moderne Betriebssystem über eine virtuelle Speicherverwaltung. Der Begriff „Region“ beschreibt einen virtuellen Speicherbereich mit einer Größe von 2–8 GB.
Festplattenspeicher werden im Umfeld von Mainframes als „direct access storage devices“, kurz DASD bezeichnet. Dateien werden „data sets“ genannt. Unter UNIX Betriebssystemen beispielsweise sind Dateien byte-orientiert organisiert, deren interne Struktur wird vom verwendenden Anwendungsprogramm verwaltet. z/OS data sets hingegen sind record-orientiert, das Betriebssystem verwaltet hier also die interne Aufteilung in einzelne Datensätze. Es gibt verschiedene Arten von data sets. Die häufigsten drei davon sind sequential data sets, partitioned data sets (PDS) und key sequential data sets (KSDS). Auf

sequential data sets kann nur sequenziell zugegriffen werden, d. h. um den n-ten Datensatz zu lesen, müssen zunächst die vorherigen (n-1) Sätze gelesen werden. PDS enthalten ein internes Verzeichniss (directory), welches auf einzelne Einträge (members) verweist. Diese wiederum enthalten eine sequentiell gespeicherte Menge an Datensätzen. Der Zugriff auf einzelne member kann durch das directory direkt erfolgen. Programme werden beispielsweise als PDS gespeichert. KSDS enthalten für jeden Datensatz zusätzliche Informationen (keys), über die direkt auf einzelne Sätze zugegriffen werden kann, ohne vorherige Sätze lesen zu müssen.
Zugriffe auf data sets erfolgen mittels sog. Zugriffsmethoden (access methods), welche die entsprechende API für den Zugriff festlegen. Die wichtigsten Zugriffsmethoden sind:

- QSAM (Queued Sequential Access Method)
  Sequentieller Zugriff auf Datensätze in der Reihenfolge ihres Einfügens. Datensätze werden vom System zu Blöcken zusammengefasst. Sätze werden für bessere Performance vor dem Zugriff in den Hauptspeicher geladen.
- BSAM (Basic Sequential Access Method)
  Sequentieller Zugriff auf Datensätze in der Reihenfolge ihres Einfügens. Datensätze werden von der Anwendung zu Blöcken zusammengefasst
- BDAM (Basic Direct Access Method)
  Datensatzreihenfolge vom Anwendungsprogramm festgelegt. Zugriff erfolgt über Adresse bzw. Suche nach Sätzen ggf. ab einer Startadresse
- BPAM (Basic Partitioned Access Method)
  Datensätze werden wie PDS organisiert.
- VSAM (Virtual Sequential Access Method)
  Erlaubt direkten Zugriff auf Datensätze mittels Index Key bzw. Adresse oder sequentiellen Zugriff.

Auf z/OS aufbauend ist häufig eine IBM DB2 Datenbank installiert. DB2 ist eine relationale Datenbank, welche mittels SQL abgefragt werden kann. Die verschiedenen Anwendungsprogramme können diese Datenbank zur Datenhaltung verwenden. Die Anwendungsprogramme sind meistens Cobol-Anwendungen, jedoch werden auch C/C++ Programme oder Java-Programme eingesetzt. Die Anwendungsprogramme beinhalten die Programmlogik, welche im Umfeld betrieblicher Informationssysteme Geschäftsprozesse abbildet. Ein solcher Geschäftsprozess kann beispielsweise die Buchung einer Bestellung oder eine Datenaufbereitung für Statistikzwecke sein.
Die Anwendungsprogramme können grundsätzlich in die folgenden zwei sich gegenseitig ausschließende Gruppen eingeteilt werden:

- Batch-Programme
- Online-Programme (Transaktionen)

Batch-Programme führen Stapelverarbeitungen durch und laufen einmalig manuell oder automatisch regelmäßig, z. B. jeden Tag um 21:00 Uhr. Lange dauernde Vorgänge werden meist als Batch implementiert und zu Zeiten geringerer Lasten auf dem Mainframe

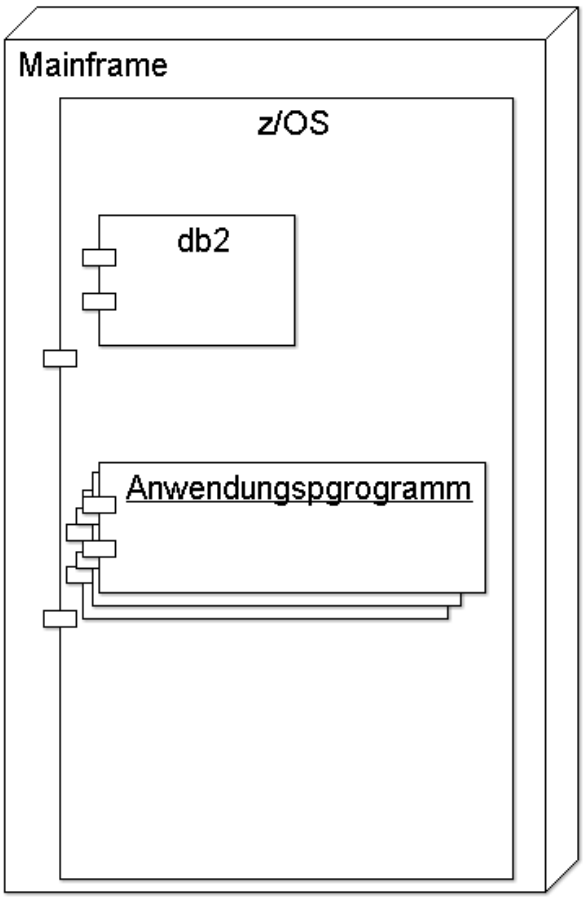


Abbildung 2.1.: Architekturbild eines Mainframe

(beispielweise Nachts) ausgeführt. Dieses Vorgehen eignet sich jedoch nur für Vorgänge, deren Ergebnis nicht sofort benötigt wird. Ein Beispiel hierfür wäre die Datenaufbereitung für statistische Reports. Da diese auf großen Teilen des Datenbestandes ausgeführt wird und entsprechend lange Laufzeiten hat, möchte man diesen Vorgang nicht während der Hauptlast durchführen. Zudem ist es hier ggf. ausreichend, die Reports mit einer täglichen Auflösung zu Beginn eines Tages vorliegen zu haben. Die Ausführung von Batch-Programmen wird durch das job entry subsystem (JES) kontrolliert. Dem JES werden auszuführende Jobs übergeben. Es terminiert dann die Ausführung und kontrolliert die von dem Job benötigten Ein-/Ausgabeströme. Um einen Batch-Job beim JES einzureichen, muss der geplante Job mittels der sog. JCL (job control language) spezifiziert werden. Mittels der JCL wird neben den auszuführenden Befehlen auch die Umgebung des Jobs (z. B. die verwendeten data sets) festgelegt.
Transaktionen hingegen führen interaktive Verarbeitungen durch, wie sie beispielsweise bei Buchungen benötigt werden. Hier wartet der Anwender unmittelbar auf ein Ergebnis der Verarbeitung (War die Buchung erfolgreich oder nicht?). Zur Bereitstellung von Web-Anwendungen oder Webservices auf dem Mainframe werden deshalb Transaktionen ver-

Eigenschaft	Online	Batch
Interaktion	ja	nein
Verarbeitete Datenmenge	einzelne Aufträge	Massenverarbeitung
Aufrufhäufigkeit	häufig	selten
Laufzeit	kurz	lang
Parallele Ausführungen	ja	nein
Steuerung durch	Transaktionsmanager	JES

Tabelle 2.1.: Typische Eigenschaften von Online- und Batch-Programmen

wendet. Der Begriff „Online“ (für Online-Programme bzw. Online-Verarbeitung) bezieht sich dabei jedoch nicht auf die Anbindung des Internets (z. B. bei Webservices), sondern dient der Abgrenzung zum Batch.
Transaktionen werden typischerweise von vielen Nutzern parallel und insgesamt in hoher Anzahl ausgeführt. Um diesen Anforderungen gerecht zu werden, laufen Transaktionen innerhalb von Transaktionsmanagern ab. Diese stellen Funktionen zum Scheduling, Dispatching, zur Zugriffskontrolle und zum Locking zur Verfügung, vergleichbar mit einem Application-Server in Java Enterprise Anwendungen. Die beiden am häufigsten eingesetzten Transaktionsmanager für z/OS sind IBM CICS und IBM IMS.
Die Tabelle 2.1 stellt die Eigenschaften einer typischen Transaktion sowie eines typischen Batch-Programmes gegenüber. Die Eigenschaft „Parallele Ausführungen“ meint hierbei die parallele Ausführung mehrerer Instanzen des Programms, nicht eine Parallelität in der Abarbeitung des Programmes selbst. Bis auf die letzte Eigenschaft („Steuerung durch“) sind diese Eigenschaften nicht vorgeschrieben, sondern lediglich charakteristisch für die meisten Programme in der jeweiligen Kategorie. So kann es z. B. Transaktionen geben, die selten aufgerufen werden oder Batch-Programme, die mehrfach parallel ausgeführt werden.
Anwendungen auf Mainframes bestehen aus mindstens einem Modul. Ein Modul ist ein ausführbares Programm. Jedes Modul selbst besteht wiederum aus mindestens einer *Control Section* (*CSECT*). Eine solche CSECT definiert einen im Speicher unabhängig positionierbaren Code-Block. Die Aufteilung eines Moduls in mehr als eine Control Section erlaubt es, ein Modul weiter zu strukturieren, ohne separate Module definieren zu müssen.
In Cobol-Code eingebettete SQL-Befehle werden vom Cobol-Compiler nicht direkt übersetzt. Statt dessen werden diese (für DB2) von einem DB2 Precompiler bzw. DB2 Coprocessor aus dem Cobol-Code entfernt und separat als ein sogenanntes *Database request module*, kurz *DBRM* gespeichert. Ein DBRM enthält das ursprüngliche SQL-Statement, allerdings nicht mehr als „Klartext“, sondern in einer speziellen internen Repräsentation. Im Cobol-Quellcode wird das SQL-Statement durch einen Aufruf entsprechender Datenbank-Schnittstellen ersetzt. Der somit modifizierte Cobol-Code kann anschließend als reguläres Cobol-Programm compiliert werden. Die DBRMs werden zu einem Plan (oft auch als Application Plan bezeichnet) oder zu Packages gebunden (*bind*). Ein Plan beinhaltet Informationen, die von der Datenbank benötigt werden, um tatsächlich auf die Daten zugreifen zu können. Ein Plan kann aus mehreren Packages bestehen. Der Vorteil davon, DBRMs zunächst zu Packages und diese dann zu einem Plan zu binden besteht in einer verbes-

serten Wartbarkeit. Wird ein DBRM geändert, muss dadurch lediglich das Package neu gebunden werden und nicht der komplette Plan.

2.2. Performance-Analyse

Die Performance-Analyse untersucht das zeitliche Verhalten eines Softwaresystems, d. h. wieviel Zeit für dessen Ausführung benötigt wird und wie sich diese Zeit auf die verschiedenen Programmbestandteile aufteilt. Es existieren zwei unterschiedliche Ansätze, um Informationen über das Zeitverhalte einer Software zu beschaffen:

- Statische Analyseverfahren
- Dynamische Analyseerfahren

2.2.1. Statische Analyseverfahren

Statische Verfahren untersuchen die zu prüfende Software, ohne diese auszuführen. Sie arbeiten in der Regel auf Basis des Sourcecodes oder Objektcodes, aber auch auf höheren Abstraktionsschichten, wie z.B. auf Modellen bei MDD Ansätzen. Statische Analyse-Werkzeuge arbeiten ähnlich zu einem Compiler, d. h. sie wandeln die Eingabe (das zu untersuchende Programm) in eine interne Repräsentation um und führen darauf Analysen durch. Ein großer Vorteil statischer Analyseverfahren ist, dass keine spezielle Umgebung notwendig ist, da die Analyse ohne Ausführung des Programms durchgeführt wird. Abhängigkeiten zu bestimmten Hardware-Plattformen, Betriebssystemen oder externen Programmen müssen zur statischen Analyse nicht erfüllt werden. Die Programme müssen zur Analyse auch nicht compiliert werden.
Da die analysierten Programme nicht ausgeführt werden, kann jedoch nicht ermittelt werden, wie oft Schleifen durchlaufen werden müssen oder welcher Ausführungspfad bei bedingten Anweisungen gewählt wird. Für eine effiziente Performance-Optimierung sind genau diese Informationen jedoch notwendig. Ausführungspfade oder Schleifenkörper, die nur selten oder überhaupt nicht ausgeführt werden, sind für eine Optimierung nicht lohnenswert, selbst wenn darin ineffiziente Code-Teile aufgerufen werden. Eine statische Analyse kann also keine Informationen über den tatsächlichen Verbrauch einer Anwendung unter realen Bedingungen liefern.

2.2.2. Dynamische Analyseverfahren

Dynamische Analyseverfahren berücksichtigen auch das dynamische Verhalten der untersuchten Programme. Die Programme werden dafür zur Ausführung gebracht und die notwendigen Informationen während der Ausführung ermittelt. Diese werden entweder von zusätzlichem Code, der mit dem zu untersuchenden Programm ausgeführt wird (Messcode) geliefert oder von externen Messprogrammen „beobachtet“ (Sampling). Der Messcode kann beispielsweise einen Zähler definieren und am Anfang des Rumpfes einer bestimmte Funktion diesen Zähler inkrementieren. Damit könnte die Anzahl der Aufrufe dieser Funktion gezählt werden. Um die Laufzeit einer Funktion zu ermitteln, könnte der

Messcode beim Ein- und Austritt in bzw. aus der jeweiligen Funktion die aktuelle Systemzeit ermitteln und die Ausführungszeit als Differenz daraus berechnen. Unter Instrumentierung versteht man die Einbringung von zusätzlichem Messcode für die dynamische Analyse, welche

- manuell durch den Programmierer
- durch den Compiler
- zur Programmausführung

erfolgen kann.
Eine manuelle Instrumentierung des Quellcodes eignet sich, wenn lediglich einzelne Abschnitte, beispielsweise eine einzelne Methode, gemessen werden sollen. Eine Messung größerer Programmabschnitte oder gar ganzer Programme sowie eine feingranulare Messung wird schnell sehr aufwändig, da viele solche Messbefehle eingefügt werden müssen. Zudem wird die Lesbarkeit des Codes durch die zusätzlichen Anweisungen stark beeinträchtigt. Aus diesem Grund werden die Instrumentierungen häufig automatisiert vom Compiler eingefügt. Der Quelltext bleibt dadurch unverändert und eine hohe Granularität kann einfach erreicht werden. Eine weitere Möglichkeit besteht darin, den Messcode erst zur Programmausführung einzufügen. Dies kann vor dem Laden des Programms in den Speicher oder bei Ausführung in einer virtuellen Maschine durch diese Maschine erfolgen.
Dynamische Analyseverfahren können ebenfalls danach eingeteilt werden, von welcher Ebene ausgehend die Instrumentierung erfolgt: [13]

- Source analysis
  Analyse ausgehend vom Source-Code. Abhängig von der verwendeten Programmiersprache, unabhängig von der Zielplattform (Architektur und Betriebssystem)
- Binary analysis
  Analyse ausgehend von Maschinen-Code. Unabhängig von der verwendeten Programmiersprache, abhängig von der Zielplattform

Werkzeuge, die solch dynamische Analysen durchführen nennt man „Profiler", deren Anwendung „Profiling". Ein Beispiel für einen Profiler, der dynamisch zur Laufzeit instrumentiert, ist Valgrind [13] [1].
Vorteil eines Profilings durch Instrumentierung ist dessen hohe Genauigkeit, da der eingebrachte Messcode bei Ausführung der instrumentierten Codestelle garantiert mit ausgeführt wird. Zudem lassen sich mittels Instrumentierung sehr detaillierte Profiling-Ergebnisse sammeln, wie z. B. Call-Stacks oder exakte Aufrufzahlen von Methoden. Bei der Instrumentierung großer Anwendungen ist zu beachten, dass auf Grund des hohen Detaillierungsgrades auch sehr große Informationsmengen gesammelt werden.
Ein Nachteil dieses Verfahrens ist, dass der für die Messung eingebrachte Code zusätzlich zum Code der gemessenen Anwendung ausgeführt wird und damit zusätzliche Ressourcen verbraucht. Dieser Zusatzverbrauch beeinflusst selbstverständlich auch die Messergebnisse. Zudem ist beim Profiling mittels manueller Instrumentierung und mittels Instrumentierung durch den Compiler ein erneutes compilieren vor der Messung notwendig.

[1] http://www.valgrind.org/, zuletzt aufgerufen am 27.02.2013

Diese beiden Nachteile stören besonders bei der Messung produktiver Anwendungen, da hierbei möglichst wenige zusätzliche Ressourcen für die Messung aufgewendet werden sollen. Ein erneutes Compiling (und damit Deployment) sollte ebenfalls vermieden werden. Zudem sollte der Messcode nach Abschluss der Messung wieder entfernt werden (um nicht dauerhaft den erhöhten Ressourcenbedarf zu haben), wozu die Anwendung ohne Instrumentierung ein weiteres Mal deployed werden muss.
Um diese Nachteile zu vermeiden kann das Profiling mit einem weiteren Verfahren durchgeführt werden. Dieses „Sampling" genannte Verfahren, beobachtet periodisch den aktuellen Ausführungszustand des zu messenden Programmes, nimmt also regelmäßig eine Art Stichprobe (Sample). Profiler, die dieses Verfahren verwenden benötigen deshalb keine Instrumentierung. Eine erneute Übersetzung des zu messenden Programmes entfällt daher vollständig, sondern das Programm kommt komplett unverändert zur Ausführung. Aus diesem Grund ist das Sampling-Verfahren besonders gut zur Messung von produktiven Anwendungen geeignet. Zudem ist der zusätzliche Aufwand für die Messung in der Regel geringer als beim Profiling mittels Instrumentierung.
Dadurch, dass beim Sampling nicht die Anwendung sich selbst misst (wie bei der Instrumentierung), sondern von extern beobachtet wird, können auch Aufrufe von Systemmodule und Bibliotheken mit gemessen werden.
Ein Nachteil dieses Verfahrens ist jedoch, dass jegliche Programmausführung, die zwischen zwei Samples geschieht, nicht erfasst werden kann. Wieviel Information dadurch „verpasst" wird, hängt von der Abtastrate (Sampling-Rate), also der Anzahl pro Zeiteinheit genommener Samples, ab. Die Genauigkeit lässt sich grundsätzlich mit einer höheren Abtastrate verbessern, wobei sich dadurch gleichzeitig auch der Messaufwand erhöht. Die Sampling-Rate stellt also einen Zielkonflikt zwischen Genauigkeit und Messaufwand dar. Typische Abtastraten liegen im Bereich von etwa 50 bis 200 Samples pro Sekunde.
Durch das Sampling wird lediglich der aktuelle Ausführungszustand und ggf. dessen direkter Aufrufer erfasst. Aus diesem Grund liefern Profiler, die ihre Informationen mittels Sampling sammeln, keine kompletten Call-Stacks, wie sie durch Instrumentierung ermittelt werden können, sondern höchstens eine Aufrufebene. Zur Optimierung wird häufig genau diese Information, also durch welche Folge von Methodenaufrufen eine bestimmte Methode erreicht wurde, benötigt. Bei der Verwendung von Profiling mittels Sampling muss versucht werden, diese Information aus anderen Quellen, z. B. aus einer statischen Analyse des Sourcecodes zu ermitteln. Die statische Analyse kann lediglich mögliche Ausführungspfade liefern, welche die untersuchte Methode beinhalten. Gibt es mehr als einen solchen Ausführungspfad, so ist diese Information nicht eindeutig.

3. Vorhandene Arbeiten

3.1. Profiling

3.1.1. Valgrind

Valgrind[1] ist ein Profiler, der mit dynamischer Instrumentierung arbeitet. Durch die Instrumentierung ist Valgrind auch in der Lage Call-Stacks zu liefern. Valgrind ist allerdings nicht dafür geeignet, Profiling auf Mainframes vorzunehmen.

3.1.2. Strobe

Strobe von Compuware[2] ist ein weit verbreitets Profiling Werkzeug für Anwendungen auf Mainframes. Es arbeitet mittels Sampling. Strobe bietet zudem die Möglichkeit, Informationen zu DB2-Datenbanknutzungen direkt aus DB2 abzufragen. So werden auch die SQL-Texte der ausgeführten Statements im Profiling-Report aufgeführt.

3.1.3. Application Performance Analyzer for z/OS

Der „Application Performance Analyzer for z/OS“ von IBM [3] arbeitet ähnlich wie Strobe. Er liefert nahezu identische Informationen. Die Ausgabe dieses Profilers kann auch als XML-Datei erfolgen, was ein Parsing erleichtert. „Application Performance Analyzer for z/OS“ ist allerdings nicht so weit verbreitet wie Strobe.

3.1.4. PathPoint

Die Software „PathPoint“[4] sammelt Informationen über die Ausführung einzelner Transaktionen auf Mainframes. Die gesammelten Informationen werden mit zusätzlichen Daten wie z. B. Zugriffspfaden für Ausführungen von SQL-Statements auf DB2 angereichert. Die Software sammelt auch Ein-/Ausgaben eines zur gemessenen Transaktion gehörenden Terminals.
Zur Auswertung der Daten wird ein Frontend geliefert, welches eine Navigation innerhalb der Performancedaten ermöglicht. Eine Suche nach Anti-Pattern ist jedoch nicht implementiert.

[1] http://www.valgrind.org/, zuletzt aufgerufen am 27.02.2013
[2] http://www.compuware.com, zuletzt aufgerufen am 26.11.2012
[3] http://www-01.ibm.com/software/awdtools/apa/, zuletzt aufgerufen am 12.02.2013
[4] http://www.pathpointsoftware.com/, zuletzt aufgerufen am 18.03.2013

3.2. Performance Anti-Pattern

Thomas Lamperstorfer beschreibt in seiner Master's Thesis [11] die Suche nach Performance Anti-Pattern mittels statischer Codeanalyse. Für die statische Analyse führt er ein „Performance Modell" genanntes Verfahren ein, mit dessen Hilfe er Kosten für einzelne Ausführungspfade abschätzt. Lamperstorfer behandelt insgesamt acht performancerelevante Anti-Pattern im Zusammenhang mit betrieblichen Informationssystemen und der Nutzung von SQL-Datenbanken. Er entwickelt ein Werkzeug für die statische Codeanalyse von Cobol-Programmen. In einer Fallstudie führt er mit diesem Ansatz schließlich eine Analyse eines solchen Systems durch.
Auch Kwiatkowski und Verhoef [10] verfolgen einen ähnlichen Ansatz. Sie suchen nach „teuren" SQL-Statements, die in Cobol-Programme eingebettet sind. Dazu suchen Sie syntaktische Anti-Pattern, die auf Performance-Probleme hindeuten, wie z. B. die Verwendung von *SELECT* * oder von Aggregatfunktionen. Sie führen die Suche in IMS-Transaktionen eines Unternehmens aus dem Finanzsektor durch, bestehend aus mehr als 23000 Cobol-Programmen. Basierend auf den Einsparungen, die an einem kleinen Teil dieses Portfolios realisiert wurden, stellen sie eine Abschätzung über das Einsparpotential auf, welches mit der Umsetzung der Optimierungsvorschläge in weiteren Modulen möglich wäre.
Connie Smith und Lloyd Williams haben in drei Artikeln [19, 17, 18] Performance Anti-Pattern gesammelt und beschrieben. Die Beschreibungen erfolgen mittels des Verhaltens eines Programmteils bzw. mehrerer Programmteile. Zudem schlagen sie Lösungsansätze vor, allerdings gehen sie nicht darauf ein, wie diese Anti-Pattern in Programmen konkret gesucht und erkannt werden können.
Parson und Murphy [15] hingegen schildern in Ihrer Arbeit den Entwurf eines Werkzeuges für eine Suche nach Anti-Pattern in komponentenbasierten Unternehmensanwendungen auf Basis von Java Enterprise Technologien. Sie stellen zudem auch fest, dass die Informationsmenge, welche eine Performance-Messung bei einer Unternehmensanwendung liefert, zur manuellen Interpretation deutlich zu groß ist und daher passende Darstellungsformen gefunden werden müssen.
Tony Shediak behandelt in seiner Arbeit [16] Performance Anti-Pattern in betrieblichen Informationssystemen auf Mainframes. Er beschreibt verschiedene Anti-Pattern und schlägt entsprechende Lösungen vor. Zudem hat er die Verbesserungen durch die Umsetzung der jeweiligen Lösungsvorschläge in Form von reduzierter Ausführungszeit bzw. eingesparter CPU-Nutzung gemessen.

Teil II.

Konzepte zur Performance-Analyse

4. Anforderungen

Ziel des zu entwickelnden Analyse-Werkzeuges ist es, einem Performance-Experten den Zugang zu den für ihn relevanten Informationen zu erleichtern. Aus Gesprächen mit Performance-Experten konnten die im folgenden beschriebenen Anforderungen extrahiert werden. Die Anforderungen lassen sich in Anforderungen zur Analyse und zur Usability unterteilen.

4.1. Anforderungen zur Analyse

Anf. 4.1 a) Identifikation der Top-Verbaucher
In der Optimierung von betrieblichen Anwendungen werden in der Regel zunächst diejenigen Programmteile untersucht, welche viel CPU-Zeit in Anspruch nehmen (bzw. lange Ausführungszeiten verursachen), da eine Optimierung dieser den meisten Nutzen für die Gesamtanwendung bringt. Selbst wenn beispielsweise ein Modul durch Optimierungsmaßnahmen 90% seines Verbrauchs einsparen kann, jedoch gemessen am Gesamtverbrauch der Anwendung nur 1% der CPU-Nutzung verursacht, erzeugt diese Optimierung lediglich eine Einsparung von 0,9% bezogen auf den Gesamtverbrauch des Programms.

Anf. 4.1 b) Identifikation der SQL-Statements mit hohem Verbrauch
SQL-Statements können grundsätzlich einen hohen Gesamtverbrauch durch einen hohen Verbrauch pro einzelnem Aufruf verursachen oder insgesamt auf Grund hoher Aufrufzahlen. Beide Fälle sind für Optimierungsmaßnahmen interessant und sollten von einem Analysewerkzeug aufgezeigt werden. Im Fall hoher Ausführungszahlen lässt sich ggf. eine Optimierung durch eine Reduktion der Aufrufe, beispeilsweise durch Zwischenspeicherung mehrfach verwendeter Daten, erreichen. Statements, die hingegen einen besonders hohen Verbrauch pro einzelner Ausführung verursachen, deuten auf eine unzureichende oder sogar fehlende Indexnutzung oder auf die Verwendung von aufwändigen SQL-Konstrukten (z. B. JOINs über viele Tabellen, UNIONs etc.) hin.

Anf. 4.1 c) Hinweis auf bereits bekannte Probleme
Bereits bekannte, wiederkehrende Probleme (Anti-Pattern) sollten von einem Analyse-Werkzeug automatisch erkannt werden können. Einige dieser bekannten Problem werden im Kapitel 6 erläutert.
Ein Werkzeug zur Unterstützung von Performance-Optimierungen soll ein gemessenes System auf die Präsenz aller hinterlegten Anti-Pattern untersuchen und somit dem Analyse-Experten ein Absuchen der gesamten Perfor-

mance-Daten nach dem Auftreten eines jeden einzelnen Anti-Pattern ersparen. Zudem soll dadurch verhindert werden, dass ein bereits bekanntes, allgemeines Problem in einer Messung enthalten ist, aber unerkannt bleibt. Dies könnte vorkommen auf Grund der hohen Datenmenge oder weil dieses spezielle Anti-Pattern dem nutzenden Optimierungsexperten bisher nicht bekannt war.

Anf. 4.1 d) Ermittlung der Quellen hohen Systemverbrauchs
Verbrauch durch Systemmodule wird von Anwendungsmodulen ausgelöst. Wird beispielsweise das Modul SMS (Storage Manager) ausgeführt, so liegt dessen Ursache in Zugriffen auf Daten, die vom Storage Manager verwaltet werden. Systemmodule selbst können jedoch nicht optimiert werden. Eine Verbrauchsreduktion muss also durch eine Optimierung des aufrufenden Moduls erreicht werden, z. B. durch Vermeidung der betreffenden Aufrufe oder effizienterer Nutzung des Systemmoduls. Aus diesem Grund ist es notwendig, dass ein Performance-Experte schnell die Aufrufer eines Systemmoduls identifizieren kann.

Anf. 4.1 e) Hinterlegung einer Knowledge-Base
Die Systemmodule bzw. deren Control Sections erfüllen stets die selben Funktionen, unabhängig von der jeweiligen Messung. Es bestehen lediglich Unterschiede im Detail auf Grund von verwendeten Versionen und Konfigurationsoptionen. Die Namen solcher Systembestandteile lassen in der Regel jedoch keine Rückschlüsse auf deren Funktion zu. Daher ist eine Verknüpfung mit einer (möglicherweise von Experten gesammlten) Beschreibungsdatenbank (Knowledge-Base) sehr hilfreich und erspart das Aufsuchen dieser Informationen in anderen Quellen.
Weiterhin kann eine solche Datenbank Hinweise auf mögliche Ursachen für hohen Verbrauch des jeweiligen Moduls beinhalten.

4.2. Anforderungen zur Usability

Anf. 4.2 a) Schneller Überblick über des gemessene System
Performance-Experten sollen für die Analyse in einem ersten Schritt einen schnellen Überblick über das gemessene System erlangen können. Daraus ermitteln sie zum Beispiel, ob das System aus Sicht der CPU-Nutzung sehr datenbanklastig ist, viel CPU-Zeit in bestimmten Systemroutinen verbringt oder einzelne Anwendungsmodule hohen CPU-Verbrauch aufweisen. Auf Basis der daraus ermittelten Informationen werden anschließend die betroffenen Programmteile genauer betrachtet.

Anf. 4.2 b) Anzeige von Detailinformationen
Detailinformationen zu den einzelnen Programmbestandteilen sollen direkt angezeigt werden. Dies sind z. B. Aufrufzahlen für SQL-Statements oder Informationen darüber, ob ein SQL-Statement statisch oder dynamisch ist.

Anf. 4.2 c) Einfache und schnelle Navigation
Das Analyse-Werkzeug soll eine möglichst einfache Navigation innerhalb der enthaltenen Informationen ermöglichen. Wichtig hierbei sind Drilldowns, z. B. von Modulebene herunter auf einzelne Control Sections innerhalb eines bestimmten Moduls, sowie Querverweise zwischen Aufrufern und aufgerufenen Programmteilen (Caller/Callee). Zudem sollten die betroffenen Programmteile einer Anti-Pattern Suche gem. Anf. 4.1 c) durch direktes Anklicken aufrufbar sein und nicht nach deren Namen gesucht werden müssen. Diese Tätigkeiten erfordert auf Basis der Textreports sehr hohen Aufwand.

Anf. 4.2 d) Verbindung zum Sourcecode
Der Sourcecode analysierter Module soll direkt in einem Editor aufgerufen werden können. Zudem sollen die einzelnen Intervalle einer Contol-Section bei Cobol mit Hilfe des Compile-Listings übersetzt werden. Dies ermöglicht einen direkten Aufruf betroffener Zeilen im Sourcecode.

Anf. 4.2 e) Einstellbarer Detaillierungsgrad
Die Verbrauchsinformationen sollen auf unterschiedlichen Ebenen analysiert werden können. So sollen von globaleren Sichten auf die gemessene Anwendung schrittweise detailliertere Betrachtungen abrufbar sein (sog. Drilldown). Beispielsweise soll von der Aufteilung des Verbrauchs eines Moduls in einzelnen Control Sections eine dieser Sections ausgewählt werden können, um dessen Verbrauch zu analysieren.

5. Analyse von Strobe Reports

Als Grundlage zur Identifikation von Optimierungspotentialen dienen die Messergebnisse einer Performance-Messung. Im Rahmen dieser Arbeit wird zur Performance-Messung das Werkzeug „Strobe“ verwendet. Die Ergebnisse der Messung eines Jobs mittels Strobe werden als Textdatei ausgegeben. Diese Datei wird von Compuware als „Strobe Performance Profile“ bezeichnet. Innerhalb dieser Arbeit wird synonym der Begriff „Strobe-Report“ verwendet. Der Umfang einer solchen Textdatei umfasst häufig mehr als 10000 Zeilen, was diese für eine manuelle Auswertung sehr unübersichtlich macht. Der Aufbau dieser Reports wird im folgenden Abschnitt 5.1 detailliert erläutert.

5.1. Aufbau von Strobe-Reports

Das Format der Strobe-Reports ist ursprünglich für eine Betrachtung am Mainframe-Terminal vorgesehen worden. Aus diesem Grund sind die Reports in Seiten unterteilt. Jede Seite beginnt dabei mit einem „Seiten-Header“. Bei einem Export der Strobe-Reports als Textdatei werden alle enthaltenen Seiten untereinander aufgelistet. Die Seiten-Header werden dabei mit übernommen.
Ein solcher Strobe-Report ist in verschiedene Abschnitte strukturiert, welche verschiedene Aspekte der Messung beschreiben. So gibt es beispielsweise Abschnitte über die Nutzung der CPU, über Wartezeiten, über die Ausführung von SQL-Statements und über Dateizugriffe. Nicht alle Abschnitte sind in jedem Report enthalten. Je nach gemessenem Programm sowie den zur Messung eingestellten Optionen werden zusätzliche Abschnitte aufgeführt. Abschnitte über SQL-Nutzung einer DB2-Datenbank werden z. B. nur dann gelistet, wenn das gemessene Programm überhaupt DB2 nutzt und das DB2-Modul von Strobe für diese Messung auch aktiviert wurde.
Die Abbildung 5.1 zeigt eine Auflistung der Abschnitte eines Strobe-Reports. Diese Auflistung stammt aus einem Inhaltsverzeichnis, wie es von manchen Strobe-Versionen zu Beginn eines Reports erzeugt wird und welches die darin tatsächlich enthaltenen Abschnitte auflistet. Die Kürzel auf der rechten Seite des Inhaltsverzeichnisses (z. B. *#MSU*) dienen dazu, einen bestimmten Abschnitt schneller mittels der Suchfunktion in einem Texteditor auffinden zu können. Der oben erwähnte Seiten-Header listet neben der Seitenzahl und der Überschrift des auf der Seite gelisteten Report-Abschnitts auch das entsprechende Abschnittskürzel (sofern der Report über ein Inhaltsverzeichnis verfügt). Eine Suche nach diesem Kürzel von Beginn des Reports an findet also nach der betreffenden Zeile des Inhaltsverzeichnisses den ersten Seiten-Header zum gesuchten Abschnitt.
Einige dieser Abschnitte beinhalten Zusammenfassungen von Informationen aus anderen Abschnitten um eine manuelle Auswertung zu erleichtern. Der Abschnitt *Most Intensively Executed Procedures* führt bespielsweise die 10 Prozeduren auf, welche am meisten CPU-Nutzung verursachen. Diese Informationen können allerdings auch aus dem Ab-

Abbildung 5.1.: Abschnitte in einem Strobe-Report

schnitt *Program Usage By Procedure* ermittelt werden, allerdings muss hierzu der gesamte Abschnitt nach den größten Verbrauchswerten durchsucht und anschließend absteigend sortiert werden. Weil diese zusammenfassenden Abschnitte keine zusätzlichen Informationen liefern, sondern lediglich bereits anderweitig vorhandene Informationen in einer anderen Ansicht zur Verfügung stellen, können sie bei einer automatischen Auswertung ausser Acht gelassen werden. Die zusammenfassende Ansicht kann aus den eigentlichen, detaillierten Daten bei Bedarf innerhalb eines Analysewerkzeugs erneut generiert werden. Im folgenden werden die für eine Analyse des CPU-Verbrauchs relevanten Abschnitte mehr im Detail beschieben:
Der Abschnitt *Measurement Session Data* beschreibt die Rahmenbedingungen der Messung. Dies beinhaltet u.a. den Namen des gemessenen Jobs, den Messzeitpunkt, Informationen über die Sampling-Rate (Siehe 2.2.2) und die Systemumgebung (Versionen des Betriebssystems und ggf. genutzter Subsysteme, wie beispielsweise DB2), sowie eingestellte Report-Parameter. Eine Besonderheit stellt der von Strobe für das gesamte Programm gelieferte Wert für „ServiceUnits“ dar, welcher laut der Strobe Dokumentation einen maschinenunabhängigen Wert für die CPU-Nutzung repräsentiert. Dieser Wert ist nicht zu verwechseln mit den von IBM zur Abrechnung der Lizenzkosten genutzten Einheiten, die ebenfalls „ServiceUnits“ genannt werden. IBM's ServiceUnits sind nicht maschinenunabhängig.
Dieser Abschnitt führt auch die während der gesamten Messung genutze CPU-Zeit auf. Diese wird nur hier absolut als Zeit angegeben. Alle weiteren Angaben über die CPU-Nutzung einzelner Programmteile werden von Strobe als prozentualer Anteil an der gesamten CPU-Nutzung angegeben, also beispielsweise Modul ABC1234 verursacht 26,31% der gesamten CPU-Nutzung.
Beim Abschnitt *Most Intensively Executed Procedures* handelt es sich, wie bereits beschrieben, um einen zusammenfassenden Abschnitt der zehn Prozeduren, die am meisten CPU-Nutzung verursachten.
Änlich dazu beschreibt der Abschnitt *Program Section Usage Summary* den CPU-Verbauch einzelner Control Sections. Auch hierbei handelt es sich um eine Zusammenfassung.

Die Detailinformationen zu den beiden zuvor erläuterten Abschnitten können im Abschnitt *Program Usage by Procedure* ermittelt werden. Hier wird die CPU-Nutzung in einzelnen Control Sections noch weiter unterteilt in hexadezimale Intervalle innerhalb des Codes. Liegt zu einem Cobol-Programm das Compile-Listing vor, so können mit dessen Hilfe diese Intervalladressen auf Abschnitte innerhalb des Source-Codes abgebildet werden.
Der Abschnitt *Attribution of CPU Execution Time* führt für System-Module die aufrufenden Module und Control Sections auf, sowie den Anteil (an der Gesamtnutzung) der dadurch verursachten CPU-Nutzung durch jeden solchen Aufrufer. Dadurch lassen sich die Ursachen für „teure“ System-Aufrufe ermitteln. Sind SQL-Statements aus DB2 Verursacher von Systemaufrufen, so werden diese ebenfalls hier aufgelistet, sofern das DB2-Modul von Strobe zur Messung aktiviert wurde.
Zur Betrachtung von SQL-Ausführungen in DB2 ist der Abschnitt *CPU Usage by SQL Statememt* relevant. Er listet den SQL-Text der verwendeten Statements (je nach Einstellung von Strobe z.T. nicht vollständig) sowie Informationen zum Statement-Typ (statisch / dynamisch bzw. Cursor / Non-Cursor). Vor allem aber werden für jedes SQL-Statement die Anzahl der Ausführungen einzelner Operationen (z. B. *OPEN* oder *UPDATE*) sowie deren anteilige CPU-Nutzung genannt. Der Abschnitt ist gruppiert nach den Namen der DBRMs, welche die jeweiligen Statements beinhalten. Die Abbildung 5.2 zeigt ein Beispiel für die Darstellung von SQL-Statements in einem Strobe-Report.

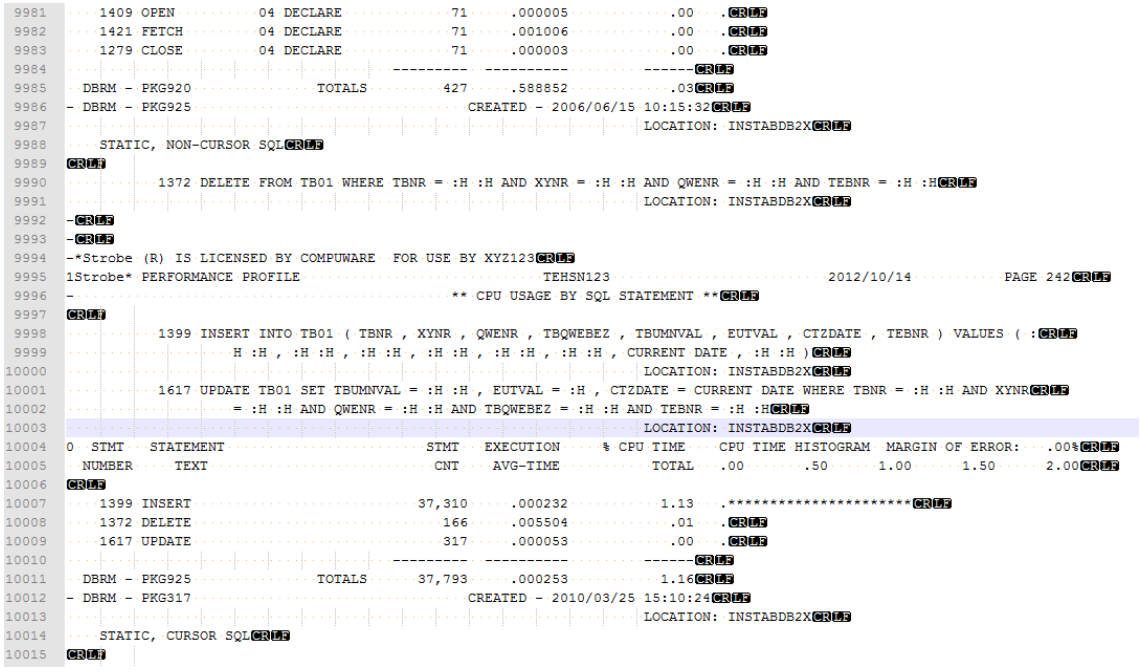


Abbildung 5.2.: CPU-Nutzung durch SQL-Statements im Strobe-Report

5.2. Workflow

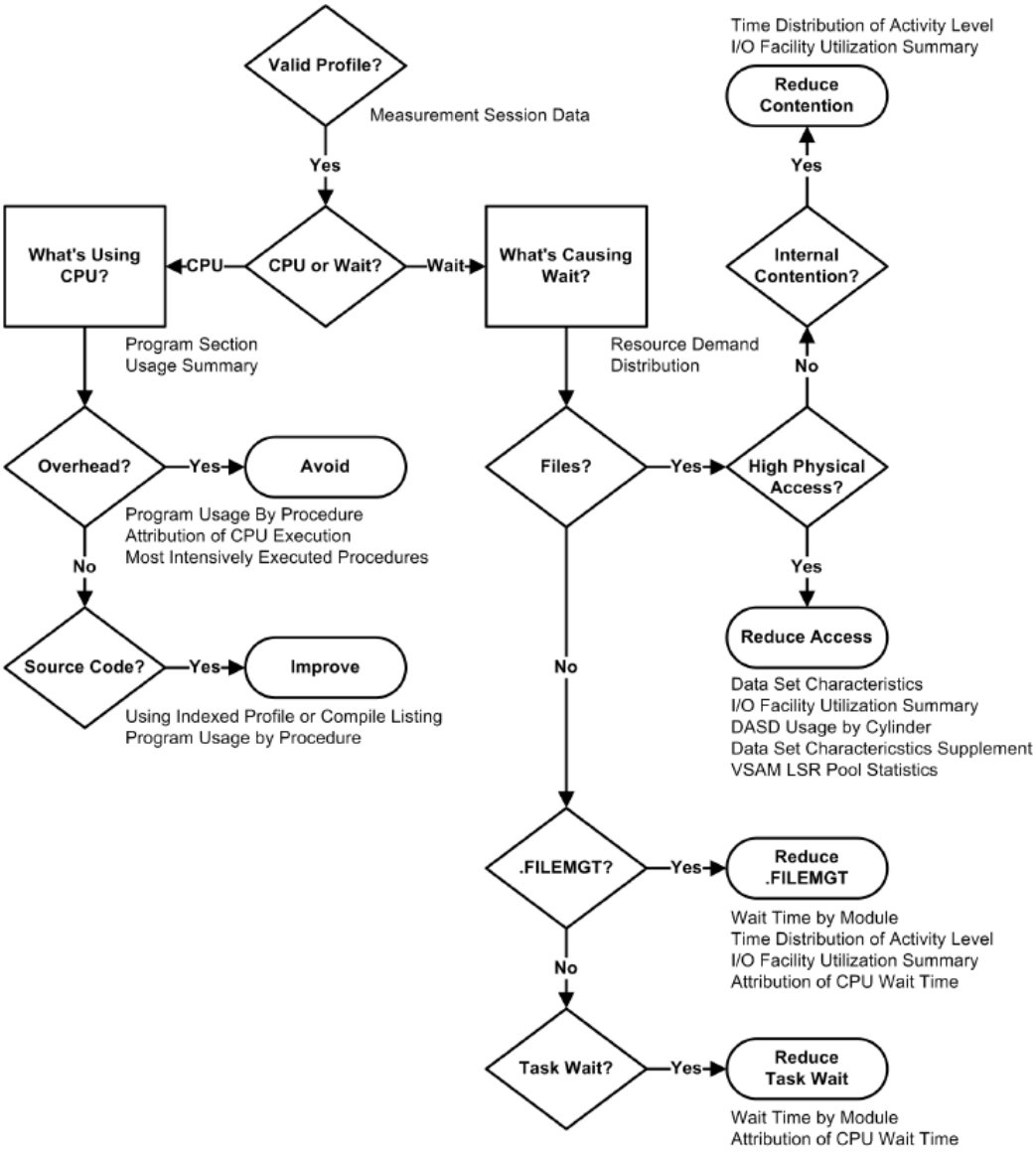


Abbildung 5.3.: Workflow zur Interpretation von Strobe-Reports

Compuware liefert mit seinem Profiling-Werkzeug Strobe ein Handbuch [5] aus, welches - zusätzlich zu Beschreibungen der einzelnen Felder innerhalb der generierten Reports - auch einen typischen Arbeitsablauf für eine Performance-Analyse vorstellt. Dieser behandelt einen „regulären“ Strobe-Report ohne zur Messung aktivierter Zusatzmodule, d. h. ohne Abschnitte zu Transaktionen, SQL Nutzung auf DB2 usw. Die Auswertung von Reports, welche solche Informationen zusätzlich enthalten, muss diese ebenfalls berücksichtigen, was den Arbeitsablauf entsprechend verändert. Der vorgestellte Ablauf soll im Fol-

genden an Hand des Flussdiagramms in Abbildung 5.3, welches ebenfalls dem erwähnten Handbuch entnommen ist, erläutert werden.
Entscheidungspunkte sind in diesem Diagramm durch Rauten dargestellt. Rechtecke hingegen beschreiben Fragestellungen, die der Optimierungsexperte beantworten soll, ohne dass in Abhängigkeit von deren Beantwortung eine Verzweigung erfolgt. Die Begriffe rechts neben den Rauten bzw. Rechtecken benennen die einzelnen Abschnitte des Strobe-Reports, in denen zur Beantwortung der jeweiligen Fragestellung relevante Informationen enthalten sind. Rechtecke mit abgerundeten Ecken beinhalten schließlich abstrakte Handlungsempfehlungen zur Verbesserung des jeweiligen Performance-Problems.
Der Ablauf schlägt vor, zunächst an Hand des Abschnitts *Measurement Session Data* zu prüfen, ob der analysierte Report eine gültige Messung beschreibt. Dies umfasst im Wesentlichen eine Prüfung danach, ob der Job, der gemessen wurde auch dem Job entspricht, der analysiert werden soll und ob die Messung eine ausreichende Genauigkeit aufweist. Die Messgenauigkeit wird von der Sampling-Rate bestimmt. Diese wird unter *Measurement Session Data* ausgewiesen, zusammen mit einer von Strobe berechneten Fehlertoleranz (Margin of Error). Zudem ist relevant, ob das gemessene Programm während der Messung beendet wurde (entweder ordnungsgemäß oder mit einem Fehler abgebrochen) oder ob die Messung nur einen Teil des Programmlaufs gemessen hat. Dies kann vorkommen, falls das Programm länger läuft, als Strobe Samples aufzeichnet. Diese Zeit lässt sich bei der Anfertigung der Messung indirekt über die Anzahl der aufgezeichneten Samples und die Sampling-Rate konfigurieren.
Liegt eine gültige Messung vor, so muss der Performance-Experte entscheiden, ob er CPU-Nutzung oder Wartezeiten optimieren möchte. Dazu liefert der Abschnitt *Measurement Session Data* Gesamtsummen der Zeiten, in der das gemessene Programm CPU-Nutzung verursachte und in der es warten musste. Diese Entscheidung ist nicht exklusiv, d. h. es können auch beide Fragestellungen untersucht werden. In der Regel jedoch lassen sich die gemessenen Programme in eine der beiden Gruppen (hauptsächlich CPU-Zeit oder hauptsächlich Wartezeit) einordnen.

5.2.1. Optimierung der CPU-Nutzung

Die wesentliche Fragestellung zur Optimierung der CPU-Nutzung ist, wodurch der hohe CPU-Bedarf verursacht wird. Dazu soll der Abschnitt *Program Section Usage Summary* untersucht werden. Da dieser Report die CPU-Nutzung jeder Control Section auflistet, kann daraus abgelesen werden, welche dieser Control Sections aussergewöhnlich viel CPU in Anspruch nimmt und daher detaillierter untersucht werden sollte.
Falls ein oder mehrere System-Module hohen CPU-Bedarf aufweisen, so spricht die Strobe-Dokumentation von „Overhead“. Der Abschnitt *Program Usage by Procedure* hilft dabei, den Verbrauch der betreffenden System-Module auf einzelne Control Sections herunterzubrechen und somit weiter einzugrenzen. Der Report *Attribution of CPU Execution Time* listet die Verteilung der CPU-Nutzung innerhalb von System-Routinen unter deren Aufrufern auf. Dadurch lassen sich die Aufrufer der System-Routinen mit hohem CPU-Bedarf ermitteln. Da sich die System-Routinen selbst in der Regel nicht optimieren lassen, muss untersucht werden, ob sich die Anwendungsmodule so verändern lassen, dass die betreffenden Systemaufrufe vermieden werden.
Sollte der Abschnitt *Program Section Usage Summary* Anwendungsmodulen die Verantwor-

tung für die starke CPU-Nutzung zuschreiben, so schränkt ebenfalls der Report *Program Usage by Procedure* die Quellen weiter ein bis auf Control Sections bzw. Source-Code Intervalle. Diese Intervalle können mit Hilfe des Compile-Listings direkt auf Abschnitte innerhalb des Source-Codes abgebildet werden. Für diese Code-Fragmente kann dann nach optimaleren Lösungen gesucht werden.
Anstatt die Abschnitte *Program Usage by Procedure* und *Attribution of CPU Execution* zu untersuchen, können die zehn Prozeduren mit dem höchsten CPU-Verbrauch auch im zusammenfassenden Abschnitt *Most Intensively Executed Procedures* aufgefunden werden.

5.2.2. Optimierung der Wartezeiten

Für eine Analyse der Ursachen von Wartezeiten sollte zunächst der Abschnitt *Resource Demand Distribution* untersucht werden. Dort lässt sich ermittlen, welche Tasks oder Dateizugriffe lange Wartezeiten verursachen. Wartezeiten, die keinem Dateizugriff direkt zugeordnet werden können, (z. B. das Einhängen eines Bandes in das Dateisystem) werden als „Dateimanagement“ (.FILEMGT) zusammengefasst. In diesem Report können ggf. mehrere unterschiedliche Verursacher identifiziert werden (z. B. ein Task und zwei Dateien). Für diese ist dann der folgende Pfad jeweils einzeln zu untersuchen.
Die folgenden drei Auswahlstellen im Ablaufdiagramm repräsentieren inhaltlich eine sich gegenseitig ausschließende Auswahl, ob es sich bei dem gerade untersuchten Verursacher entweder um Dateizugriffe, um Dateimanagement oder um einen Task handelt.
Im Fall von Wartezeiten, die von einem Task verursacht werden, kann im Abschnitt *Wait Time by Module* das Modul bzw. die Control Section festgestellt werden, welche für die Wartezeiten verantwortlich ist. Diese Programme können dann weiter untersucht werden, um Lösungen zu entwickeln, die weniger Wartezeiten verursachen. Sollte es sich um ein Systemmodul handeln, so können im Abschnitt *Attribution of CPU Wait Time* dessen Aufrufer nachgeschlagen werden. Die Aufrufer können dann dahingehend optimiert werden, dass die Systemkomponenten, welche Wartezeiten verursachen seltener bzw. optimaler genutzt werden. Systemkomponenten selbst lassen sich in der Regel nicht optimieren.
Wird die Wartezeit von Dateimanagement-Operationen verursacht, so können deren Verursacher ebenso in den Abschnitten *Wait Time by Module* und *Attribution of CPU Wait Time* ermittelt werden. Der Abschnitt *Time Distribution of Activity Level* zeigt an Hand von einem in der Dokumentation [5] beschriebenen Muster, wenn während der Messung auf Dateien eines Bandelaufwerkes zugegriffen wird, welche sich über mehr als ein Band erstrecken. Dann erkennt man in diesem Abschnitt in bestimmten zeitlichen Abständen Aktivitäten von .FILEMGT, die auf das Ein- bzw. Aushängen des Bandes im Dateisystem durch den Bandwechsel zurückzuführen sind. Der Abschnitt *I/O Facility Utilization Summary* liefert schließlich detaillierte Informationen darüber, auf welche Geräte zugegriffen wird, die Wartezeiten verursachen.
Kann die Wartezeit direkt einem Dateizugriff zugeordnet werden, so ist zu untersuchen, ob die lange Wartezeit durch exzessive physikalische Zugriffe oder durch konkurrierende Zugriffe verursacht wird. Sind häufige phyikalische Zugriffe die Ursache langer Wartezeit, so sollte versucht werden, die Zahl der Zugriffe zu verringern. Der Abschnitt *Data Set Characteristics* liefert Informationen über physikalische Eigenschaften der Speichermedien, auf die zugegriffen wird, wie z. B. Blockgrößen und Pufferoptionen. Diese Optionen können bei ungünstiger Parametrisierung erhöhte Wartezeiten durch erhöhte Zugriffs-

zahlen zur Folge haben. Für VSAM und QSAM Zugriffe werden im Abschnitt *Data Set Characteristics Supplement* zusätzliche Informationen über ausgeführte Operationen (z. B. Löschoperationen von Records) aufgeführt, mit deren Hilfe physikalische Zugriffe optimiert werden können. Zudem liefert der Abschnitt *VSAM LSR Pool Statistics* für VSAM Zugriffe zusätzliche Daten über die Puffernutzung. Auf weitere Optimierungen solcher Zugriffe geht die zu Grunde liegende Dokumentation jedoch nicht ein. Der Abschnitt *DASD Usage by Cylinder* wird in diesem Zusammenhang auch aufgeführt. Dort können die Dateizugriffe auf Zylinder-Ebene detailliert werden.
Compuware spricht bei konkurrierenden Zugriffen (Konflikten) von „inneren“ und „äußeren“ Konflikten. „Innere“ Konflikte („internal contention“) entstehen durch sich zeitlich überschneidende Zugriffe auf zwei (oder mehr) Dateien innerhalb des selben Speichermediums. Zu dessen Bestimmung lässt sich im Abschnitt *I/O Facility Utilization Summary* zunächst ermitteln, ob die gemessene Anwendung tatsächlich auf mehrere Dateien zugreift, die sich auf dem selben Speichermedium befinden. Ist dies der Fall, so kann im Abschnitt *Time Distribution of Activity Level* ermittelt werden, ob die Zugriffe auf diese Dateien sich zeitlich stark überschneiden. Existieren solche Dateizugriffe, so empfiehlt dieser Arbeitsablauf eine Aufteilung der betroffenen Dateien auf unterschiedliche Speichermedien, um parallele Zugriffe besser handhaben zu können. „Äußeren“ Konflikte („external contention“) hingegen entstehen durch sich zeitlich überschneidende Zugriffe auf Dateien innerhalb des selben Speichermediums durch mehrere Anwendungen. Da ein Strobe-Report lediglich eine bestimmte Anwendung misst, kann diese Konstellation nicht direkt im Report festgestellt werden. Er beinhaltet jedoch Hinweise auf ein Vorliegen von „äußeren“ Konflikten. Weist der Abschnitt *Resource Demand Distribution* für den betrachteten Dateizugriff eine deutlich höheren Wartezeit-Anteil (im Feld „Causing CPU Wait“) aus für die Durchführung von I/O Operationen (Feld „Serviced by I/O“), deutet dies darauf hin. Ein Performance-Experte sollte für diesen Fall untersuchen, ob gleichzeitig zur gemessenen Anwendung andere Anwendungen ausgeführt wurden, die auf die selbe Datei zugreifen.

5.3. Datenmodell

Zur Speicherung und weiteren Auswertung der aus einem Profiling-Report extrahierten Informationen wird ein Meta-Modell (Abb. 5.4) definiert. Dieses Datenmodell wird später sowohl dazu verwendet, optimierte Darstellungen für manuelle Interpretationen zu erzeugen, als auch dazu eine automatische Interpretation der darin enthaltenen Performance-Daten durchzuführen.
Kern dieses Modells ist die Klasse *ProgramPart*, welche die Gemeinsamkeiten der verschiedenen Bestandteile eines Programms repräsentiert (Konzept der Generalisierung). Für jede Instanz von *ProgramPart* bzw. dessen Subklassen wird die prozentuale CPU-Nutzung und die verursachte Wartezeit gespeichert; der Zugriff auf diese Informationen ist, durch die Vererbung bedingt, für alle Subklassen einheitlich.
Zur Ablage der hierarchischen Struktur eines Programms findet das Composite-Pattern nach Gamma et al. [8] Verwendung. Diese Struktur ermöglicht es, zur Auswertung der Daten über einen Drilldown den gewünschten Detaillierungsgrad einzustellen und somit die Menge der Informationen erfassbar darzustellen.
Die Klasse *Program* repräsentiert das gesamte gemessene Programm. Programme bestehen

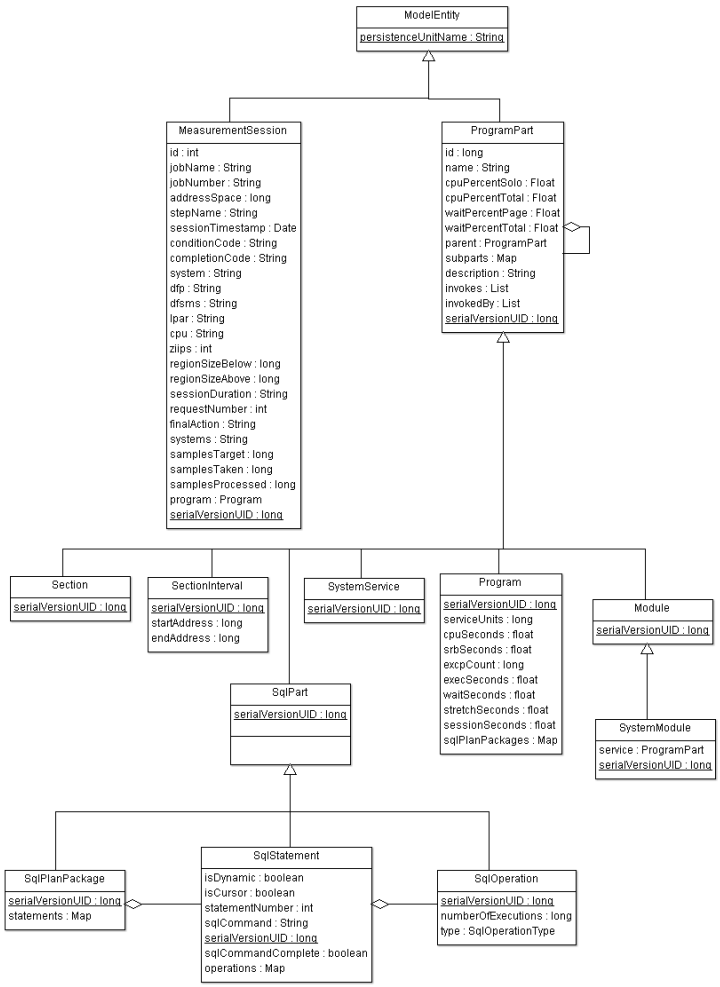


Abbildung 5.4.: Meta-Modell zur Speicherung von Performance-Daten

aus mindestens 1 Modul, jedes Modul wiederum kann verschiedene Sections (Control-Sections) beinhalten. Dieser Unterteilung spiegelt sich in den entsprechenden Klassen (*Module*, *Section*) im Meta-Modell wider. Das Sampling von Strobe liefert Ausführungszeiten noch detaillierter unterteilt nach Intervallen innerhalb einer Control Section. Ein solches Intervall ist durch die hexadezimale Anfangsadresse sowie duch dessen Länge bzw. Endadresse (welche sich gegenseitig ineinander überführen lassen) charakterisiert. Das Meta-Modell definiert zur Aufnahme dieser Informationen die Klasse *SectionInterval*. Diese verwendet die Endadresse anstatt der Länge des Intervals.
Eine Sonderform der Klasse *Module* stellt das *SystemModule* dar. Ein *SystemModule* ist einem bestimmten Systemdienst (*SystemService*) zugeordnet.
Für SQL-Ausführungen auf DB2 liefert Strobe den Namen des Plans (bzw. des Packages)

und die ausgeführten Statements mit deren SQL-Text. Je nach Einstellung der Messung ist der SQL-Text jedoch nicht vollständig dargestellt (Standard-Einstellung 300 Zeichen). Die Anzahl der Ausführungen wird separat für jede Operation eines Statements erfasst. Für die Modellierung der Operationen wird der Aufzählungsdatentyp *SqlOperationType* definiert (Abbildung 5.5).
SQL-Statements können statisch oder dynamisch sein. Statische Statements sind vorcom-

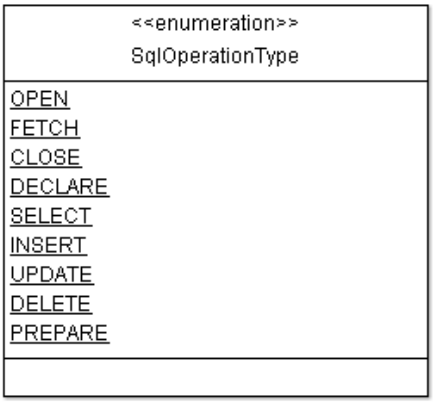


Abbildung 5.5.: Enum SqlOperationType

piliert. Der Zugriffspfad statischer SQL-Statements wird bereits bei der Erstellung (Binden zu einem Plan) ermittelt. Dynamische Statements hingegen werden erst zur Laufzeit interpretiert und optimiert. Diese Unterscheidung ist für Performance-Betrachtungen wichtig, da statische Statements bei Ausführung zwar den zusätzlichen Aufwand zur Interpretation des SQL-Textes und Erstellung des Zugriffspfads sparen, jedoch den Zugriffspfad nicht auf die zur Laufzeit aktuelle Datenverteilung und die dem Statement übergebenen Parameterausprägungen anpassen können.
SQL-Statements können zudem einen Cursor verwenden oder ohne Cursor ausgeführt werden. Diese Information wird ebenfalls im Meta-Modell erfasst.
Die Messung betreffende, allgemeine Informationen werden mittels der Klasse *MeasurementSession* verwaltet. Dazu zählen beispielsweise die Anzahl der genommenen Samples, die Sampling-Rate, die eingesetzten Systeme und deren Version usw.
Die Klasse *ModelEntity* generalisiert lediglich die Verwaltung der Subklassen wie z. B. deren Persistierung und trägt keine semantische Bedeutung. In der Regel beinhaltet ein Programm auch Programmteile, die andere Programmteile oder Systemmodule aufrufen. Zur Abbildung solcher Aufrufe steht die Klasse *Invocation* zur Verfügung. Diese „verknüpft“ zwei Instanzen von *ProgramPart* bzw. einer Subklasse dessen. Die Attribute *caller* und *callee* beschreiben eine bidirektionale Relation, deren anderes Ende das Attribut *invokes* bzw. *invokedBy* der Klasse *ProgramPart* beschreibt. Diese Art der Speicherung erlaubt es auf einfache Weise alle Aufrufe zu erhalten, die von einem bestimmten Programmteil ausgehen bzw. die Ursprünge aller Aufrufe eines bestimmten Programmteils zu ermitteln.
Die Abbildung 5.7 zeigt eine exemplarische Ausprägung des Datenmodells in Form eines UML Objektdiagramms. In dieser Darstellung wird die hierarschische Struktur deutlich,

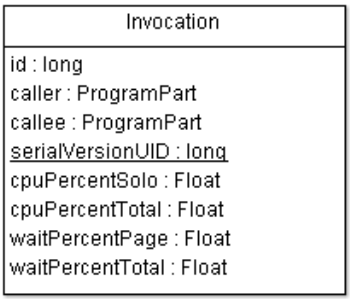


Abbildung 5.6.: Klasse *Invocation* zur Speicherung von Aufrufe

die im Klassendiagramm nicht unmittelbar sichtbar ist. Hierbei sind gegenseitige Aufrufe von Programmteilen (wie sie durch die Klasse *Invocation* repräsentiert werden) nicht dargestellt, da das Diagramm sonst zu unübersichtlich werden würde.
Um benötigte Informationen, wie z. B. den aggregierten Anteil der CPU-Nutzung aller Sql-Statements, aus einem solchen Baum zu extrahieren, kann dieser traversiert werden. Dazu bietet sich an, das Visitor-Pattern nach Gamma et al. [8] zu implementieren und beim Besuch der einzelnen Knoten die Informationen aufzusammeln.
Dieses Datenmodell zur Speicherung und Auswertung von Performance-Daten wurde bereits auf der BTW 2013 vorgestellt. [12]

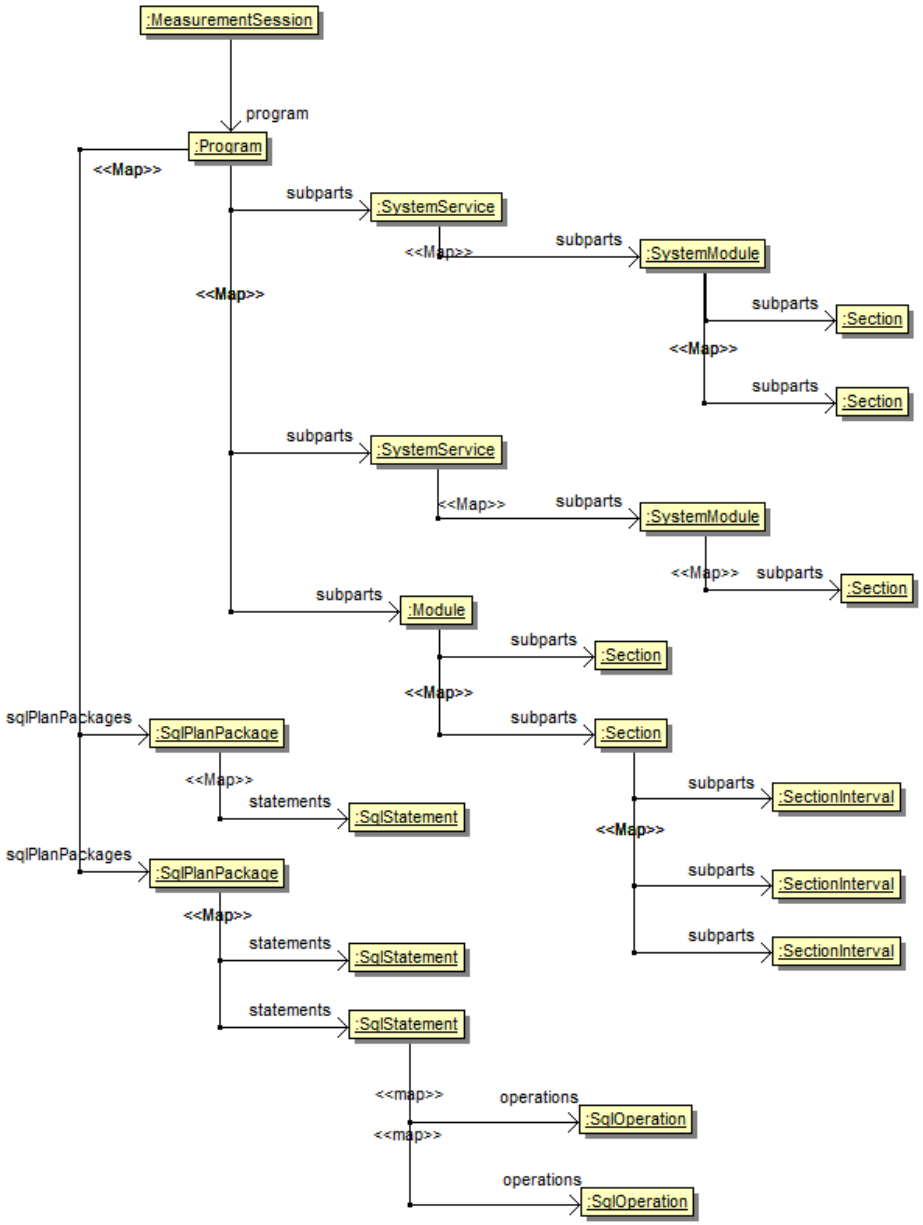


Abbildung 5.7.: Exemplarischer Baum mit Performance-Daten

6. Anti-Pattern

Ähnlich wie die sogenannte „Gang of Four“ in dem Buch „Design Pattern“ [8] allgemein bewährte Lösungen zu häufig auftretenden Problemstellungen in der Softwareentwicklung definiert, so beschreibt ein „Anti-Pattern“ nach Smith und Williams [19] eine häufig verwendete Lösung für ein gängiges Problem, dessen Anwendung negative Folgen mit sich bringt. Solche negativen Folgen können beispielsweise eine schlechte Wartbarkeit der Software, aber auch Performance-Probleme oder schlechte Skalierbarkeit sein. Viele der Anti-Pattern, die im Zusammenhang mit Performance-Probleme stehen, spiegeln sich auch Form von bestimmten Mustern in den Profiling-Reports wider. Diese können auch automatisch auf Basis der eingelesenen Daten gesucht werden. Einen vergleichbaren Ansatz auf Basis von JEE Anwendungen verfolgen auch Parson und Murphy [15].
Im Folgenden werden vier performance-relevante Anti-Pattern, wie sie häufig in Unternehmensanwendungen auftreten, beschrieben.

6.1. Einzelsatz-Fetch mit Cursor

Das Lexikon der Informatik [7] definiert einen Cursor wie folgt:

> „logischer Zeiger, der in typisch mengenorientierten Datenstrukturen, z. B. Relationen, die typisch satzorientierten Teilstrukturen, z. B. Tupel, diskret zugänglich macht“

Ein Cursor bei Datenbanken dient dazu, innerhalb der Ergebnismenge einer Abfrage zu navigieren und auf einzelne Elemente (oder ggf. auf mehrere Elemente als Block) zuzugreifen. Zum Zugriff bietet der Cursor eine *FETCH* genannte Operation, welche die Daten an der aktuellen Cursorposition liefert und diese Position auf das nächste Element (oder den nächsten Block) setzt. Der Vorteil der Nutzung eines Cursors zum Zugriff auf die Ergebnismenge liegt darin, dass diese nicht als Ganzes auf einmal in die Anwendung geladen werden muss, sondern lediglich kleinere Teilmengen davon (in der Regel einzelne Tupel, Cursor mit mehreren Tupeln pro Zugriff sind ebenso möglich; siehe hierzu auch Abschnitt 6.2). Dieses Vorgehen ist jedoch mit zusätzlichem Aufwand für die Erstellung und Verwaltung des Cursors verbunden.
Häufig werden dennoch auch bei Datenbankabfragen, welche die Rückgabe von höchstens einem Wert sicherstellen, Cursor eingesetzt. In diesem Fall könnte statt dessen der gesuchte Datensatz auch unmittelbar abgefragt und der Aufwand zur Erzeugung eines Cursors, Ausführung der Fetch-Operation und Freigabe des Cursors vermieden werden. Die Zusicherung, dass nicht mehr als ein Datensatz in der Ergebnismenge auftritt ist bei Abfragen auf Primärschlüssel oder bei Nutzung von Aggregatfunktionen ohne GROUP BY Klausel per Definition gewährleistet. Auf die Verwendung eines Cursors kann dann verzichtet

werden.
In den Performance-Daten erkennt man dieses Muster am Verhältnis von Open- zu Fetch-Ausführungen für ein SQL-Statement:

$$N_{Fetch} \leq N_{Open}, bzw. \frac{N_{Fetch}}{N_{Open}} \leq 1 \tag{6.1}$$

Die Formel 6.1 beschreibt eine notwendige, aber nicht hinreichende Bedingung für einen Einzelsatz-Fetch mit Cursor. Auch Abfragen, welche die Randbedingung höchstens einen Datensatz zu liefern nicht erfüllen, können diese Ungleichung dennoch erfüllen. Einerseits könnten viele Ausführungen keinen Datensatz zurückliefern, was in einem *OPEN* ohne *FETCH* resultiert. Andererseits könnte auch auf Grund der Struktur der abgefragen Daten im Mittel höchstens ein Datensatz pro *OPEN* selektiert werden, eine andere Datenbasis jedoch mehr als einen Satz für die betreffende Abfrage zulassen. Eine Entscheidung, ob diese Randbedingung erfüllt ist und damit dieses Anti-Pattern auf das betrachtete SQL-Statement zutrifft, kann nur mit einer weitere Untersuchung des SQL-Textes (Aggregatfunktion ohne GROUP BY) und ggf. der Datenbankstruktur (Selektion auf eindeutigen Schlüssel) getroffen werden.
Ein weiterer Grund der dazu führen kann, dass ein Statement irrtümlich als Implementierung dieses Anti-Patterns eingeordnet wird, ist ein „Multi-Fetch“. Dabei werden mehrere Datensätze in einer einzigen *FETCH* Operation aus der Datenbank geladen. (Siehe hierzu auch Kapitel 6.2)
Ob ein SQL-Statement, welches einen Cursor verwendet, dieses Anti-Pattern erfüllt, kann wie bereits beschrieben nicht eindeutig entschieden werden. Es kann jedoch ermittelt werden, ob ein SQL-Statement ein möglicher Kandidaten für dieses Anti-Pattern ist oder ob dessen Vorliegen ausgeschlossen werden kann. Dazu wird für dieses Statement das oben beschriebene Verhältnis berechnet. Liegt es zwischen einem konfigurierbaren Schwellwert und 1, so wird das betrachtete Statement als Kandidat für dieses Anti-Pattern betrachtet. Der Schwellwert dient dazu, ggf. auftretende Abfragen, die keine Records liefern, ebenfalls zu akzeptieren (z. B. wenn nach einem eindeutigen Schlüssel gesucht wird, der gesuchte Wert jedoch in der Tabelle nicht exisitert). Wird *FETCH* jedoch häufiger ausgeführt als *OPEN*, muss mindestens eine Ausführung des Statements eine Ergebnismenge mit mehr als einem Datensatz enthalten (Dirichlet'sches Schubfachprinzip) und dieses Anti-Pattern kann sicher nicht vorliegen.

6.2. Einzelsatz-Fetch für Massenoperationen

In betrieblichen Informationssystemen werden häufig Datenbankabfragen ausgeführt, deren Ergebnismengen sehr viele Datensätze enthalten, die zur weiteren Verarbeitung abgerufen werden müssen. Dabei ist es nicht optimal, jeden Datensatz einzeln oder jeweils nur kleine Mengen an Datensätzen mit einem *FETCH* Aufruf von einem Cursor in eine lokale Datenstruktur zu laden. Effizienter hingegen ist es, je nach der erwarteten Größe der Ergebnismenge größere Partitionen davon (z. B. 100 Datensätze) auf einmal in die Anwendung zu laden. Dieses Anti-Pattern ist eine Variante des „Empty semi-trucks“-Anti-Pattern von Smith und Williams [18].
Jeder Aufruf von *FETCH* erzeugt jeweils einen Kontextwechsel vom Anwendungspro-

gramm in die Datenbank und zurück von der Datenbank in die Anwendung. Der dadurch verursachte zusätzliche CPU-Aufwand kann durch eine Reduktion der Anzahl an *FETCH* Aufrufen in Summe verringert werden. Falls die genutzte Datenbank über ein Netzwerk angesprochen wird, führt diese Reduktion zugleich auch zu verringerten Wartezeiten, da für jeden einzelnen Aufruf eine Netzwerkkommunikation erfolgen muss, welche mindestens eine Round-Trip-Time benötigt.
Die Formel 6.2 beschreibt den Zusammenhang zwischen der Anzahl an *FETCH*- Operationen N_{Fetch}, der Gesamtzahl der von der Abfrage selektierten Datensätze $N_{Records}$, sowie der Anzahl der pro *FETCH* geholten Sätze N_{Cursor}.

$$N_{Records} + C = N_{Fetch} * N_{Cursor} \tag{6.2}$$

Dabei beschreibt C die Anzahl der Cursorelemente, die beim letzten *FETCH*-Aufruf leer bleiben. C ergibt sich entsprechend 6.2 aus $N_{Records} \mod N_{Cursor}$. Vernachlässigt man C, so ist N_{Fetch} indirekt proportional zu N_{Cursor}.
Die Formel 6.3 beschreibt den Zusammenhang zwischen der Anzahl an *FETCH* und *OPEN* Operationen beim Vorliegen dieses Anti-Patterns.

$$\frac{N_{Fetch}}{N_{Open}} \geq Y, Y \gg 1 \tag{6.3}$$

Aus 6.2 und 6.3 erkennt man, dass sich eine hohe Anzahl an Aufrufen der *FETCH* Operation durch eine Vergrößerung von N_{Cursor} verringern lässt. Als Nachteil dieses Verfahrens sei erwähnt, dass die Anwendung dafür mehr Arbeitsspeicher benötigt, um die größere Menge an Datensätzen aufzunehmen. Auf modernen Mainframes stellt der Arbeitsspeicher jedoch meist nicht einen so stark limitierenden Faktor dar, als dass eine solche Optimierung nicht realistisch durchführbar ist. Betrachtet man eine Abfrage, deren Ergebnisdatensätze eine Länge von 200 B haben, so wird für eine Erhöhung von $N_{Cursor} = 1$ auf $N_{Cursor} = 100$ lediglich zusätzlicher Speicher in folgender Größe benötigt: $(100 - 1) * 200B = 19800B \approx 20KiB$
Die Anzahl an Kontextwechseln für den Abruf von Datensätzen vom Cursor verringert sich dadurch aber auf $\frac{1}{100}$ der für $N_{Cursor} = 1$ notwendigen Anzahl.
Auch Cobol bietet entsprechende Möglichkeiten um mehrere Datensätze auf einmal von einem Cursor zu laden. Dazu müssen statt einzelner Variablen, in die dann die Datenfelder vom Cursor gespeichert werden, Arrays mit N_{Cursor} Elementen definiert werden. Zum Abrufen der Datensätze wird anschließend eine erweiterte Syntax verwendet:

```
EXEC SQL
  FETCH NEXT ROWSET FROM TAB1
  FOR N ROWS
  INTO :FLD1, :FLD2, :FLD3
END-EXEC.
```

Dabei ist für N der Wert von N_{Cursor} einzusetzen, also z. B. *... FOR 100 ROWS ...*. Die Anzahl der tatsächlich geholten Datensätze wird über *SQLERRD(3)* der Anwendung zur Verfügung gestellt. Dies ist notwendig, da der letzten Aufruf von *FETCH* lediglich $(N_{Records} \mod N_{Cursor})$ Datensätze liefert und dies bei der nachfolgenden Bearbeitung der Arrays für die Datenfelder berücksichtigt werden muss.

6.3. Schnitzeljagd, Join-zu-Fuß

Die „Schnitzeljagd“ wird von Smith und Williams [19] auch als „Circuitous Treasure Hunt“ bezeichnet. Der Name lehnt sich an das Kinderspiel an, bei dem eine Art Schatz gesucht wird, indem ausgehend von einem Starthinweis jeweils der nächste Hinweis gesucht wird, bis letztendlich der eingentlich gesuchte „Schatz“ gefunden wird.
Häufig werden in betrieblichen Informationssystemen auf ähnliche Art und Weise Informationen aus einer Datenbank ermittelt. Ausgehend von einem bestimmten Datum werden zugehörige Informationen aus der Datenbank abgefragt. Mit Hilfe dieser Abfrageergebnisse und ggf. einer weiteren Aufbereitung, wird dann die nächste Abfrage aufgebaut und abgesetzt. Dieser Vorgang wird u.U. mehrfach wiederholt, bis die letztendlich zur weiteren Verarbeitung notwendigen Informationen ermittelt wurden. Stattdessen ließe sich häufig mittels einer einzigen - dann jedoch komplexeren - Abfrage die gesuchte Information direkt aus der Datenbank ermitteln, ohne die Zwischenschritte im Anwendungsprogramm auszuführen. Da die Datenbank Optimierungen der Zugriffspfade vornehmen kann, welche dem Anwendungsprogramm nicht möglich sind, ist dieser Zugriff trotz erhöhter Abfragekomplexität in der Regel performanter und nimmt weniger CPU-Rechenzeit in Anspruch. Zudem entfällt der zusätliche Aufwand für den mehrfachen Datentransfer aus der Datenbank in die Anwendung.
Ein allgemeinerer Fall davon ist der Join-zu-Fuß, bei dem im Gegensatz zur Schnitzeljagd auch Daten aus den Zwischenschritten benötigt werden. Dabei wird eine SQL *JOIN*-Operation anstatt sie von der Datenbank ausführen zu lassen, „zu Fuß“ im Anwendungsprogramm nachgebildet. Anstatt beispielsweise

```
select ... from A join B on A.col1 = B.col1 where A.col2 = ...
```

auszuführen, werden mit einer Abfrage wie

```
select A.col1, ... from A where A.col2 = ...

FOR EACH QUALIFIED RECORD rec
{
        select ... from B where B.col1 = rec.col1
}
```

zunächst die relevanten Tupel aus A abgefragt und für jedes Ergebnis aus A die zugehörigen Ergebnisse aus B ermittelt. In einer Schleife wird anschließend über die qualifizierten Records iteriert (im Beispiel wird der aktuelle Wert als „rec“ bezeichnet), und mit einer separaten Abfrage diejenigen Records aus B ermittelt, bei denen die Join-Spalte den Wert der Join-Spalte des aktuellen Records aus A enthält (im Beispiel rec.col1). Werden weitere Tabellen benötigt, werden diese mit weiteren Schleifen innerhalb der bestehenden Schleifen verschachtelt.
Diese Anti-Pattern sind nicht nur im Zusammenhang mit *SELECT*-Statements zu finden, sondern auch mit *INSERT*, *UPDATE*, *DELETE*. Dies tritt auf, wenn bespielsweise alle Kunden ermittelt werden, die eine bestimmte Bedingung erfüllen und in einer Schleife für jeden dieser Kunden per *UPDATE* ein Status gesetzt wird. Statt dessen, könnte die Qualifizierungsbedingung der Kunden-Relation direkt in einem *UPDATE*-Statement verwendet werden.

Die zu einer Schnitzeljagd oder einem Join-zu-Fuss gehörigen Statements zeichnen sich durch gleiche Ausführungszahlen aus. Bei Cursor-Statements ist auch die Anzahl der *FETCH*-Operationen relevant, da diese der Anzahl der Durchläufe der äusseren Schleife entspricht. Aus diesem Grund werden auch Statements, deren Ausführungszahl identisch zur Anzahl an *FETCH*-Operationen eines anderen Statements ist, als zu diesem gehörig betrachtet. In der Regel sind die zusammengehörigen Statements im selben DBRM enthalten. Die Suche nach identisch oft ausgeführten Statements kann mit dieser Information auf jeweils ein DBRM beschränkt werden.
Problematisch sind Statements mit sehr niedrigen Ausführungszahlen. Betrachtet man beispielhafte Statements A und B, die jeweils drei Ausführungen aufweisen. Diese Statements werden gemäß obiger Definition als Kandidaten für diese beiden Anti-Pattern eingestuft. Allerdings ist es sehr viel wahrscheinlicher, dass A und B fachlich unabhängig sind (und damit auch nicht sinnvoll per JOIN abgefragt werden können), als 2 Statements, die z. B. 13487 Ausführungen aufweisen. Als extremes Beispiel stelle man sich eine Anwendung vor, die zu Beginn ihrer Ausführung zum Caching eine Datenstruktur aus einer Datenbank aufbaut. Dazu wird für jede in diesem Cache zu speichernde Tabelle eine Datenbankabfrage ausgelöst. Die Tabellen können jedoch fachlich unterschiedliche Entitäten selektieren (z. B. Bundesländer und Bankleitzahlen) und können somit nicht per JOIN verknüpft werden, haben jedoch beide exakt 1 Ausführung. Aus diesem Grund sollten Statements mit Ausführungszahlen unterhalb eines bestimmten Schwellwertes von der Suche nach diesem Anti-Pattern ausgeschlossen werden.

6.4. Cobol Library

Es gibt Konstrukte in Cobol, die den Compiler dazu veranlassen, Routinen aus der Cobol Library aufzurufen, anstatt direkt Maschinenbefehle zur erzeugen. Diese Tatsache ist für den Cobol-Programmierer nicht direkt sichtbar - bei manchen solchen Anweisungen wird vom Compiler eine Warnung erzeugt, andere hingegen sind vollig transparent. Die Aufrufe dieser Bibliotheksroutinen finden sich natürlich in den Strobe-Reports wieder. Kann einem solchen Aufruf eine hohe CPU-Nutzung zugeordnet werden, bedeutet dies eine potentielle Optimierungsmöglichkeit durch Ersetzung des verursachenden Aufrufs gegen eine effizientere Implementierung.
Ein Beispiel hierfür ist der „variable length move“. Diese Bibliotheksroutine wird vom

```
 MODULE   SECTION        FUNCTION          INTERVAL    % CPU TIME
   NAME     NAME                            LENGTH   SOLO   TOTAL

....
 IGZCPAC  IGZCXDI  DOUBLE PRECISION DIVIDE     1161  65.13   65.13
....
 IGZCPAC  IGZCXMU  DOUBLE PRECIS. MULTIPLY      993   2.26    2.26
....
```

Abbildung 6.1.: Intensive Nutzung der Cobol Library im Strobe-Report

Compiler verwendet, wenn eine überlappende Struktur kopiert wird, sich also die Spei-

cherbereiche von Quell- und Zieloperand einer Zuweisung überschneiden. Im Quellcode sieht dieser Aufruf sieht nach einem regulären *MOVE*-Befehl (Zuweisung in Cobol) aus. Der exakte Unterschied wird erst bei genauer Betrachtung der verwendeten Argumente deutlich. Dieser Fall erzeugt eine Compile-Warnung, geschieht also für den Programmierer nicht völlig unbemerkt.
Das folgende Beispiel verdeutlicht die zu Grunde liegende Situation:

```
WORKING-STORAGE SECTION.
...
01          W-ABCD-INDEX.
   05       W-ABCD-INIT.
      10    FILLER        COMP-3  PIC S9(9)V9(6) VALUE 0.
      10    FILLER        COMP-3  PIC S9(5)      VALUE 0.
   05       W-ABCD-TAB.
      10    W-MAILA               OCCURS 999.
         15 W-MAILA-XYZZ          PIC S9(9)V9(6) COMP-3.
         15 W-MAILA-QWERT         PIC S9(5) COMP-3.
...
MOVE W-ABCD-INDEX        TO W-ABCD-TAB.
```

Die hier aufgeführte Zuweisung (*MOVE*) arbeitet mit zwei Operanden, deren Speicher sich überlappt, da *W-ABCD-TAB* ein Teil der Struktur *W-ABCD-INDEX* ist. Der Compiler erzeugt die folgende Warnung:

```
Data items "W-ABCD-INDEX (GROUP)" and "W-ABCD-TAB (GROUP)"
had overlapping storage. An overlapping move will
occur at execution time.
```

Abgesehen von dieser Warnung sieht die Zuweisung selbst nach einer regulären Zuweisung aus und lässt nicht vermuten, dass hierzu der Aufruf einer CPU-intensiven Bibliotheksroutine benötigt wird.
Andere Anweisungen, die den Compiler veranlassen solche Bibliotheksaufrufe zu erzeugen, erfolgen hingegen komplett ohne Warnung. Ein Beispiel hierfür ist „Inspect“ zur Bearbeitung von Strings. Tony Shediak [16] hat diesen Fall bereits untersucht und festgestellt, dass das gleiche Ergebnis mittels einer einfachen Schleife und Substring-Operationen deutlich performanter erzielt werden kann.
Auch für Fliesskommaoperationen mit doppelter Genauigkeit erzeugt der Cobol Compiler Bibliotheksaufrufe, deren Präsenz sich auf Quellcode-Ebene nur durch eine Betrachtung der Datentypen erschließt. Auf Mainframes werden typischerweise Unternehmensanwendungen ausgeführt, innerhalb derer die Verwendung doppelt genauer Fliesskommaoperationen höchst wahrscheinlich nicht notwendig und lediglich in einer unangemessenen Variablendefinition begründet ist. Durch eine Änderung der Datentypen lassen sich diese Aufrufe mit geringem Aufwand eliminieren.
Abbildung 6.1 stellt einen Auszug aus einem Strobe-Report eines produktiven Systems dar. Gemessen wurde ein Batch, der 67,39 % CPU-Nutzung alleine für Divisionen und Multiplikationen mit doppelter Genauigkeit verursacht.

Teil III.

Werkzeug zur Performance-Analyse

7. Parser

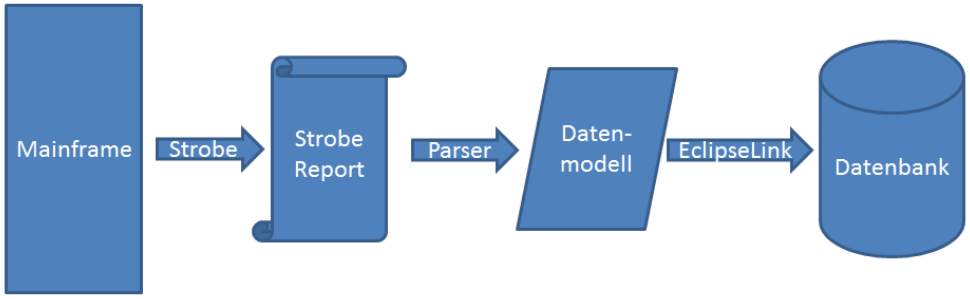


Abbildung 7.1.: Überführung von Performance-Daten in relationale Datenbank

Auf dem Mainframe wird von Strobe die gewählte Anwendung gemessen und der textbasierte Strobe-Report erzeugt. Um die darin enthaltenen Informationen zu extrahieren und in das im Abschnitt 5.3 beschriebene Datenmodell zu überführen, wird ein Parser benötigt. Das Datenmodell wird schließlich unter Verwendung eines O-R-Mappers in eine relationale Datenbank geschrieben. Hierfür wird die Java Persistence API (JPA) bzw. dessen konkrete Implementierung EclipseLink[1] verwendet. Ein Überblick über diesen Prozess ist in Abbildung 7.1 dargestellt.
Die Formatierung der Reports ist wie bereits beschrieben auf eine Darstellung am Bildschirm und eine manuelle Auswertung ausgerichtet. Eine formelle Beschreibung dieser Formatierung mittels kontextfreier Grammatiken ist nicht möglich, besonders auf Grund der häufig verwendeten Tabellendarstellungen. Eine Formulierung mit kontextsensitiven Grammatiken ist extrem komplex. Der im Rahmen dieser Arbeit entworfene Parser verwendet statt dessen einzelne reguläre Ausdrücke innerhalb eines Java-Programms und verwaltet bei Notwendigkeit eigene Statusvariablen zur Abbildung von Kontextinformationen.
Das im Abschnitt 5.3 vorgestellte Datenmodell wird in einem eigenen Java Package umgesetzt (*de.tum.in.laner.model*), welches sowohl vom Parser, als auch von dem im nächsten Kapitel 8 beschriebenen Analyse-Werkzeug gemeinsam eingesetzt wird.
Der Parser selbst ist in zwei Java Packages aufgeteilt:

- *de.tum.in.laner.parser*
  Konzeptioneller Teil des Parsers, unabhängig vom konkreten Report-Format
- *de.tum.in.laner.parserImpl*
  Konkrete Implementierung für Strobe-Reports

[1] http://www.eclipse.org/eclipselink/, zuletzt aufgerufen am 14.02.2013

Diese Aufteilung ermöglicht eine Erweiterbarkeit für die Erstellung weiterer Parser zum Einlesen von Reports in anderen Formaten (z. B. von anderen Profilern wie IBM's APA). Das Package *de.tum.in.laner.parser* beinhaltet die abstrakte Klasse *AbstractParser*, welche allgemeine Parserfunktionalitäten zur Verfügung stellt, wie z. B. das Einlesen einer übergebenen Textdatei und deren Aufteilung in einzelne *Sections*, also einzelne Abschnitte. Die konkreten Implementierungen des Parsers erben dann alle von dieser Klasse. Die abstrakte Klasse *SectionParserFactory* sieht eine Methode zur Erzeugung eines solchen section-spezifischen Parsers an Hand des Namens einer Section vor. Deren konkrete Implementierung für Strobe (*SectionParserFactoryStrobe*) sucht per Reflection nach einer entsprechend dem Section-Namen benannte Klasse innerhalb des für den Strobe-Parser vorgesehenen Paketes *de.tum.in.laner.parserImpl* und instanziert diese (sofern eine passende Klasse gefunden werden konnte).
Die Reports enthalten wie bereits erwähnt häufig tabellenartige Darstellungen. Eine solche Tabelle ist in Abbildung 7.2 dargestellt. Die ersten beiden Zeilen sind im originalen Strobe-Report nicht vorhanden, sondern wurden zur Visualisierung des Spaltenindex nachträglich manuell eingefügt. Diese Tabellendarstellungen erleichtern zwar eine manu-

```
0000000000111111111122222222223333333333444444444455555555556666666666777777777
0123456789012345678901234567890123456789012345678901234567890123456789012345678
 MODULE  SECTION   FUNCTION                  % CPU TIME  MARGIN OF ERROR  3.88%
  NAME    NAME                              SOLO  TOTAL  .00     1.00     2.00

 IGZCPAC           COBPACK  GENERAL           .78    .78  .********
 IGZEINI           ENVIRONMENT INITIALIZE     .47    .47  .*****
                                            ------ ------
  SECTION    .COBLIB  TOTALS:                1.25   1.25
```

Abbildung 7.2.: Beispiel für eine Tabellendarstellung

elle Interpretation, für ein Parsing jedoch bringen sie einige Schwierigkeiten mit sich, da Textdateien in Zeilen gelesen werden.
Zur Aufteilung der Daten in einzelne Zellen einer solchen Tabelle wird die Klasse *DataTable* zur Verfügung gestellt. In dieser wird ein Modell der jeweiligen Tabelle gebildet, mit dessen Hilfe diese Aufteilung dann vorgenommen werden kann. Die einzelnen Spalten einer Tabelle werden in diesem Modell durch Instanzen der Klasse *Column* repräsentiert. Eine Spalte definiert sich über die horizontale Start- und Endposition innerhalb der Textdatei, im Beispiel 7.2 Spaltenindex 9 bis 15 für die Spalte „SECTION NAME". Die Spaltenbreiten bzw. Start- und Endposition werden grundsätzlich aus den Spaltenüberschriften ermittelt. Allerdings bietet die Klasse *DataTable* die Möglichkeit, diese automatische Ermittlung zu „überschreiben" und eine Spalte auf eine vorgegebene Breite auszudehnen. Diese Funktionalität wird benötigt, wenn eine Spalte eine bekannte Mindestbreite hat, die von der Breite der Überschrift nicht ganz überdeckt wird. Im Beispiel hätte die Spalte „SOLO" auf Grund der Überschrift eine Breite von 4 Zeichen, es müssen jedoch mindestens 5 Zeichen zur Verfügung stehen, damit ein prozentualer Wert von „99.99" dargestellt werden kann.[2]
Die Ausdehnung kann wahlweise nach links oder rechts erfolgen. Die Klasse *DataTable* stellt zudem eine Methode *makeGreedy* bereit, mit der eine Spaltendefinition auf beiden Seiten vollständig bis zu den benachbarten Spalte ausgedehnt wird. Dieses Verhalten wird

[2]Die Darstellung von 100.00% kann vernachlässigt werden, da wohl kein sinnvolles Programm nur aus einem einzigen Modul besteht und keinerlei System- oder Bibliotheksaufrufe verursacht.

benötigt, um im Beispiel 7.2 die Spalte „FUNCTION“ nicht von Index 19 bis 26 zu definieren, wie es die Überschrift eigentlich vorgibt, sondern statt dessen von 16 bis 45.[3]
Im Package *de.tum.in.laner.parserImpl* ist die konkrete Implementierung des Parsers für Strobe-Reports enthalten. Die Klasse *StrobeParser* erbt von *AbstractParser* und stellt die notwendigen Methoden zum Parsen von Strobe-Reports zur Verfügung. Das Package stellt für jede *Section* innerhalb des Reports eine eigene Klasse zur Verfügung, die von *StrobeParser* erbt. Diese spezielle Implementierung übernimmt die zum Parsen des jeweiligen Abschnitts notwendigen Aufgaben.
Das Parsen der Tabellen in den Strobe-Reports erfolgt im Wesentlichen in zwei Schritten. Zunächst wird der Tabellenkopf geparsed und daraus wie oben beschrieben ein Modell der Tabelle (unter Verwendung der Klassen *DataTable* und *Column*) aufgebaut. Da die Tabellenköpfe nicht zwingend die Spaltenbreiten vorgeben, werden im Nachgang noch zusätzliche Informationen über die Breiten einzelner Spalten hinzugefügt. Im zweiten Schritt werden dann die Datenzeilen der Tabelle in einzelne Zellen aufgeteilt, deren Ränder vom gebildeten Modell der Tabelle bestimmt werden. Die Unterscheidung, welcher Schritt aktuell bearbeitet wird (Tabellenüberschriften parsen zur Modellerstellung oder Parsen der Tabellendaten), geschieht mit einer Zustandsvariable. Oftmals haben die Tabellen einen zweizeiligen Kopf, dessen Parsen dann entsprechen auf zwei Zustände aufgeteilt wird.
Die Erkennung und Verarbeitung der Tabellenköpfe geschieht mit Hilfe regulärer Ausdrücke. Eine besondere Schwierigkeit dabei besteht darin, dass verschiedene Strobe-Reports die Kopfzeilen gleicher Tabellen zum Teil unterschiedlich bezeichnen (teilweise abkürzen).

```
---------------------------WAS INVOKED BY----------EXECUTING--------------------TARGET----------------------  CPU TIME %
XACTION  QUERY NAME                  TIME         STMT TEXT                   STMT TEXT                    SOLO  TOTAL

---------------INVOKED BY------------------EXECUTING--------TARGET---CPU TIME %-
XACTION  QUERY NAME                TIME     TEXT     STMT  TEXT     SOLO  TOTAL
```

Abbildung 7.3.: Beispiel für verschiedene Tabellenköpfe

Das Beispiel 7.3 zeigt den Tabellenkopf für SQL-Statements in der Section *Attribution of CPU Execution Time* in zwei verschiedenen Darstellungen, die bei der Erstellung der regulären Ausdrücke berücksichtigt werden müssen. Der zugehörige reguläre Ausdruck ist wie folgt definiert:

```
^\s?([\-]+((WAS )?INVOKED BY|VIA|EXECUTING|TARGET)[\-\s]+\s*)+
([\-\s]*(WAIT|CPU) TIME\s?%)[\-]*\s*$
```

Eine weitere Problematik beim Parsen solcher Reports besteht darin, dass manche Zeilen am linken Rand (in den ersten Zeichen einer Zeile) zusätzliche Leerzeichen, Bindestriche sowie die Ziffern 0 und 1 aufweisen. Diese Zeichen werden in der Strobe Dokumentation nicht erwähnt und sind auch nicht in allen im Rahmen dieser Arbeit untersuchten Reports vorhanden. Eine mögliche Erklärung wäre, dass diese Zeichen durch den Export der Reports vom Terminal in eine Textdatei entstehen. Die jeweiligen reguläre Ausdrücke müssen dieser Tatsache Rechnung tragen.

[3] bzw. 44, wenn zuvor die Spalte „SOLO“ auf 5 Zeichen nach links erweitert wurde

8. Auswertung

Zur eigentlichen Auswertung der Performance-Daten wird eine Software entwickelt, welche die im Kapitel 4 beschriebenen Anforderungen erfüllen soll. Die Software verwendet als Grundlage die Eclipse Rich Client Platform (RCP) [2], da diese viele grundlegende Funktionen bereitstellt, die für dieses Programm nützlich sind.

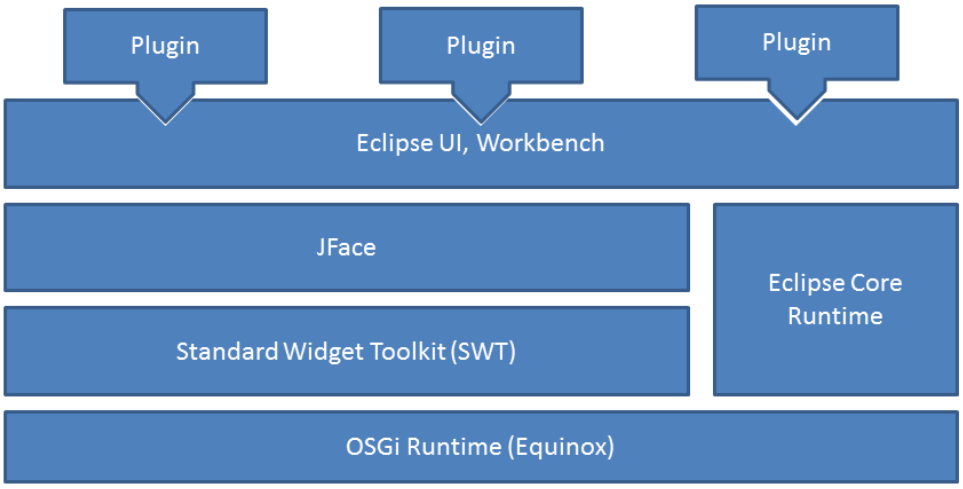


Abbildung 8.1.: Eclipse RCP Architektur

Die Abbildung 8.1 stellt die Architektur einer Eclipse RCP Anwendung dar. Die Architektur-Informationen sind dem Ebook von Ralf Ebert [6, S. 3f] entnommen.
Die Basis bildet Equinox, eine Implementierung einer OSGi Service Platform. Die OSGi Service Platform der OSGi Alliance[1] spezifiziert ein Framework zur Ausführung von Modulen (sog. Bundles) sowie zur Verwaltung von deren Lebenszyklen.
Darauf aufbauend stellt die Eclipse Core Runtime Basisfunktionen, die nicht UI-relevant sind, zur Verfügung. Dies beinhaltet z. B. eine generische Schnittstelle zum Dateisystem.
Die beiden Schichten SWT und JFace bilden die Grundlage für das UI. SWT stellt eine Schnittstelle zu den nativen Widgets des zu Grunde liegenden Betriebssystems bereit, JFace die Schnittstelle zur Befüllung dieser Widgets mit Daten.
Darauf stellt die Eclipse UI die sog. Workbench zur Verfügung. Die Workbench ist der Rahmen für eine Eclipse Anwendung und stellt z. B. Funktionen für die Anordnung verschiedener Ansichten in Perspektiven bereit. Die Workbench wird mit Inhalten über sog.

[1] http://www.osgi.org/, zuletzt aufgerufen am 26.02.2013

Plugins gefüllt. Als Plugin (auch Plug-in) wird nach [6, S. 60] ein OSGi Bundle in Eclipse bezeichnet. Die Eclipse-Dokumentation definiert ein Plug-in wie folgt:[2]

> „A plug-in is a structured component that contributes code (or documentation or both) to the system and describes it in a structured way."

Die Eclipse Rich Client Platform nutzt sog. *Extension Points*, um Schnittstellen für Erweiterungen zu definieren. Ein *Extension Point* definiert dabei lediglich eine Vereinbarung bzgl. dieser Schnittstelle unter Verwendung von XML und Java Interfaces. Ein so definierter *Extension Point* kann von Plugins verwendet werden, um gemäß dieser Schnittstelle Funktionaltiät beizutragen. Das definierende Plugin eines *Extension Points* kennt keine Details der verbundenen Komponenten, sondern lediglich diese Schnittstelle. Dieser Mechanismus wird verwendet, um Plugins zur Suche nach Anti-Pattern sowie zum Import von Performance-Reports einzubinden. Diese Plugins müssen mit einem solchen *Extension Point* spezifizierte Schnittstellen umsetzten und können sich darüber mit der Anwendung verbinden.
Für die Erzeugung der Tortendiagramme nutzt die Eclipse Business Intelligence and Reporting Tools (BIRT) [3]. Für die Steuerelemente des Analyse-Werkzeuges wird das Eclipse SWT [4] mit JFace verwendet.

8.1. Einbindung von Anti-Pattern

Eine zentrale Funktion des Analyse-Werkzeuges ist die automatische Suche nach Anti-Pattern. Dazu wird ein Anti-Pattern mittels einer entsprechenden Klasse (*AntiPattern*) modelliert, deren Klassendiagramm in Abbildung 8.2 dargestellt ist. Neben einem Namen,

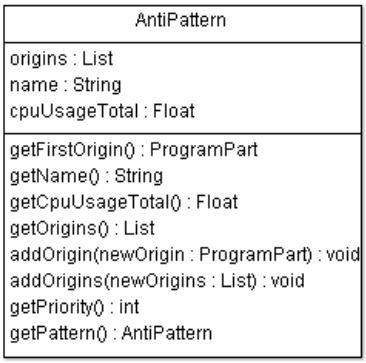


Abbildung 8.2.: Klasse AntiPattern

der das Anti-Pattern bezeichnet (z. B. „Join-zu-Fuß"), verfügt die Klasse über ein Attribut *origins*, welches eine Liste von *ProgramPart*-Instanzen enthält, die für das Vorliegen

[2] http://help.eclipse.org/juno/topic/org.eclipse.platform.doc.isv/guide/runtime.htm, zuletzt aufgerufen am 14.02.2013
[3] http://www.eclipse.org/birt/phoenix/, zuletzt aufgerufen am 18.02.2013
[4] http://www.eclipse.org/swt/, zuletzt aufgerufen am 18.02.2013

des Anti-Patterns verantwortlich sind. Es wird deshalb eine Liste verwendet, da manche Anti-Pattern mehreren Verursachern gemeinsam zugeordnet werden müssen. Beispielsweise sind an einem Join-zu-Fuss mindestens zwei SQL-Statements beteiligt. Zudem wird die gesamte CPU-Nutzung aller beteiligten Verursacher gespeichert. Die beiden Methoden *getPriority* und *getPattern* werden zur Erstellung einer Pattern-Liste von der Benutzeroberfläche benötigt (siehe hierzu Abschnitt 8.2).
Um nun zu entscheiden, ob und in welchen Programmteilen ein bestimmtes Anti-Pattern vorliegt, werden sog. „Anti-Pattern Matcher" verwendet. Diese traversieren den in Abbildung 5.7 exemplarisch dargestellten Baum mit Performance-Daten ausgehend von dessen Wurzel, der *MeasurementSession*. Während der Traversierung werden für die besuchten Knoten Regeln angewandt, um zu entscheiden, ob diese zu einem Anti-Pattern beitragen. Die konkrete Definition dieser Regeln hängt vom jeweiligen Anti-Pattern ab.
Um solche Matcher in die Analyse-Anwendung einzubinden, stellt diese den Extension Point *de.tum.in.laner.strobeanalyse.antiPatternMatcher.AbstractAntiPatternMatcher* zur Verfügung. Klassen, die diesen Extension Point implementieren, müssen von der Klasse *AbstractAntiPatternMatcher* erben und deren abstrakte Methode

```
public abstract List<AntiPattern> find(MeasurementSession ms)
throws AntiPatternMatcherException
```

implementieren, die eine Liste von Anti-Pattern zurückgibt. Ein Aufruf dieser Methode startet die Suche des Matchers nach Anti-Pattern ausgehend von der als Parameter übergebenen *MeasurementSession*. Mittels dieser Schnittstelle können beliebige Matcher in das Analyse-Werkzeug eingebunden werden. Führt dieses eine Suche nach Anti-Pattern durch, werden alle registrierten Plugins der zu Grunde liegenden Eclipse-Plattform ermittelt, welche diesen Extension-Point implementieren. Die zugehörigen Matcher werden dann instanziert und deren *find*-Methode ausgeführt.
Um Parameter für die einzelnen Anti-Pattern Matcher bzw. für deren hinterlegte Regeln konfigurieren zu können, wird der Extension Point *org.eclipse.ui.preferencePages* der Eclipse RCP-Plattform verwendet. Dieser definiert eine Schnittstelle zur Einbindung von Seiten (*PreferencePage*) im Konfigurationsmenü der Anwendung.
Im Rahmen dieser Arbeit wurden folgende Matcher implementiert:

- OpenFetchRationMatcher
- JoinZuFussMatcher
- SpecialModulesMatcher

Sowohl der *OpenFetchRatioMatcher*, als auch der *JoinZuFussMatcher* behandeln Anti-Pattern im Zusammenhang mit der Nutzung von SQL, weshalb diese in einem gemeinsamen Plugin zusammengefasst sind.

8.1.1. OpenFetchRatioMatcher

Der *OpenFetchRatioMatcher* berechnet das Verhältnis zwischen *OPEN* und *FETCH* Operationen von SQL-Cursorn. Wie in 6.1 beschrieben, liegt möglicherweise die Verwendung eines Cursors für einen Einzelsatz-Fetch vor, wenn die Anzahl der *FETCH* Operationen

pro *OPEN* Operation nicht größer als 1, jedoch über einem gewissen Schwellwert (der zwischen 0 und 1 betragen muss) liegt. Werden sehr viele *FETCH* Operationen pro geöffnetem Cursor ausgeführt bedeutet dies hingegen, dass die Anzahl der pro *FETCH* in die Anwendung transferierten Datensätze zu niedrig ist. In diesem Fall liegt das in Abschnitt 6.2 beschriebene Anti-Pattern vor.
Da das mögliche Vorliegen dieser beiden Anti-Pattern jeweils auf Basis dieses Verhältnisses entschieden wird, ist deren Suche in nur einem Matcher zusammengefasst. Dadurch muss dieses Verhältnis nur einmal berechnet werden, bei Verwendung von zwei Matchern hingegen müsste jeder der beiden Matcher es berechnen.
Der Matcher selbst besteht im Wesentlichen aus zwei verschachtelten Schleifen. Diese durchlaufen für alle DBRMs innerhalb des aktuell analysierten Programms alle SQL-Statements. Die beiden Anti-Pattern betreffen nur Statements, die einen Cursor verwenden. Aus diesem Grund werden Non-Cursor-Statements von vorne herein von der Suche ausgeschlossen. Für jedes tatsächlich betrachtete Statement wird dann das *FETCH / OPEN* Verhältnis berechnet und an Hand der konfigurierbaren Schwellwerte entschieden, ob es sich um einen Kandidaten für eines der beiden Anti-Pattern handelt.
Der *OpenFetchRatioMatcher* stellt auch eine *PreferencePage* zur Verfügung, auf welcher der untere Schwellwert für den Einzelsatz-Fetch mittels Cursor sowie der Schwellwert für den Einzelsatz-Fetch bei Massenoperationen konfiguriert werden kann.

8.1.2. JoinZuFussMatcher

Der *JoinZuFussMatcher* behandelt die in Abschnitt 6.3 beschriebenen verwandten Anti-Pattern „Schnitzeljagd“ und „Join-zu-Fuß“. Dazu wird nach SQL-Statements gesucht, die „nahezu identisch oft“ ausgeführt werden.
Um im Sinne dieses Anti-Patterns zusammengehörige Statements zu identifizieren, werden die Statements gemäß ihrer Ausführungszahlen in Gruppen eingeteilt. Ein Statement wird genau dann einer Gruppe zugeordnet, wenn seine Ausführungszahl um höchstens einen bestimmten Wert vom Mittelwert der Ausführungszahlen der Gruppe abweicht. Die maximal zulässige Abweichung vom Mittelwert kann in der zugehörigen *PreferencePage* konfiguriert werden. Als Mittelwert wird das arithmetische Mittel der Ausführungszahlen in der Gruppe enthaltener Statements verwendet. Nach dem Einfügen eines Statements in eine solche Gruppe muss daher der Mittelwert erneut berechnet werden. Wird für ein Statement keine passende Gruppe gefunden, wird eine neue Gruppe erzeugt, deren einziges Mitglied dieses Statement und deren Mittelwert die Ausführungszahl dieses Statements ist.
Cursor-Statements werden sowohl nach Anzahl der *OPEN*, als auch der *FETCH* Operationen eingeteilt (siehe 6.3) und können daher zwei Gruppen zugeordnet werden. Eine Datenbankabfrage kann in einem Join-zu-Fuss die Rolle der äusseren oder der inneren Schleife einnehmen. Die Durchlaufanzahl der äusseren Schleife entspricht der Anzahl der *FETCH* Operationen des äusseren SQL-Statements, sowie der Anzahl der *OPEN* Operationen des inneren Statements. Da aus der Performance-Messung die Rolle des Statements nicht hervorgeht, müssen eben beide Fälle betrachtet werden. In einer Schnitzeljagt ist auch eine tiefere Schachtelung an Schleifen denkbar. Die Statements, die den Schleifen auf den mittleren Ebenen (alle ausser der äußersten bzw. innersten Schleife) entsprechen, nehmen dann beide Rollen ein.

Nach der Ermittlung der Gruppen werden Instanzen von *AntiPattern* für jede Gruppe mit mindestens zwei Statements erzeugt. Das *origins*-Attribut dieser *AntiPattern* enthält die Liste der Statements in der betreffenden Gruppe.
Zur Verwaltung der Gruppen wurde die Klasse *JoinZuFussGroup* eingeführt. Diese führt eine Liste der *SqlStatement* Instanzen, die ihr zugeordnet sind. Zudem speichert sie in einer Instanzvariablen den Mittelwert der relevanten Ausführungszahlen dieser Statements. Beim Hinzufügen eines Statements, welches noch nicht in der Liste enthalten ist, wird der Mittelwert unter Berücksichtigung der neuen relevanten Ausführungszahl erneut berechnet. Die Neuberechnung kann dadurch, dass sowohl Mittelwert als auch Anzahl der Statements vor dem Hinzufügen bekannt sind, ohne eine Iteration über alle enthaltenen Statements erfolgen.
Die Bildung dieser Gruppen erfolgt entweder für jedes DBRM (im Datenmodell Klasse *SqlPlanPackage*) separat oder übergreifend über alle DBRMs. Dies kann vom Nutzer des Programms konfiguriert werden. Eine nicht auf einzelne DBRMs beschränkte Suche liefert ggf. SQL-Operationen, die zwar gleich oft ausgeführt wurden, jedoch fachlich nicht zusammen gehören.
Um zu vermeiden, dass besonders selten ausgeführte Statements als Join-zu-Fuss oder Schnitzeljagd klassifiziert werden, kann eine untere Schranke eingestellt werden, unterhalb derer die Statements nicht in die Betrachtung für diese Anti-Pattern eingehen.

8.1.3. SpecialModulesMatcher

Dieser Matcher sucht speziell nach dem Auftreten bestimmter Module. Die Namen der gesuchten Module können in einer von diesem Plugin zur Verfügung gestellten *PreferencePage* konfiguriert werden. Die Konfiguration kann zweistufig erfolgen. Es kann also z. B. das Modul bzw. die Control Section „XYZ“ unterhalb von „ABC“ gesucht werden. Wird ein solches Modul bzw. eine solche Control-Section in der Messung gefunden, wird dies als Auftreten des Anti-Pattern gewertet.
Damit lässt sich die Ausführung ineffizienter Module aus der COBOL-Bibliothek (siehe Abschnitt 6.4) auffinden. Hierzu wird z. B. nach dem Modul „IGZCPAC“ unterhalb von „.COBLIB“ gesucht. Wie bereits beschrieben, lassen sich auch weitere Suchanfragen konfigurieren. So kann beispielsweise die Ausführung eines Debuggers in produktiven Umgebungen oder die Nutzung weiterer Monitoring-Tools detektiert werden. Dazu muss lediglich der Modulname bekannt sein und eingestellt werden.

8.2. User Interface

Das User Interface bringt nun die einzelnen Bestandteile des Analyse-Werkzeuges zusammen. Es erlaubt dem Benutzer, den Import eines weiteren Strobe-Reports zu starten, zeigt die bereits eingelesenen Reports an, visualisiert die darin enthaltenen Daten und listet die gefundenen Anti-Pattern auf. Zudem erlaubt es, Konfigurationen bequem in Dialogen vornehmen zu können, z.B. für die Parametrisierung der Anti-Pattern Matcher (siehe auch Kapitel 8.1).
Das Programm stellt in der im Rahmen dieser Masterarbeit implementierten Fassung die

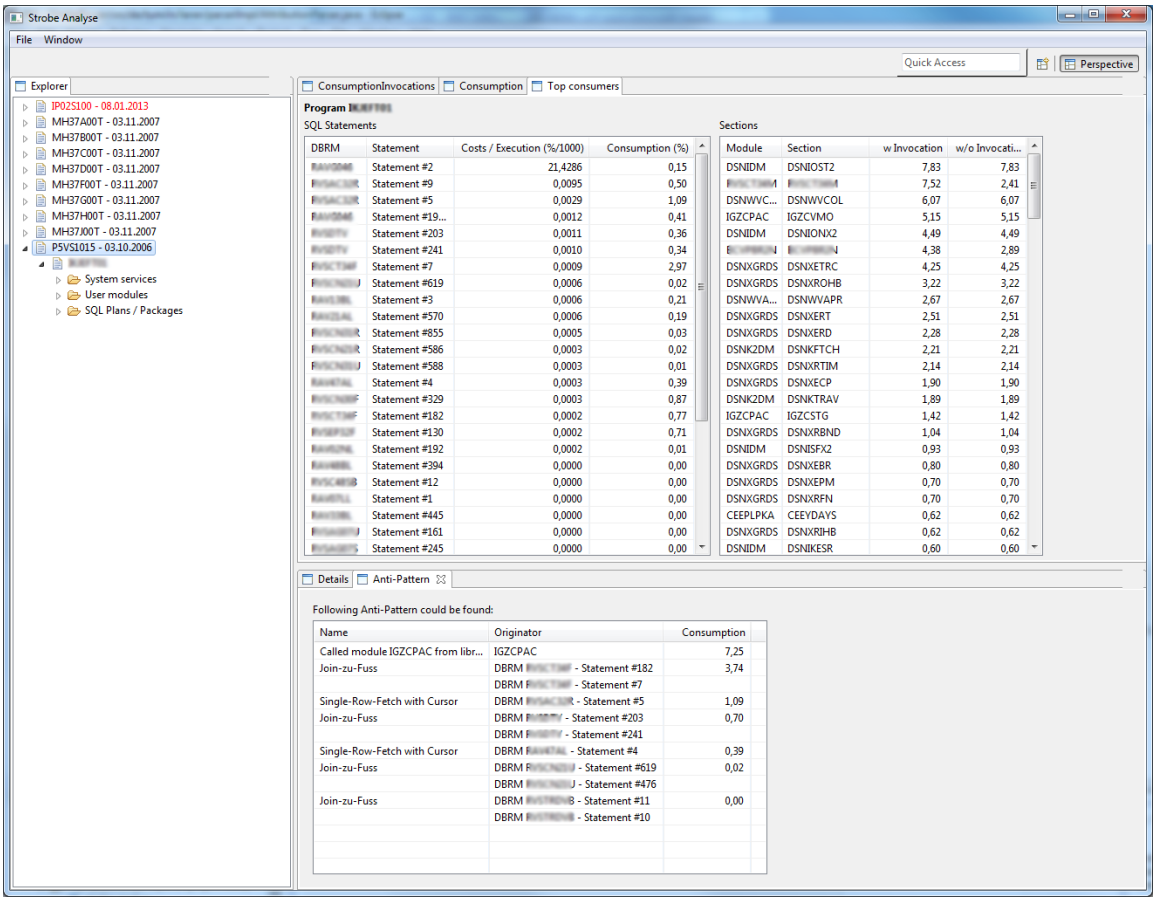


Abbildung 8.3.: Hauptansicht des Analyse-Werkzeuges

CPU-Nutzung (auch als CPU-Verbrauch bezeichnet) dar. Die angezeigten Verbrauchswerte verstehen sich, sofern nicht explizit abweichend angegeben, relativ zum Gesamtverbrauch der gemessenen Anwendung und werden in Prozent angegeben.
Die Abbildung 8.3 zeigt zunächst einen Überblick über das Analyse-Werkzeug. Die verwischten Einträge dienen der Anonymisierung der im Screenshot dargestellten Daten, da diese aus realen Strobe-Reports stammen. Die Anwendung ist grundsätzlich in drei Bereiche unterteilt. Im linken Bereich (Explorer) werden alle bereits eingelesenen Reports aufgeführt. Der rechte Bereich des Werkzeuges teilt sich nochmals in zwei Bereiche auf. Der obere Bereich (Hauptbereich) enthält drei Karteikarten mit den wichtigsten Informationen über die Verteilung des Verbrauchs. Im unteren Bereich wird jeweils eine Karteikarte mit Detailinformationen sowie eine mit Anti-Pattern angezeigt.
In den folgenden fünf Unterkapiteln werden die einzelnen Ansichten bzw. Karteikarten jeweils genauer erläutert.

8.2.1. Explorer

Der Explorer dient als Hauptnavigationselement innerhalb des Programms. Er besteht im Wesentlichen aus einer Baumdarstellung, welche alle eingelesenen Reports und die einzelnen Programmbestandteil der gemessenen Jobs repräsentiert. Eine Messung wird mit dem Name des Jobs und dem Datum der Messung betitelt. Der Knoten einer Messung innerhalb dieses Baumes wird mit roter Schrift dargestellt, wenn die zu Grunde liegende Sampling-Rate unterhalb eines bestimmten (konfigurierbaren) Wertes liegt.
Unterhalb des Knotens für eine Messung werden baumartig das gemessene Programm sowie drei Ordner für Systemdienste (bzw. deren Module), Anwendungsmodule sowie für SQL-Pakete (DBRMs) aufgeführt. Diese Ordner lassen sich anschließend für einen weiteren Drilldown entsprechend ausklappen. Im Datenmodell, welches zur Speicherung der Performance-Daten verwendet wird (siehe Abschnitt 5.3), sind diese Ordner nicht enthalten. Sie werden vom Analyse-Werkzeug zur Verbesserung der Übersichtlichkeit erzeugt und eingefügt.

8.2.2. Top Consumers

Beim Aufruf eines Reports bzw. beim Wechsel zwischen verschiedenen Reports wird im Hauptbereich zunächst die Karteikarte „Top consumers“ angezeigt. Zweck dieser Ansicht ist es, schnell sog. Top-Verbraucher, also Programmteile innerhalb des gemessenen Jobs, die sehr viel CPU-Nutzung relativ zu anderen Programmteilen verursachen, zu identifizieren. Die Ansicht besteht aus zwei Tabellen.
Die linke Tabelle listet den Verbrauch aller SQL-Statements auf. Der Verbrauch eines Statements ist dabei die Summe des Verbrauchs seiner Operationen. Zusätzlich zum Gesamtverbrauch des Statements für alle Ausführungen wird der Verbrauch pro Ausführung berechnet. Um die Lesbarkeit/Übersichtlichkeit zu verbessern, werden die Verbräuche pro Ausführung in tausendstel Prozent angegeben. Dadurch lassen sich sehr kleine Zahlenwerte mit vielen vorausgehenden 0-Ziffern vermeiden. Die Tabelle kann nach diesen beiden Verbrauchsspalten absteigend sortiert werden, indem der jeweilige Spaltenkopf angeklickt wird. Zur Erkennung, welche SQL-Statements insgesamt hohen Verbrauch verursachen oder welche Statements viel CPU-Nutzung pro einzelner Ausführung verursachen, muss lediglich eine Sortierung nach der jeweiligen Spalte erfolgen.
Die zweite Tabelle führt die CPU-Nutzung aller Control Sections auf. Diese Auflistung ist unabhängig davon, ob es sich um Control Sections von Systemdiensten oder Anwendungsmodulen handelt. Der Verbrauch wird auch hier in zwei absteigend sortierbaren Spalten dargestellt. Die rechte Spalte („w/o Invocation“) beinhaltet den Verbrauch der jeweiligen Control Section selbst. In der Spalte links daneben („w Invocation“) wird zusätzlich berücksichtigt, wenn innerhalb einer Control Section weitere Programmteile aufgerufen wurden (sog. „Invocation“). Bei der Berücksichtigung solcher Aufrufe wird der dadurch in der aufgerufenen Control Section (Callee) verursachte Verbrauch dem Aufrufer (Caller) zugeschlagen. Im Gegenzug wird der Verbrauch des aufgerufenen Programmteils entsprechend vermindert. Dieser Betrachtung liegt die Tatsache zu Grunde, dass in der Regel der Callee Bestandteil eines Systemmoduls ist, dessen Code sich nicht optimieren lässt, die Ursache des Verbrauchs jedoch vom Aufrufer zu verantworten ist. Sind im Datenmodell keine Aufrufe hinterlegt, sind die beiden Verbrauchswerte identisch.

Die Abbildung 8.3 beinhaltet ein Beispiel für diese Situation. Hier hat die in der zweiten Zeile aufgeführte, anonymisierte Control Section selbst einen Verbrauch von 2,41%. Unter Anrechnung seiner gemachten Aufrufe jedoch verursacht sie 7,52%.
In beiden Tabellen kann für eine weitere Analyse durch einen Doppelklick auf eine Zeile direkt zum jeweilige SQL-Statement bzw. zur jeweiligen Control Section innerhalb des Analyse-Werkzeuges gesprungen werden.

8.2.3. Consumption

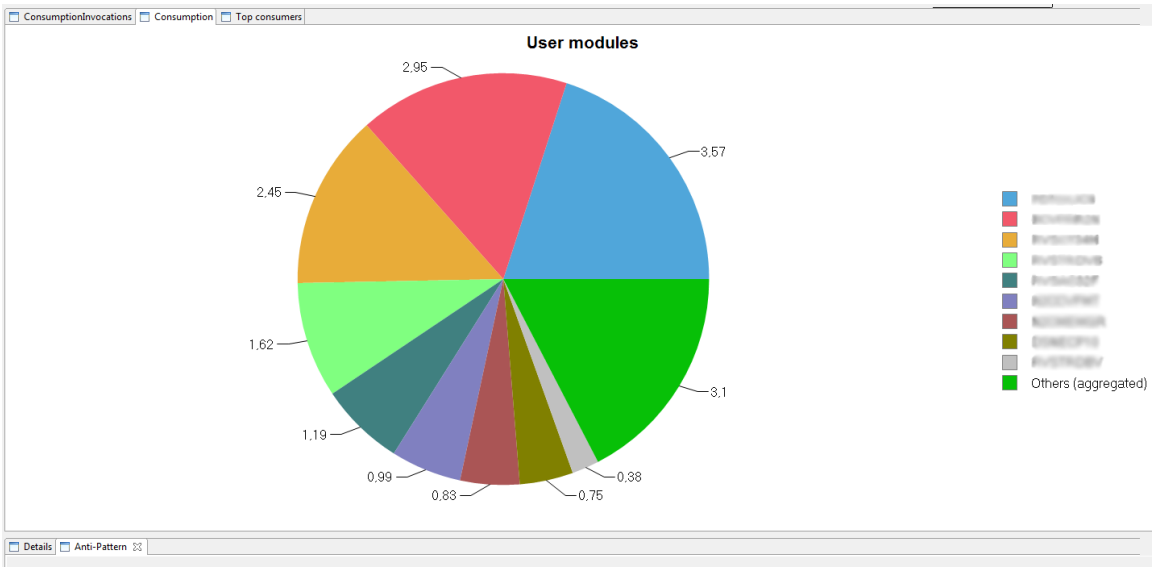


Abbildung 8.4.: Tortendiagramm zur Visualisierung des Verbrauchs in Subkomponenten

Die beiden Karteikarten „Consumption“ und „Consumption Invocation“ werden im Hauptbereich angezeigt. Sie stellen die Aufteilung des Verbrauchs des ausgewählten Programmbestandteils in seine untergeordneten Bestandteile dar. Die Darstellung erfolgt grafisch in Form eines Tortendiagramms. Der Aufbau dieser beiden Ansichten ist identisch. Der einzige Unterschied besteht in der zu Grunde liegenden Datenbasis. Unter „Consumption Invocation“ erfolgt die bereits beschriebene Verrechnung von Verbräuchen aufgerufener Programmteile, in der Ansicht „Consumption“ hingegen wird der isolierte Verbrauch betrachtet. Eine gesonderte Darstellung des Verbrauchs unter Anwendung der Verrechnungsmethodik ist notwendig, da sich dadurch ggf. eine veränderte Verbrauchsaufteilung ergibt.
Zur Erstellung dieser Diagramme kann ein Schwellwert konfiguriert werden (in Prozent der Gesamtsumme des Diagramms). Für Programmteile, deren Verbrauch weniger als dieser Schwellwert beträgt, wird kein eigenes Tortenstück erzeugt. Die Verbräuche dieser Teile werden zu einem gesamten Stück („others“) aggregiert.
Wird im Tortendiagramm ein Stück angeklickt (mit Ausnahme von „others“), so wird der von diesem repräsentierte Programmbestandteil (z. B. Modul ABCD) selektiert. Das

Diagramm zeigt dann dessen Verbrauchsaufteilung in Subkomponenten (z. B. Control Sections von ABCD) an. Dieses Verhalten entspricht der Durchführung eines Drilldowns. Das Beispiel in Abbildung 8.4 zeigt die Aufteilung des Verbrauchs für den Ordner „User modules“ ohne Berücksichtigung gegenseitiger Aufrufe. Der Schwellwert zur Aggregation ist hierbei auf 2% der Gesamtsumme des Diagramms (entspricht hier dem Gesamtverbrauch aller Anwendungsmodule) eingestellt.

8.2.4. Anti-Pattern

Die Karteikarte „Anti-Pattern“ (siehe Abbildung 8.3) listet die gefundenen Anti-Pattern auf. Die Auflistung ist absteigend sortiert nach dem Verbrauch, den diejenigen Programmteile verursachen, denen dieses Anti-Pattern zugerechnet wird. Die Spalte „Originator“ gibt an, welche Programmteile dies sind. Auch in dieser Ansicht kann durch einen Doppelklick auf die jeweilige Zeile direkt der Verursacher aufgerufen werden. Nicht alle Anti-Pattern können jedoch einem Programmteil eindeutig zugeordnet werden. Einige Anti-Pattern entstehen erst aus dem Zusammenspiel mehrerer Programmteile (z. B. der Join-zu-Fuss). In diesem Fall wird der Name und Verbrauch nur einmal in der Tabelle aufgeführt. Alle weiteren Verursacher werden darunter in einer eigenen Zeile angezeigt. In einer Zelle einer SWT Tabelle kann nicht ein weiteres SWT Element enthalten sein. Dies würde jedoch benötigt, um die Zeile in einem solchen Fall in der Höhe entsprechend zu vergrößern und innerhalb einer Zelle alle Verursacher in einer Liste aufzuführen. Eine einfache Auflistung in einer Zelle (mit Zeilenumbruch) kann nicht verwendet werden, da beim Anklicken die Elemente der Liste nicht mehr unterschieden werden können (lediglich der Klick auf eine Zelle einer SWT Tabelle kann erfasst werden) und somit ein direkter Sprung zum jeweiligen Programmteil nicht möglich ist. Aus diesem Grund wird für die Erzeugung der Anti-Pattern-Tabelle bei einem Anti-Pattern mit mehreren Verursachern einmal das Anti-Pattern selbst sowie für jeden Verursacher eine *AntiPatternDummy*-Instanz in das zu Grunde liegende Modell (JFace ContentProvider) eingefügt. Die Klasse *AntiPatternDummy* ist eine Subklasse von *AntiPattern* (siehe 8.2). Um zu unterscheiden, ob ein *AntiPattern* das tatsächliche Anti-Pattern oder eines dieser Dummys repräsentiert, wird die Methode *getPriority* verwendet. Diese gibt für das *AntiPattern* selbst den Wert 0 zurück, das *AntiPatternDummy* überschreibt diese Methode und liefert den Wert 1. Die Methode *getPattern* ist nach dem selben Muster aufgebaut wie *getPriority* und liefert für *AntiPattern* eine Referenz auf sich selbst, für *AntiPatternDummy* eine Referenz auf das eigentliche Anti-Pattern. Darüber ist der Zugriff auf das Anti-Pattern einheitlich.

8.2.5. Details

Die Karteikarte „Details“ liefert weiterführende Informationen zum aktuell ausgewählten Element. Die Abbildung 8.5 zeigt die Standarddarstellung dieser Karte.
Die Tabelle im linken Bereich listet zusätzliche Daten zum aktuellen Programmteil auf. Der Inhalt dieser Tabelle ist abhängig vom Typ dieses Programmteils. So wird beispielsweise für die Messung selbst die Sampling-Rate angezeigt oder für Systemmodule eine zusätzliche Kurzbeschreibung.
Der rechte Teil der Ansicht zeigt die Aufruf-Abhängigkeiten des aktuell betrachteten

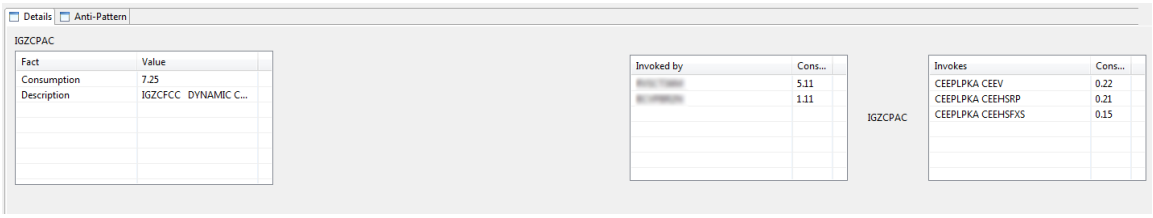


Abbildung 8.5.: Detailansicht mit Aufrufern

Programmteils. Dessen Name wird in der Mitte zwischen zwei Tabellen angezeigt (im Beispiel „IGZCPAC“). Die linke Tabelle zeigt, von welchen Programmteile der aktuell betrachtete Programmteil aufgerufen wird (Callers). In der rechten Tabelle werden die von ihm selbst aufgerufenen Programmteile (Callees) aufgeführt. Die Anordnung wurde so gewählt, da die Leserichtung von links nach rechts geht und somit der Zusammenhang für den Optimierungsexperten intuitiv klar wird. Der zeitliche Ablauf von Aufrufen an einen Programmteil B durch ein Programmteil A ist, dass zuerst Teil A aufgerufen wird, welcher anschließend B aufruft. Ruft nun B wiederum einen Teil C auf, so ergibt sich eine Aufruf-Reihenfolge A-B-C. Ist nun B der aktuell betrachtete Programmteil, so würde A in der linken Tabelle, C in der rechten Tabelle und B zwischen den beiden Tabellen erscheinen.
Sowohl für Aufrufer als auch für aufgerufene Programmteile werden die diesen zugeordneten Verbrauchswerte angezeigt. Auch diese beiden Tabellen erlauben (ebenfalls durch einen Doppelklick) die Navigation zu einem Caller bzw. Callee.

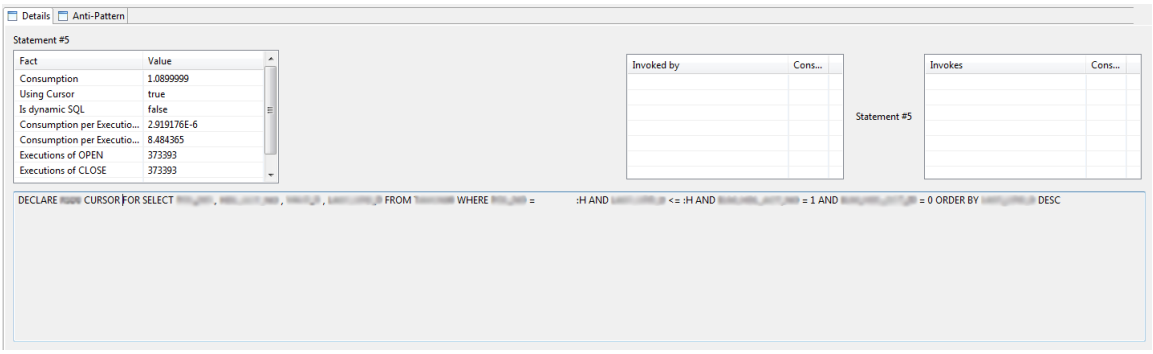


Abbildung 8.6.: Detailansicht für SQL-Statements

Für SQL-Statements besitzt diese Karte eine erweiterte Ansicht (siehe Abbildung 8.6). Hier wird im unteren Bereich zusätzlich der SQL-Text des Statements angezeigt, so wie er von Strobe ausgegeben wird. Die Faktentabelle führt hierbei auch die Ausführungszahlen der einzelnen SQL Operationen auf.

9. Evaluierung

Zur Evaluierung wurde eine Befragung an Performance-Experten verschickt. Die Experten wurden gebeten, Ihre Einschätzung zu fünf Fragen abzugeben und diese auch zu begründen. Die gestellten Fragen werden im folgenden Abschnitt 9.1 aufgeführt und beziehen sich speziell auf bereits umgesetzte oder mögliche künftige Funktionen des Analyse-Werkzeuges. Damit soll evaluiert werden, ob das vorgestellte Werkzeug eine Relevanz für die tägliche Arbeit von Optimierungsexperten hat. Die Ergebnisse dieser Befragung werden hierzu im Abschnitt 9.2 zusammengefasst.

9.1. Evaluierungsfragen

Die folgenden Fragen wurden als Umfrage an Performance-Experten verschickt:

Eval. 1 Wäre ein Werkzeug zur Analyse von Performance-Daten für Sie als Performance-Optimierer hilfreich? Bitte begründen Sie kurz Ihre Antwort.

Eval. 2 Ist es nützlich automatisch nach gewissen, definierten Anti-Pattern innerhalb von Performance-Daten zu suchen? Bitte begründen Sie kurz Ihre Antwort.

Eval. 3 Können Sie sich vorstellen, weitere Anti-Pattern während Ihrer Arbeit als Performance-Optimierer zu identifizieren, für die es sinnvoll wäre, sie in die automatische Suche zu integrieren? Bitte begründen Sie kurz Ihre Antwort.

Eval. 4 Halten Sie es für hilfreich, die Verteilung des Verbrauchs einer Komponente auf ihre Subkomponenten mittels eines Tortendiagramms zu visualisieren? Bitte begründen Sie kurz Ihre Antwort.

Eval. 5 Welche anderen Funktionen würden Sie sich für ein solches Werkzeug wünschen?

a) Integration in eine IDE

b) Vergleich zweier Messungen unterschiedlichen Datums, z. B. bevor und nachdem eine Optimierung durchgeführt wurde

c) Verbindung zu Datenbank Informationen (Tabellenkardinalitäten, Indexdefinitionen etc.)

d) Andere Funktionen. Bitte beschreiben Sie diese.

9.2. Evaluierungsauswertung

Die sechs befragten Experten sind Informatiker mit mehreren Jahren Erfahrung in der Analyse, Wartung und Optimierung von Softwaresystemen. Eine der befragteten Personen beschäftigt sich bereits seit mehr als zehn Jahren mit dieser Thematik. Die Tabelle 9.1 gibt einen kurzen Überblick über den befragten Personenkreis und deren Berufserfahrung in Jahren.

	Erfahrung
Experte 1	über 5 Jahre
Experte 2	über 10 Jahre
Experte 3	7 Jahre
Experte 4	5 Jahre
Experte 5	3 Jahre
Experte 6	1 Jahr

Tabelle 9.1.: Erfahrungslevel der befragten Experten

Die Auswertung ihrer Antworten ergab folgende Ergebnisse:
Alle Experten waren sich einig, dass gemäß Frage Eval. 1 ein Analyse-Werkzeug sie in ihrer täglichen Arbeit unterstützen könnte. Gerade die große Datenmenge solcher Profiling-Reports, sowie der zur Analyse erforderliche Zeitaufwand begründen dessen Notwendigkeit. Ein Experte betonte speziell den Nutzen eines solchen Werkzeuges zur schnellen Identifikation von Programmteilen mit besonders hohem Verbrauch (Top-Verbraucher). Ein weiterer Experte merkte jedoch an, dass eine solche Software zwar eine Analyse erleichtern, jedoch eine individuelle (und somit manuelle) Analyse nicht vollständig ersetzen kann.
Eine automatische Suche nach Anti-Pattern (Eval. 2) halten nahezu alle befragten Experten für sinnvoll. Allerdings lassen sich mit einer solche Suche laut ihrer Einschätzung lediglich sehr einfache Muster erkennen und auch nur Hinweise auf diese liefern. Diese jedoch sind für eine Optimierung sehr hilfreich, da die Ursachen für solche Anti-Pattern häufig mit verhältnismäßig geringem Aufwand behoben werden können. Dies führt oftmals zu kurzfristig realisierbaren Einsparungen. Derartige Optimierungen bieten also ein gutes Aufwand/Nutzen-Verhältnis.
Ein Experte merkte an, dass eine Unterscheidung danach benötigt wird, wieviel Verbrauch der Programmteil verursacht, der ein gefundenes Anti-Pattern enthält. Diese Unterscheidung wird benötigt, da ein Anti-Pattern in einem Programmteil mit sehr geringem Verbrauch zwar grundsätzlich eine schlechte Lösung darstellt, jedoch eine Optimierung unter Performance- und Kostenaspekten nicht sehr lohnenswert ist.
Die Frage Eval. 3 nach zusätzlichen Anti-Pattern, die durch weitere Analysen in Zukunft entdeckt werden und in diese automatische Suche integriert werden sollen, wurde ähnlich zur Frage Eval. 2 beantwortet. Allgemein werden regelmäßig weitere Anti-Pattern identifiziert. Die Optimierungsexperten vermuten allerdings, dass sich nur ein geringer Anteil davon auf Basis von Performance-Daten aus Profilern automatisch erkennen lässt. Dennoch fänden sie eine Erweiterbarkeit des Analyse-Werkzeuges um weitere Anti-Pattern

sinnvoll.
Die Visualiserung der Verbrauchsaufteilung von Sub-Komponenten innerhalb einer Komponente mittels eines Tortendiagramms (Frage Eval. 4) bewerteten alle befragten Optimierungsexperten als hilfreich. Ein Experte betonte die Unterstützung der Kommunikation mit den Verantwortlichen der analysierten Anwendung. Ein anderer Experte merkte an, dass die Definition der Verbrauchsaufteilung wesentlichen Einfluss auf die Ergebnisse einer solchen Darstellung habe, eine reine Aufteilung nach Verbrauch der jeweiligen Sub-Komponenten also andere Ergebnisse liefere als eine Aufteilung inklusive des Verbauchs aufgerufener Komponenten. Ein weiterer Befragter schlug vor, die Erzeugung des Diagramms nicht fest an den Verbrauch zu koppeln, sondern auch eine Möglichkeit anzubieten, andere Daten (z. B. Laufzeit) entsprechend zu visualisieren.
Bei der Frage nach weiteren Funktionen eines Analyse-Werkzeuges (Frage Eval. 5) unterscheiden sich die Antworten der Optimierungsexperten deutlicher. Lediglich eine Integration in eine IDE wurde von keinem der Befragten explizit gewünscht.
Die Möglichkeit zum Vergleich zweier Messungen wurde insgesamt schon gewünscht. Einige Experten stuften dies jedoch als völlig eigenständige Thematik ein. Durch eine Optimierung verändern sich die Bedingungen einer Messung (z. B. durch verkürzte Laufzeit). Dies müsste für einen aussagekräfigen Vergleich zweier Messungen „normalisiert“ werden. Diese „Normalisierung“ ist nach Meinung der Experten nicht trivial, sondern unterliegt einer sehr hohen Komplexität.
Die Verbindung zu Datenbankinformationen wurde sehr unterschiedlich bewertet. Einige Experten stufen diese Funktion als sehr wichtig ein, andere Experten hingegen als nicht besonders wichtig. In diesem Zusammenhang wurde speziell eine Verbindung zu DB2 Explain[1] bzw. dessen Ausgaben genannt, die bei einem Analyse-Werkzeug sinnvoll sein könnte. Die Umsetzbarkeit dieser Funktion wird zudem als schwierig betrachtet, da ein Online-Zugang zu Datenbanken in der Regel nicht möglich ist.
Als weitere Funktionen eines solchen Werkzeuges wurden der Einsatz in Verbindung mit den gängigen Profilern (Strobe, APA, TriTune) sowie die Analyse der Aggregation mehrerer Messungen genannt. Als Anwendungsbeispiel für diese Aggregation nennt ein Experte die Messung von zwei CICS-Regions. Daraus möchte er eine aggregierte Verbrauchsdarstellung unter Berücksichtigung der Gewichtung der einzelnen Messungen.
Mehrere der befragten Experten benötigen zusätzlich eine Exportfunktion der dargestellten Daten, um diese z. B. mit Excel weiter aufbereiten zu können.
Ein Experte möchte die eingelesenen Performance-Daten als eine Art AST (abstract syntax tree) zugreifbar, um mit einem Visitor selbst weitere Funktionen auf diesen Daten ausführen zu können. Als konkretes Anwendungsbeispiel führt er den Aufbau einer Datenbank mit Performance-Daten an, die man mit einem geeigneten Visitor daraus befüllen könnte.

[1] DB2 Explain ist ein Programm, welches für SQL Statements, die auf einer IBM DB2 Datenbank ausgeführt werden, Informationen über die verwendeten Zugriffspfade und Indexnutzungen liefert.

Teil IV.

Zusammenfassung

10. Zusammenfassung und Ausblick

Im folgenden Abschnitt 10.1 dieses Kapitels werden zunächst die Ergebnisse noch einmal zusammengefasst. Im Abschnitt 10.2 wird die praktische Relevanz diskutiert. Diese Master's Thesis endet mit einem Ausblick (Abschnitt 10.3) auf über den Rahmen dieser Arbeit hinausgehende, weiterführende Arbeiten.

10.1. Zusammenfassung

Ziel dieser Arbeit war es, ein Software-Werkzeug zu entwickeln, welches Performance-Experten die Auswertung von Reports des Sampling basierten Profilers „Strobe" erleichtert. Im Fokus der Arbeit stand eine schnelle Identifikation von Optimierungspotentialen durch das Aufzeigen von Hot-Spots und durch eine automatische Suche nach definierten Anti-Pattern.
Zur Entwicklung eines solchen Werkzeuges wurden zunächst in Gesprächen mit Performance-Experten Anforderungen erfasst. An diesen Anforderungen orientiert wurde das Werkzeug dann entworfen und implementiert. Zudem wurden vier Anti-Pattern, nach denen innerhalb der Daten von Profiling-Reports gesucht werden kann, konkret definiert. Für die Suche nach diesen Anti-Pattern wurden Algorithmen entworfen und implementiert.
An Hand einer Befragung an Performance-Experten wurde schließlich evaluiert, ob ein solches Werkzeug für ihre tägliche Arbeit nützlich und hilfreich sein kann. Die Ergebnisse der Befragung wurden schließlich mit den Möglichkeiten des entwickelten Ansatzes verglichen.

10.2. Diskussion

Unzureichende Performance von Anwendungen führt gerade auf Mainframe-Systemen auf Grund der nutzungsabhängigen Lizenzmodelle zu hohen Kosten. Um diese Kosten effizient zu senken, müssen Optimierungspotentiale innerhalb der Anwendungen identifiziert werden. Gängige Praxis dazu ist es, die Anwendungsperformance mit einem Profiler zu messen und die daraus erzeugten Reports zu analysieren. Ausgehend vom Ziel, diese Analysen schneller durchführen zu können, wurden im Kapitel 4 Anforderungen ermittelt, die an ein entsprechendes Softwarewerkzeug gestellt werden. Das im Rahmen dieser Arbeit entwickelte Werkzeug erfüllt diese, bis auf die Anforderungen Anf. 4.1 e) („Hinterlegung einer Knowledge-Base") und Anf. 4.2 d) („Verbindung zum Sourcecode").
Die Ergebnisse der durchgeführten Experteninterviews zeigen, dass die Funktionen, welche Performance-Experten von einem solchen Programm erwarten, weitgehend von dem entwickelten Werkzeug zur Verfügung gestellt werden. So wird der Einwand eines Experten auf die Frage Eval. 2, dass der Verbrauch eines Programmteils, welcher ein Anti-Pattern

beinhaltet, angezeigt werden muss, vom erstellten Werkzeug erfüllt. Auch der Frage Eval. 3 nach einer späteren Erweiterung um zusätzliche Anti-Pattern, welche von Experten als relevant eingestuft wird, kann Rechnung getragen werden, da weitere *AntiPatternMatcher* als Plugins eingebunden werden können (siehe Abschnitt 8.1).
Da die entwickelte Lösung eine modulare Architektur einsetzt, kann sie als erweiterbares Framework für Performance-Analysen betrachtet werden. Die Einbindung von Anti-Pattern hinterlegt dauerhaft bereits gewonnenes Expertenwissen und vereinfacht dessen Transfer.

10.3. Ausblick

Die Erweiterungsmöglichkeiten mittels Plugins bieten Anknüpfungspunkte für weitergehende Arbeiten. Einen zentralen Bestandteil bei der Suche nach Optimierungspotentialen stellt die automatische Suche nach Anti-Pattern dar. Im Rahmen dieser Arbeit wurden bereits vier Anti-Pattern beschrieben und deren Suche implementiert. Laut der durchgeführten Umfrage erwarten Performance-Experten, dass in Zukunft neue Anti-Pattern identifiziert werden. Die Implementierung und Integration von Suchalgorithmen für weitere Anti-Pattern könnte den Nutzen der entwickelten Lösung erweitern.
Diese Arbeit verwendet als Basis Daten aus dem Profiling-Werkzeug „Strobe“. Zum Profiling von Anwendungen auf Mainframes werden jedoch auch andere Profiler verwendet, welche ähnliche Daten liefern können. Die Entwicklung von Parsern für diese Profiler könnte auch für daraus erzeugte Reports leichtere Auswertungen ermöglichen.

Anhang

Abkürzungsverzeichnis

AST	Abstract syntax tree
BDAM	Basic direct access method
BIRT	Eclipse business intelligence and reporting tools
BPAM	Basic partitioned access method
BSAM	Basic sequential access method
CSECT	Control section
DASD	Direct access storage device
DBRM	Database request module
JCL	Job control language
JES	Job entry subsystem
KSDS	Key sequential data set
LoC	Lines of code
LPAR	Logical partition
MDD	Model driven development
QSAM	Queued sequential access method
RAIM	Redundant array of independent memory
RCP	Eclipse rich client platform
SWT	Standard widget toolkit
UI	User interface
VSAM	Virtual sequential access method

Literaturverzeichnis

[1] The return of the mainframe - Back in fashion. In: *The Economist* (2010), Januar

[2] *Eclipse Rich Client Platform*. `http://wiki.eclipse.org/Rich_Client_Platform`. Version: Dezember 2012

[3] *IBM Unveils zEnterprise EC12, a Highly Secure System for Cloud Computing and Enterprise Data*. `http://www-03.ibm.com/press/us/en/pressrelease/38653.wss`. Version: August 2012

[4] COMPUTERWORLD: *MIPS Management at Mainframe Organizations*. August 2007

[5] COMPUWARE: *Strobe Interpretation and Analysis User Guide*. Release 4.3. Compuware, November 2011

[6] EBERT, Ralf: *Eclipse RCP Entwicklung von Desktop-Anwendungen mit der Eclipse Rich Client Platform 3.7*. Bd. 1.1. 2011

[7] FISCHER, Peter ; HOFER, Peter: *Lexikon der Informatik*. Springer Berlin / Heidelberg, 2011. – ISBN 978–3–642–15126–2

[8] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Design Patterns - Elements of Reusable Object-Oriented Software*. Bd. 1. Addison-Wesley Longman, 1994. – ISBN 978–0201633610

[9] IBM: *The IBM zEnterprise System – A new dimension in computing*. `http://www-01.ibm.com/cgi-bin/common/ssi/ssialias?infotype=an&subtype=ca&htmlfid=877/ENUSZG10-0249&appname=isource&language=enus`, Juli 2010

[10] KWIATKOWSKI, L.M. ; VERHOEF, C.: Reducing operational costs through MIPS management.

[11] LAMPERSTORFER, Thomas: *Statische Performance-Analyse*, TU München, Diplomarbeit, Apr. 2011

[12] LANER, Stefan ; HAUDER, Matheus: Performance-Analyse auf Mainframe-Systemen mittels Profiling. In: SAAKE, Gunter (Hrsg.) ; HENRICH, Andreas (Hrsg.) ; LEHNER, Wolfgang (Hrsg.) ; NEUMANN, Thomas (Hrsg.) ; KÖPPEN, Veit (Hrsg.): *BTW Workshops*, GI, 2013 (LNI). – ISBN 978–3–88579–610–7, S. 247–256

[13] NETHERCOTE, Nicholas: *Dynamic Binary Analysis and Instrumentation*, University of Cambridge, Diss., November 2004

[14] PARNAS, David L.: Software aging. In: *Proceedings of the 16th international conference on Software engineering*. Los Alamitos, CA, USA : IEEE Computer Society Press, 1994 (ICSE '94). – ISBN 0–8186–5855–X, 279–287

[15] PARSON, Trevor ; MURPHY, John: Detecting Performance Antipatterns in Component Based Enterprise Systems. In: *Journal of Object Technology* 7 (2008), März, Nr. 3, S. 55–90

[16] SHEDIAK, Tony: Performance Tuning Mainframe Applications "Without trying too hard". (2002)

[17] SMITH, C. U. ; WILLIAMS, L. G.: New Software Performance AntiPatterns: More Ways to Shoot Yourself in the Foot. In: *Int. CMG Conference*, Computer Measurement Group, Dezember 2002, S. 667–674

[18] SMITH, C. U. ; WILLIAMS, L. G.: More New Software Performance AntiPatterns: Even More Ways to Shoot Yourself in the Foot. In: *Int. CMG Conference*, 2003

[19] SMITH, Connie U. ; WILLIAMS, Lloyd G.: Software performance antipatterns. In: *Proceedings of the 2nd international workshop on Software and performance*. New York, NY, USA : ACM, 2000 (WOSP '00). – ISBN 1–58113–195–X, 127–136

[20] THOMAS, Bob: *90% of Fortune 1000 Now Use Mainframes*. `http://enterprisesystemsmedia.com/article/90-of-fortune-1000-now-use-mainframes`, August 2009