Ausarbeitung

Entwicklung eines Trace-Monitors zum Debuggen der Elster NPP-Geräteserie über eine Netzwerkverbindung

Author:
Tobias Wessels

Professor:
Prof.Dr.
Erik Kamsties

2. Juli 2012

Inhaltsverzeichnis

1 Einleitung

Die Firma Elster-Instromet GmbH entwickelt und vertreibt Geräte zur Messung, Regelung und Umwertung von Erdgas. Diese werden von Gaslieferanten in den Verteilerstellen eingesetzt um den Durchfluss zu regeln oder um fiskalische Abrechnungen durchzuführen. Typisch handelt es sich um eingebettete Systeme, mit kleinem Speicher und geringer Rechenleistung. Über viele Jahre hinweg arbeiten sie ohne Unterbrechung und verarbeiten Sensordaten.

Zurzeit arbeitet das Entwicklerteam an einer gemeinsamen Plattform für alle Geräte. Der Wunsch ist es, Anwendungen nur einmal für alle Geräte schreiben zu müssen. Dazu wird eine Funktionalität in einem eigenen Modul gekapselt, welche dann auf dem Basissystem ausgeführt wird. Module können einfach ausgetauscht und nachgeladen werden. Zum Beispiel könnte ein Mengenumwerter für Gas über ein Update dazu gebracht werden Flüssigkeiten zu messen.

Bei dieser Systemarchitektur gibt es **viele, parallel laufende Prozesse**. Diese Nebenläufigkeit begünstigt natürlich bekannte Probleme beim konkurrierenden Zugriff auf gemeinsame Daten. Dazu gehören Verklemmungen (oder Deadlocks), Starvation (Thread „verhungert“, da ein anderer Thread eine Ressource nicht freigibt) oder Synchronisationsprobleme. Da der Compiler nur eine syntaktische- aber keine semantische Analyse des Codes vornimmt, machen sich diese Probleme erst zur Laufzeit bemerkbar und sind somit schwerer ausfindig zu machen.

Fehler sind zudem stark abhängig von der verwendeten Konfiguration. In dem Mengenumwerter FC1 z.B. können bis zu sieben IO-Karten eingebaut werden. Je nach Parametrierung des Gerätes werden an den Eingängen verschiedene Messgrößen verarbeitet. Da Ein- und Ausgänge beliebig miteinander verschaltet werden können, entstehen so schnell Fehler.

Tritt bei einem Kunden ein Problem auf, besteht zurzeit keine Möglichkeit eine schnelle Fehlersuche vorzunehmen. In der Regel fährt ein Außendienst-Mitarbeiter raus um Fehler über die serielle Schnittstelle auszulesen. Bei schweren Fehlern wird das komplette Gerät mitgenommen.

Die Geräte dieser Plattform laufen auf einem ARM-basiertem Mikrocontroller der Firma Atmel. Neben drei Ethernet-Ports, und USB-Anschluss gibt es einen SD-Karten Steckplatz. Dieser wird dazu genutzt das Betriebssystem von einer SD-Karte zu booten. Als Echtzeit-Betriebssystem kommt dabei Integrity von der Firma Green Hills Software zum Einsatz.

Es wird eine **Schnittstelle** benötigt, um das Gerät zur Laufzeit zu untersuchen. Sie soll nicht nur den Entwicklern bei ihrer täglichen Arbeit dienen, sondern auch den Außendienstmitarbeitern zur Fehlersuche. In den Geräten ist zwar ein kleines Display verbaut, jedoch eignet sich dieses aufgrund der Größe nicht fürs Debugging. Auch die vorhandene Terminalverbindung über serielle Schnittstelle eignet sich nicht, da nur reiner Text übertragen wird und sich keine Gruppierung oder Filterung vornehmen lässt.

2 Ziele

Tracing soll zur Überwachung der NPP-Geräteserie eingesetzt werden. Beim Tracing sendet das Gerät Nachrichten über eine Netzwerkverbindung zu einem Client. Diese Nachrichten, auch *Traces* genannt, können dann zur Fehlersuche genutzt werden. Außerdem lassen sich mithilfe der Traces die vielen **Programmabläufe besser nachvollziehen**. Zur Laufzeit wird sozusagen eine Historie über den Zustand des Geräts angelegt. Wie genau dieser äußere Zustand ist, hängt von der Anzahl und den Detailgrad der Traces ab.

Stürtzt das Gerät ab oder berechnet es falsche Werte, so kann nachträglich eine Fehleranalyse vorgenommen werden. Da die Nachrichten ohnehin übers Netzwerk ausgegeben werden ist sogar ein **Fernauslesen** denkbar. Das Tracing-Konzept soll dabei nicht an ein bestimmtes Netz gekoppelt sein, sondern kann über TCP/IP, als auch USB, Bluetooth oder Firewire erfolgen. Nachrichten sollen sich anhand ihrer Wichtigkeit unterscheiden lassen. Dies wird über **Trace-Level** erreicht, welche zusammen mit dem Trace übertragen werden.

Das Tracing-Konzept ist wie ein Client-Server-Modell aufgebaut. Es werden also mindestens zwei Programme benötigt. Auf dem Gerät läuft ein kleiner Server, welcher auf Verbindungen der Clients wartet und ihnen die Trace-Nachrichten sendet. Dieses Programm wird **Tracer** genannt. Auf dem Arbeitsplatz-Rechner läuft der Client namens **Trace-Monitor**. Dieser verbindet sich zum Gerät und empfängt die Trace-Nachrichten. Auf die Anzeige der Traces kann Einfluss genommen werden, indem der **Trace-Monitor** die Trace-Level auf dem Gerät neu setzt. Informationen werden so konzentriert oder reduziert.

In diesem Projekt soll ein **Tracing-Konzept**, unter enger Zusammenarbeit mit den beteiligten Entwicklern, erstellt werden. Daraufhin wird als praktische Arbeit der **Trace-Monitor** auf der PC-Seite entwickelt.

Ein weiteres Ziel ist die Integration des Trace-Monitors in die Programmkollektion der NPP-Geräteserie namens enSuite. Über enSuite lassen sich Geräte der Firma Elster und andere Geräte, welche das MMS-Protokoll beherrschen, auslesen und konfigurieren. Statt eine Standalone-Anwendung zu entwickeln, soll der Monitor als weiteres Modul eingebunden werden. Somit lassen sich auch bestehende Funktionalitäten und Ressourcen nutzen. Durch den Zugriff auf das Geräteobjekt vereinfacht sich der Verbindungsaufbau, da IP-Adresse und Gerätetyp bereits bekannt sind.

Kunden und Außendienstmitarbeiter benötigen keine extra Software um Traces vom Gerät zu empfangen, da enSuite bereits vorinstalliert ist. Durch die Aufnahme des Trace-Monitors in den Softwarelebenszyklus von enSuite, wird sichergestellt, dass das Programm regelmäßig gewartet und gepflegt wird.

3 Anforderungen

Für ein Tracing-System gibt es keine Standard-Software. Dies wäre auch schwer Umzusetzen, da die Anforderungen je nach Problemstellung sehr verschieden sein können.

Allerdings gibt es in der Fachliteratur mehrere Richtlinien, die zu einem erfolgreichen Konzept verhelfen. Auf diese allgemeinen Anforderungen soll zunächst eingegangen werden. Im Anschluss geht es um die speziellen Anforderungen für dieses Projekt.

3.1 Allgemeine Anforderungen an ein Tracing-Konzept

3.1.1 Zeitstempel

Zum Nachvollziehen des Programmablaufs, müssen die eingehenden Traces zeitlich geordnet werden. Dazu ist ein **hochauflösender Zeitstempel** erforderlich. Da in der Sekunde durchaus mehrere Nachrichten verarbeitet werden, muss der *Timestamp* Millisekunden unterstützen. Der klassiche Datentyp `time_t` der Programmiersprache C/C++ liefert aber nur die vergangenen Sekunden seit dem 01.01.1970. Um eine höhere Präzision zu erhalten, verwendet man deshalb die **Clocks** eines Geräts. Die Clock-Rate ist die Frequenz mit der eine CPU läuft. Mit einem Teiler kann man daraus einen für die Anwendung hinreichend genauen Wert(**Ticks**) ableiten. Bei einer 100 MHz CPU und einem Teiler von 256 erhält man so:
`100.000.000 : 256 = 390.625 TICKS_PRO_Sekunde .`

Es sind drei Schritte nötig um die Ticks des Systems für einen hochauflösenden Zeitstempel zu nutzen:

1. Das Gerät sendet zu Beginn die **aktuelle Uhrzeit** und die **vergangenen Ticks seit dem Systemstart** zum Client.
2. An jedem Trace werden die **aktuellen Ticks** angehängt
3. Aus der Differenz der Ticks im Verhältnis zu einer Milisekunde, kann dann ein sehr genauer **Offset**[1] berechnet werden, welcher die vergangenen Milisekunden seit Gerätstart wiedergibt. Durch Addition auf die Referenz-Uhrzeit erhält man so einen hochauflösenden Zeitstempel.

Eine Methode um aus den übermittelten Ticks den Zeitstempel zu erhalten, könnte so aussehen:

```
private long getTimestamp(long pTicks) {
    return (utc*1000 + (( pTicks - ticks) / TICKS_PER_MILISECOND ) );
}
```

3.1.2 Trace-Level

Trace-Level dienen zur **Priorisierung** von Informationen. Jeder Trace erhält einen Level zugewiesen, um die **Wichtigkeit** dieser Nachricht anzuzeigen. Je nach Implementierung unterscheidet sich die Anzahl der Level. Für dieses Projekt sind insgesamt sieben Level definiert worden. Tabelle 1 zeigt die Abkürzungen und die vorgesehene Verwendung für

[1] 1 tick = 2,56 μs

Nummer	Level	Bemerkung
1	EMERG	Schwerwiegender Systemfehler. Betrifft alle laufenden Prozesse und führt meist zum Absturz.
2	CRIT	Kritischer Systemfehler, welcher sofort behoben werden sollte, da sonst z.B. Datenverlust droht.
3	ERR	Fehler, welcher behoben werden sollte, aber nicht zum Absturz führt
4	WARN	Warnung, dass ein Fehler auftreten könnte, wenn Programm fortgeführt wird
5	NOTICE	Ungewöhnliche Ereignisse, welche nicht zum Fehler führen und kein Eingreifen erfordert
6	INFO	Normale Nachricht. Ausgabe von gemessenen Werten und Ereignissen
7	DEBUG	Ausgaben für Entwickler zum Debuggen. Low-Level-Meldungen wie Aufruf und Rückkehr von System-Calls.

Tabelle 1: Trace-Level

jeden Level.
Höhere Trace-Level schließen alle unteren Level mit ein, d.h. Fehler werden auf dem Level 4 ebenfalls mit ausgegeben. Auf dem höchsten Level 7 werden alle möglichen.[2] Traces ausgegeben. Von da ist nur eine **Reduzierung** der Informationen möglich. So ein additives System ist schon sinnvoll, da der Entwickler beim Debugging ja zusätzlich auch alle Fehler, Warnungen und Infos sehen will. Eine Erhöhung des Levels ist gleichbedeutend mit der **Konzentration** von Informationen.

3.1.3 Einfache Anwendung

Um Traces auszugeben, benutzen die Entwickler Makro-Funktionen im Programm-Code. Diese besitzen die gleichen Namen wie die Trace-Level in der Tabelle. Statt `printf()` werden `emerg()`, `crit()`, `err()`, ... und `debug()` verwendet. Die Verwendung ist dabei die gleiche wie bei der `printf`-Funktion, sodass in dem Formatierungsstring auch alle gängigen Datentypen wie `char, short int, int, long int, float, double und string` benutzt werden könnnen.

Trace Level werden statisch zur Kompilierungszeit auf Datei-Ebene gesetzt. Ein einfaches `define` im Quellcode regelt, ob überhaupt bei diesem Level etwas ausgegeben werden soll. Zur Laufzeit ist es somit nicht möglich einen Level einzustellen, welcher unter dem statischen liegt. Der Standard Level für die meisten Module ist 4, sodass nur Warnungen und Meldungen darüber ausgegeben werden. In dem Trace-Monitor lassen

[2] wörtlich gemeint

sich die Level **dynamsich zur Laufzeit** auf Modul- und Datei-Ebene einstellen. Wird der Level eines Moduls gändert, betrifft dies eben nur die zugehörigen Dateien.

3.2 Spezielle Anforderungen an dieses Tracing-Konzept

Die **Formatierungsstrings** werden **nicht auf dem Gerät gespeichert**. Dieses Vorgehen bringt mehrere Vorteile mit sich. Zum einen müssen die Nachrichten nicht mehr auf dem Gerät formatiert werden, wordurch **Zeit und Rechenleistung** eingespart wird. Bei den vielen Ausgaben fällt somit die `printf`-Funktion nicht mehr zur Last. Für einen Desktop-Rechner stellt dies natürlich kein Problem dar. Zum anderen ist das Versenden langer Debugging-Nachrichten über das Netzwerk nicht performant. Will man Geräte aus der Ferne auslesen muss man bedenken, dass es nicht überall schnelle Anbindungen gibt. Viele eingebette Geräte sind auch heute noch über langsame Modems mit dem Netz verbunden. Neben der Rechenleistung ist der Speicherplatz eine knappe Ressource, welcher durch die Auslagerung der Strings geschont wird.

Die Lösung des Problems besteht darin, die Formatierungsstrings in **Trace-Dateien** auszulagern. Trace-Dateien sind nur auf dem lokalen Rechner des Entwicklers gespeichert. Das Gerät überträgt nur die variable Parameterliste und die Referenz auf die ursprüngliche Nachricht. Handelt es sich um einen statischen Text ist die Parameterliste leer. **Erst auf der Empfangsseite wird die Nachricht zusammengebaut**, indem der Trace-Monitor die Trace-Dateien einliest und den Formatierungsstring mit den Parametern verbindet. Jede Nachricht beinhaltet außerdem eine **Zeilennummer**, damit der Entwickler über den Trace-Monitor direkt zur passenden Stelle im Code springen kann.

Auf dem Gerät kann ein Modul mehrmals gestartet werden. Will man die laufenden **Instanzen unterscheiden können**, sollte man nicht den Modulnamen heranziehen, da dieser bei allen Instanzen gleich ist. Jede Instanz erhält deshalb eine eigene, **eindeutige Modul-ID**.

3.3 Tracing-Monitor

Nach dem **Verbindungsaufbau** muss in regelmäßigen Abständen ein Lebenszeichen zum Gerät gesendet werden, damit die Verbindung nicht unterbrochen wird. Ein Watchdog auf dem Gerät muss innerhalb eines bestimmten Intervalls dieses *Idle*-Signal erhalten. Anderfalls trennt er die Verbindung zu dem Client.

Unmittelbar nach dem Verbindungsaufbau muss der Trace-Monitor eine **Modulliste anfordern**. Diese Liste beinhaltet alle zur Zeit laufenden Module auf dem Gerät. Befindet sich das Gerät noch im Hochlauf, können durchaus noch Module hinzukommen. Diese werden dann automatisch nachgeschickt, sodass sich der Trace-Monitor nicht darum zu kümmern braucht.

Außerdem soll der **Trace-Level** einer Datei abgefragt und gesetzt werden können. Dies ist eine wichtige Aufgabe, da sich so die Informationsmenge einstellen lässt. Statt einzelne Dateien abzufragen, soll dies auch für ein ganzes Modul funktionieren. Da ein Modul selbst keinen Trace-Level besitzt, werden alle Level der zum Modul gehörigen Dateien übertragen bzw. gesetzt.

Timestamp	Modul-ID	File	Line	Trace-Level	Trace-Nachricht

Tabelle 2: Notwendige Spalten in der Trace-Tabelle

Die wichtigste Aufgabe ist wohl der **Empfang und die Darstellung von Trace-Nachrichten**. Dafür sind zwar viele Prozesse im Hintergrund notwendig, der Benutzer sieht allerdings nur diesen Bereich. Deshalb muss diese Hauptfunktionen auch stabil und zuverlässig funktionieren. Die Oberfläche soll schnell reagieren und die Traces je nach Wichtigkeit visuell hervorheben. Dabei sollen möglichst schwache Hintergrundfarben verwendet werden. Als Vorgabe soll das Netzwerk-Analyseprogramm *WireShark* dienen.

Bei der Anzeige der Trace-Nachrichten sollen nur so viele Informationen wie nötig angezeigt werden. Tabelle 2 dient als Vorlage für eine sinnvolle Spaltenbelegung.

Die **Anzeige** von Traces muss sich **stoppen lassen**, damit sich der Entwickler ein bestimmtes Ereignis genauer ansehen kann. Im Hintergrund werden jedoch weiterhin alle Traces gespeichert. Beim Aktualisieren der Anzeige erscheinen dann unter Umständen sehr viele neue Einträge in der Tabelle.

Weiterhin muss eine **Filterung** aller gespeicherten Traces möglich sein. Dadurch lassen sich z.B. nur Nachrichten einer bestimmtes Programmzeile anzeigen. Über einen Dialog sollen sich mehrere Filterkriterien speichern lassen. Dem Entwickler steht zur Wahl ob er die Bedingungen miteinander ODER- oder UND-verknüpfen will. Gesetzte Filter sollen auch für eingehende Traces gelten. Trifft keine Bedingung auf eine neue Nachricht zu, so wird diese auch nicht angezeigt.

Über die **Zeilennummer** in einer Trace-Nachricht soll es möglich sein, direkt zu der passenden Stelle im Programmcode zu „springen“. Der Editor soll dabei frei konfigurierbar sein.

Da das Programm enSuite in Java geschrieben ist und der Trace-Monitor integriert werden soll, muss als Programmiersprache ebenfalls Java verwendet werden. Bereits vorhandene Bibliotheken sollen so gut es geht genutzt werden. Das bestehende Datenbanksystem kann und soll zur Speicherung der Traces mitbenutzt werden.

4 Konzept

Nachfolgend soll ein Konzept entwickelt werden, anhand dessen sich das Tracing-System später umsetzen lässt. Die Zusammenspiel der Trace-Tools und die Abläufe beim Tracing sollen zunächst festgelegt werden um dann im Anschluss ein geeignetes Netzwerk-Protokoll für die Kommunikation erarbeiten zu können. Außerdem wird für die zusammenhängende Speicherung von Daten ein Datenmodell benötigt. Aus dem Datenmodell lässt sich dann später die Datenbank entwerfen.

4.1 Trace-System

Bild 1 zeigt das Deployment-Diagramm des Trace-Systems, welches eine besser Übersicht verschaffen soll. Eingezeichnet sind die physikalischen Geräte und die darauf laufenden

Softwarekomponenten. Auf dem lokalen Rechner ist zum Übersetzen der Quellcodes die MULTI-IDE der Firma GREEN HILLS SOFTWARE installiert. Um das Gerät zu konfigurieren ist die Programmkollektion ENSUITE installiert, welche durch den Trace-Monitor dazu befähigt wird Traces zu empfangen. Auf dem Gerät wartet der Tracer auf eingehende Verbindungen und puffert die Nachrichten der Module solange noch kein Client verbunden ist.

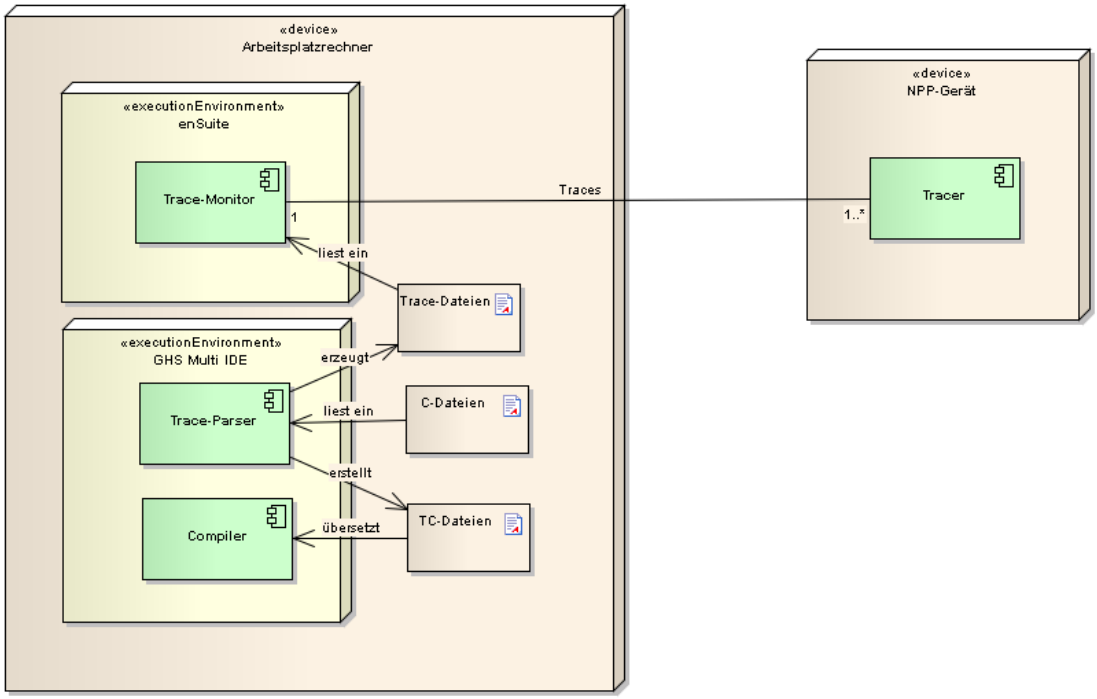


Abbildung 1: Trace Systems

Um eine Gesamtübersicht über den genauen Ablauf der Prozesse zu geben, wird folgendes, alltägliches Szenario angenommen: Ein Entwickler hat Änderungen an einem Modul vorgenommen und will es nun zur Laufzeit auf Fehler überprüfen. In der Multi-IDE startet er dazu zunächst den Buildprozess. Dadurch wird aber nicht sofort der Compiler aufgerufen sondern der **Trace-Parser**. Jedes Verhalten der IDE lässt sich durch Scripte steuern. In diesem Fall wurde der Build-Vorgang leicht modifiziert um den Trace-Parser aufzurufen. Der Trace-Parser liest daraufhin die C- und C++-Dateien ein und sucht nach Debugging-Ausgaben. Die Formatierungsstrings werden daraufhin in Trace-Dateien ausgelagert und an der Stelle durch eine Referenz auf die urprüngliche Nachricht ersetzt. Anstatt die Änderung in der Original-Datei zu speichern, wird eine temporäre Datei zum Übersetzen angelegt. Diese TC-Dateien werden dann dem Compiler übergeben, welcher daraufhin ein Image erzeugt.

Das Image wird auf eine SD-Karte überspielt und anschließend auf das Board des Gerätes gesteckt um davon zu Booten. Während des Boot-Vorgangs kann bereits der Trace-Monitor gestartet werden, da das Betriebssystem diese Task zu erst startet. Wurde

ein Modul erfolgreich geladen, sendet der Tracer die Modul-Informationen zum Monitor. Über den Modul-Namen kann dieser nun die passende Trace-Datei finden und parsen. Die Inhalte werden in einer Datenbank gespeichert. Der Trace-Monitor kennt nun alle zum Modul dazugehörigen Dateien und dessen Nachrichten. Sendet das neu übersetzte Modul nun einen Trace, setzt der Trace-Monitor die Nachricht auf dem PC zusammen und zeigt diese in einer Tabelle an.

Der Entwickler erkennt anhand der Nachrichten, ob das Modul korrekt arbeitet. Auch das Fehlen eines Traces kann schon viel Aussagen. Wird ein Fehler erkannt, kann der Entwickler direkt über den Trace-Monitor in die passende Zeile springen. Ist der Fehler ausfindig gemacht und behoben, kann das System erneut übersetzt werden und die Prozedur beginnt von neuem.

4.2 Trace-Dateien

Trace-Dateien speichern die Formatierungsstrings und lösen somit das Performanceproblem auf den Geräten. Die Struktur der Dateien muss ein einfaches Wiederauffinden der Formatierungsstrings ermöglichen. Nur anhand der Modul-, File- und Trace-ID muss der Formatierungsstring gefunden werden.

Pro Modul wird eine **Trace-Datei** angelegt, da ein Modul die höchste Instanz ist, welche geloggt werden kann. Abb. 2 veranschaulicht die Relation zwischen Modul, Dateien und Traces. Da zwei Beziehungen vorhanden sind, müssen auch zwei Informationen in der Trace-Datei gespeichert werden. Einmal die Information, welche Dateien das Modul besitzt (**Trace-File-Information**) und dann, welche Traces eine bestimmte Datei besitzt (**Trace-Data-Information**).

Abbildung 2: Relationen der Trace-Daten

Der Aufbau soll Anhand des Moduls *FlowConv* veranschaulicht werden. Der Dateiname, in diesem Fall `FlowConv-Release_02-02-B.trace`, setzt sich aus dem Modulnamen und der Versionsnummer zusammen. Das Gerät übermittelt diese beiden Informationen, sodass der Trace-Monitor nurnoch den Dateinamen richtig zusammenbauen muss. In der ersten Zeile der Auflisting 1 wird der indexierte Pfad zu einer Datei gespeichert. Die ID dient den folgenden Zeilen zur Verknüpfung der Traces mit der Datei. Der Trace-Nachricht bekommt so ihren Ursprung zugewiesen. Zeile 1 ist demnach eine Trace-File-Information. Die Traces werden ebenfalls indexiert, zu sehen in der zweiten Spalte.

```
'''
Aufbau einer Trace File Information
<File Index>: <File Name>

Aufbau einer Trace Data Information:
<File Index>. <Trace Index>. <Line>. <Level>: <Formatstring>
'''
1: src/Calculate/Calculate_Vm.c # File Info
1. 1. 69. 7: %lld 0x%08x %f %f  # Data Info
1. 2. 71. 7: FlowConv-Q %f
2: src/Calculate/Calculate_Vc.c
2. 3. 240. 7: Calculate_Vc
3: src/Calculate/Calculate_VbEM.c
3. 3. 108. 7: Calculate_VbEM
4: src/Calculate/Calculate_Cnt.c
4. 5. 63. 3: Wrong switch %d
4. 6. 84. 3: Wrong switch %d
4. 7. 104. 7: Calculate_Cnt
5: src/Calculate/Calculate.c
5. 8. 29. 7: Parameter changed
6: src/FlowConv.c
6. 9. 73. 7: %d %d'
6. 10. 92. 7: Freeze old %d
6. 11. 108. 7: %d %d
6. 12. 227. 7: InitializeDOs in FlowConv
6. 13. 230. 1: (%d) %s
6. 14. 235. 3: %s: no Tree available
```

Listing 1: FlowConv-Release_02-02-B.**trace**

Als Trennzeichen wurde für die Indizes ein Punkt und für den Formatierungsstring ein Doppelpunk verwendet. Beim extrahieren des Formatierungsstring muss darauf acht gegeben werden, dass dieser auch oben genannte Trennzeichen beinhalten kann. In Java sollte die Aufteilung des Strings durch einen Parameter limitiert werden:
`String[] idsAndFormatString = line.split(":", 2)`

4.3 Trace-Parser

Der Trace-Parser lagert Formatierungsstring in Trace-Dateien aus und wird kurz vor dem Übersetzungslauf aufgerufen. Um den Entwickler dabei nicht zu belasten, wurde das Tool in die Multi-IDE eingebaut. Beim Einlesen einer Quellcode-Datei sucht der Parser bestimmte Token, nämlich genau die Namen der sieben Makro-Funktionen: `emerg()`, `crit()`, `err()`, `warn()`, `notice()`, `info()`und `debug()`. Bei einem Fund werden Parameter innerhalb der Klammern ersetzt. Wieder soll ein Auszug aus dem Modul *FlowConv*

als Beispiel dienen.

```
1  if ( (pTheFbDoTree = FbDo_GetTree ( pmyInfo )) == NULL )
2  {
3    err("%s: no Tree available", pmyInfo->m_Descr.m_AsName);
4    return Failure;
5  }
```

Listing 2: Auszug aus FlowConv.**c**

```
1  if ( (pTheFbDoTree = FbDo_GetTree ( pmyInfo )) == NULL )
2  {
3    err(0x01400CC7, 0x00000004, pmyInfo->m_Descr.m_AsName);
4    return Failure;
5  }
```

Listing 3: Auszug aus FlowConv.**tc**

In Zeile 3 hat der Trace-Parser den Funktionsaufruf angepasst und den Formatierungsstring ausgelagert. Der erste Parameter der neuen Makro-Funktion ist vom Typ Integer und speichert **File-ID, Trace-ID, Zeilennummer und Trace-Level**. Für die File-ID werden 9 Bits aufgebracht, sodass pro Modul 512 Dateien gespeichert werden können. Die Zahl ist für die relativen kleinen Modul recht hoch, allerdings besitzt der Kernel knapp 500 Dateien. Die nächsten 12 Bits werden für die Trace-ID benötigt, sodass eine Datei 4096 Trace-Nachrichten beinhalten kann. Für die Kodierung der Trace-Level sind 3 Bits notwendig.

Der zweite Parameter, ebenfalls vom Typ Integer, wird dazu benutzt eine **Typenliste** zu übermitteln. Der Trace-Monitor verwendet die Liste um aus dem Datenfeld später die richtigen Werte herauszulesen. Das Datenfeld ist von außen betrachtet bloß ein BLOB (Binary Large Object) in dem alle Parameter hintereinander stehen. Diese Liste stellt somit die nötigen Informationen bereit, die einzelnen Parameter wieder auszupacken. Insgesamt werden sechs Datentypen unterstüzt, sodass 3 Bit für die Kodierung verwendet werden müssen. Da die Größe auf 32-Bit beschränkt ist, können demnach maximal 10 Parameter eingesetzt werden. Den elften Parameter würde der Trace-Parser einfach verwerfen. In dem Trace-Monitor führt dieser Umstand dann zu einem Formatierungs-Fehler während des Zusammenbauens.

4.4 Datenmodell

Alle Traces sollen in einer **Datenbank** gespeichert werden, da das bloße Speichern in einer GUI-Komponente(z.B. JTable) weder **performant** noch anpassungsfähig ist. Über die *Structured Query Language* (SQL) lassen sich komplexe Abfragen tätigen, sodass der Benutzer nur gewünschte Informationen angezeigt bekommt. Zur Anzeige der Traces müssen diese vorher über das Datenbanksystem abgefragt werden. Zur Installation eines

Wert	Datentyp	Beschreibung	Größe
0	`none`	keine Parameter	
1	`string`	Terminierter C-ASCII-String	(strlen+1) Byte
2	`double`	64-Bit Fließkommazahl (IEEE 754)	8 Byte
3	`long int`	64-Bit Integer-Zahl	8 Byte
4	`int`	32-Bit Integer-Zahl	4 Byte
5	`short int`	16-Bit Integer Zahl	2 Byte
6	`char|byte`	8-Bit Integer-Zahl	1 Byte

Tabelle 3: Kodierung der unterstützten Datentypen

Filters fügt man zu dieser Abfrage weitere Bedingungen hinzu. Vorerst ist keine dauerhafte Speicherung der Traces geplant. Um die Performance noch ein bisschen zu steigern, wird die Datenbank deshalb komplett im Speicher gehalten.

Das Datenbank-Schema auf Bild 3 umfasst insgesamt fünf Tabellen. Da eine n:m Beziehung zwischen Modulen und Dateien vorliegt, befindet sich zwischen ihnen eine Role-Table. Die Tabelle **MODULE_TRACE** dient zum Speichern der Traces und befindet sich im Knotenpunkt der anderen Tabellen.

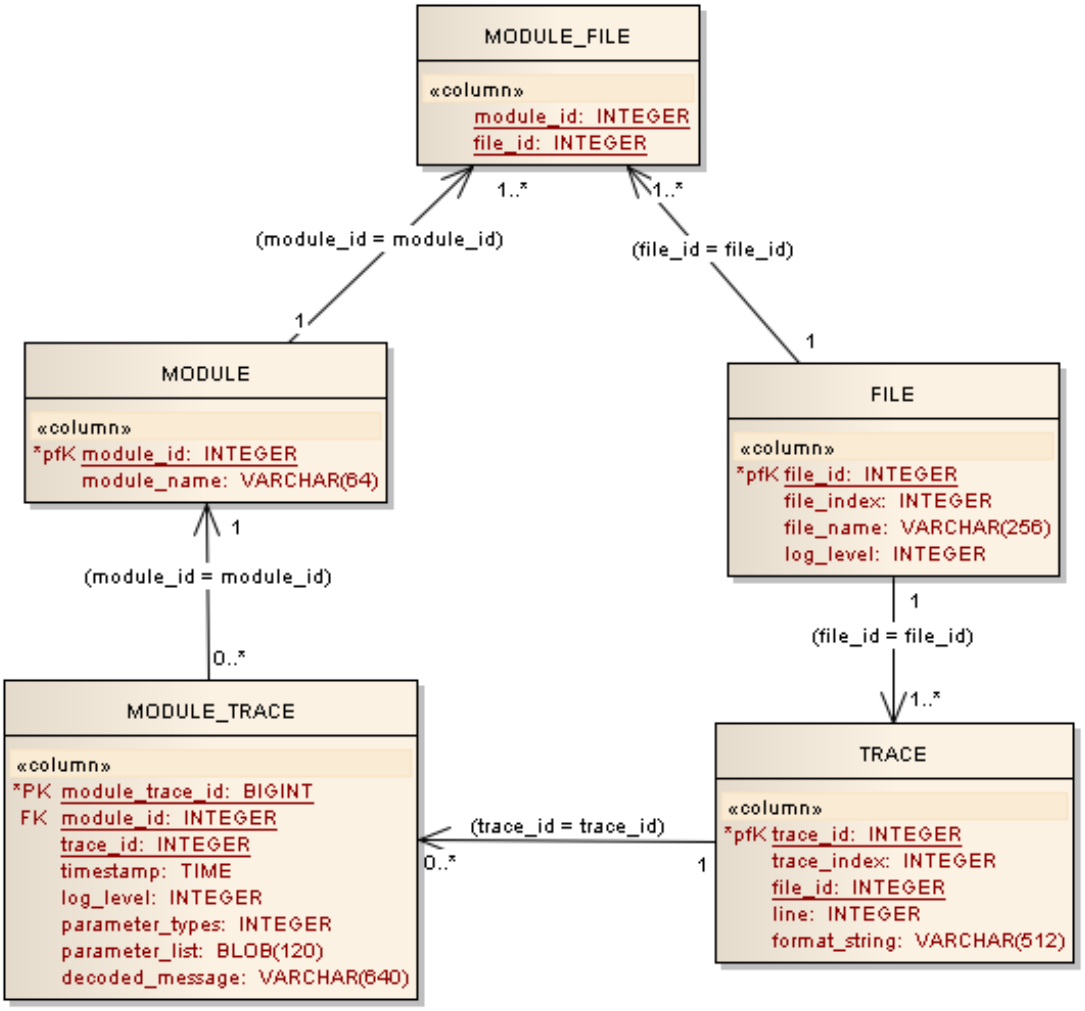


Abbildung 3: Datenbank-Schema

4.5 Trace Monitor Protokoll

Zum Auslesen von Trace-Nachrichten aus einem NPP-Gerät wird ein Protokoll auf Anwendungsebene benötigt. Im TCP/IP-Netzwerk wird hierzu eine TCP-Verbindung (transmission control protocol, RFC 793 & RFC 1323) auf Port 19790 benutzt. Da die Portnummer über 1024 und unter 49151 liegt, zählt dieser zum Bereich der User Ports dazu. User Ports können ohne Zustimmung der IETF frei gewählt werden. Bevor Traces ausgelesen werden können, hat das Protokoll viele kleine Aufgaben zu erfüllen. Nach dem **Verbindungsaufbau** muss es in bestimmten Abständen ein **Lebenszeichen zum Gerät senden**, damit der Watchdog auf dem Gerät die Verbindung nicht trennt. Außerdem wird über das Protokoll auch die **Zeit sychronisiert**. Der Client fordert in erster Linie Informationen an, welche dann vom Server geliefert werden. Kurz nach dem Verbindungs-

aufbau fordert der Client eine Modulliste an und wenig später auch die Trace-Level der Dateien. Eine Ausnahme stellen die Traces selbst dar, welche ohne Aufforderung vom Gerät gesendet werden.

Das **Trace Monitor Protokoll (TMP)** wird als zustandsloses Protokoll entworfen. Beide Kommunikationspartner können jederzeit beliebige Daten senden. Automatenfehler können nicht auftreten. Diesen Vorteil erkauft man sich durch einen kleinen Mehraufwand bei der Auswertungs der Token. Es ist Aufgabe des Empfängers, die Daten herauszufiltern, an denen er interessiert ist.

Daten werden in einem festen Format übertragen (Tab. 4). So wird sichergestellt, dass der Kommunikationspartner die Daten versteht und sich nicht im Strom verliert. Würde ein Partner beispielsweise mehr Daten versenden als von der Gegenstelle erwartet wird, so kommt die ganze Kommunikation aus dem Takt.

Header		**Payload**
Token 1 Byte	Länge 1 Byte	Datenfeld x Bytes

Tabelle 4: Trace Format

Damit der Empfänger sofort weiß, ob eine Nachricht relevant oder irrelevant ist, dient das erste Byte, genannt Token, zur Identifikation der gesamten Trace-Nachricht. Auch die Länge der folgenden Daten wird immer mit übertragen, selbst wenn es keine Daten gibt. In diesem Fall wird einfach die Länge 0 übertragen. So kann der Trace-Monitor nach dem Lesen der ersten 2 Bytes bestimmen, ob die nachfolgenden Bytes übersprungen werden können oder nicht. Beim Übersprüngen werden die Daten gelesen aber nicht gespeichert. Passende Funktionen gibt in der Socket-Klasse.

Die Tabelle 5 zeigt die sieben verschiedenen Token und wie sie kodiert werden. Aus Platzgründen steht in der Spalte der hexadezimale Wert des Bytes. In dem Token gibt das dritte Bit die Richtung an, also ob die Nachricht von dem Gerät oder von dem PC geschickt wird. Deshalb springt die erste Ziffer der Id ständig von 4 auf 6. Das Richtungs-Bit befindet sich an dieser Position um die ID auch als passenden ASCII Buchstaben schreiben zu können. Über Telnet lässt sich so auch ohne Client mit dem Gerät, über die Tastatur, kommunizieren. Ein Großbuchstabe steht so für ein Client-Kommando und ein Kleinbuchstabe für die Antwort des Servers. Hierzu ein kleines Beispiel anhand des Idle-Tokens mit der ID 0x49 bzw 0x69:

0	1	0	0	1	0	0	1	Trace-Monitor sendet Idle-Token, Direction-Bit auf 1
0	0	0	0	1	0	0	1	Gerät bestätigt Idle-Token, Direction-Bit auf 0

Token	Id	Länge	Daten
VersionInfo	0x43 ('C')	01	Protokoll Version
TimeSync	0x63 ('c')	0A	UTC und CPU-Ticks
Idle	0x49 ('I')	00	
	0x69 ('i')	00	
TraceMessage	0x74 ('t')	xx	Trace-Nachricht
ModulInfo	0x4D ('M')	00	
	0x4D ('M')	01	Modul-ID
	0x6D ('m')	xx	Modul-ID + Name inkl. Version
LevelInfo	0x4C ('L')	00	
	0x4C ('L')	01	Modul-ID
	0x4C ('L')	02	File-ID
	0x4C ('L')	03	File-ID + Trace-Level
	0x6C ('l')	03	File-ID + Trace-Level
	0x6C ('l')	xx	Modul-ID + Trace-Levels
Disconnect	0x44 ('D')	00	
	0x64 ('d')	00	

Tabelle 5: Token IDs

Es sollen nicht auf alle Datenfelder eingegangen werden, da diese zum größten Teil selbsterklärend sind. Das Datenfeld der Trace-Nachricht ist allerdings komplexer und soll nun beschrieben werden.

Feld	Bits	Name	Beschreibung
1	48	Timestamp	Beinhaltet die vergangenen Ticks seit Gerätestart
2	32	Unique IDs	Module-,Trace-,File-Index, Zeilennr., Trace-Level
3	32	Argumenten Liste	Speichert bis zu 10 Datentypen (s.Tab. 3)
4	xx	1. Argument	siehe oben
5	xx	2. Argument	
:	xx	:	
13	xx	10. Argument	

Tabelle 6: Format einer Trace-Nachricht

5 Durchführung

Als praktische Arbeit soll der Trace-Monitor entwickelt werden. Da das erlernte Wissen in der IT-Branche schnell veraltet ist, sollen bei der Umsetzung aktuelle Programmiertechniken angewendet werden. Dadurch kann sichergestellt werden, dass die Anwendung auch noch Jahre später ohne größere Anpassungen funktioniert. Die Standard Edition

der Java Plattform (Java SE) bringt schon viele Features im Bereich der Netzwerk-, Web- und Datenbankprogrammierung mit. Seit dem Kauf von Oracle im Jahre 2009 sind viele Neuheiten in Java hineingeflossen und zum Standard geworden. Sehr gute Neuigkeiten für Entwickler, die vorher ein Risiko mit *3rd party software* eingehen mussten. Durch Aufnahme neuer Funktionen bietet Oracle den Java-Kunden Sicherheit, da die Schnittstellen oft über Jahre hinweg stabil bleiben.

Ein neues Feature ist die *Java Persistence API (JPA)*, welche im Trace-Monitor die Speicherung der Traces ermöglichen soll. Deshalb wird in diesem Kapitel zunächst etwas Grundlagenforschung betrieben, bevor es an die konkrete Umsetzung geht.

5.1 Einführung in die Java Persistence API

Für dieses Projekt soll die *Java Persistence API (JPA)* verwendet werden, welche seit Java SE 5 fester Bestandteil dieser Plattform ist. Durch JPA wird das Problem der **objektrelationalen Abbildung** gelöst. Tabellen einer Datenbank und Laufzeit-Objekte in Java besitzen eine starke Ähnlichkeit: beides sind Container für Attributwerte. Eine **automatische Überführung ineinander** liegt also nahe. „The domain model has a class. The database has a table. They look pretty similar. It should be simply to convert one to the other automatically.“[3] In der Tat lassen sich die Attribute einer Klasse sehr gut auf Spalten einer Tabelle abbilden. Die Technik bei der ein Objekt automatisch auf eine Relation[4] überführt wird, nennt sich *Object-Relation Mapping* oder einfach ORM. Programmierern wird dabei viel Arbeit abgenommen: um die Ergebnisse einer SQL-Abfrage zurück in etwas Objekt-Orientiertem zu konvertieren bedarf es keiner *Data Access Objects (DAO)* mehr. In JPA lieft eine Abfrage über JDBC nun direkt ein Objekt zurück.

Die JPA-Entwickler haben ein Manifesto veröffentlich, wie ein ideales ORM-Framework aussehen sollte:

- Objekte statt Tabellen. Anwendungen sollen nach dem Domain Modell entwickelt werden. Abfragen werden deshalb nicht mehr in eine relationale- sondern in eine objektorierntiere Sprache ausgedrückt.
- Einfachheit statt Ignoranz. ORM verhindert keine Mapping Fehler. Es soll Entwicklern dienen, die sich mit relationalen Datenbanken auskennen, aber nicht tausende Zeilen Code für ein bereits gelöstes Problem schreiben wollen.
- Unaufdringlichkeit statt Transparenz. Die Anwendung muss jederzeit die Kontrolle über die zu speichernden Daten haben, weshalb die Persistenz-Lösung nicht vollständig transparent sein kann. Allerdings lässt sie sich so gestalten, dass das Domänen-Modell nicht „gestört“ wird. Es sollen keine Interfaces implementiert oder Klassen vererbt werden um Objekte persistent zu machen.
- Aus Altdaten neue Objekte erzeugen. Die meisten Applikationen müssen ein bestehendes relationales Datenbank Schema verwenden, anstatt ein neues zu entwerfen. Deshalb ist die Unterstützung für Altedaten der wichtigste Anwendungsfall. Ein Datenbankschema kann durchaus die Lebenszeit der Entwickler überdauern.

[3] [MS09, S. 2]
[4] Tabelle einer Datenbank

- Wenig Overhead. Entwickler haben Probleme zu lösen und benötigen dazu angemessene Features. Es sollen ihnen ein leichtgewichtiges Persistenz Modell angeboten werden, welches die Standard-Aufgaben performant erledigt und minimale Zeilen an Code benötigt. Für Spezialfälle gilt *Configuration by Exception*.
- Mobilität. Die Applikation entscheidet wo die Daten gespeichert werden. In einer verteilten Anwendung werden persistente Objekte automatisch übertragen.

Die Referenzimplementierung für JPA heißt **EclipseLink** und ist aus dem Projekt *TopLink* der Firma Oracle heraus gewachsen.

5.1.1 Entity Übersicht

In JPA bezeichnet man ein **persistierbares Objekt** als *Entity*. Das bedeutet, der Zustand einer Entity *kann* über die Lebensdauer des erzeugenden Prozesses hinaus gespeichert und wiederhergestellt werden. Entities werden nicht automatisch gespeichert sondern befinden sich wie alle anderen Objekte im Speicher. Den Zeitpunkt der Speicherung bestimmt die Anwendung selbst. Erst über einen API-Aufruf wird eine Entity persistiert.

Neben der einzigartigen *object identity*, besitzt dieses spezielle Objekt auch eine *persistent identity*. Dadruch ist es möglich eine bestimmte Zeile einer Tabelle zu identifizieren. Es handelt sich bei der *persistent identity* also um den Primärschlüssel.

Entities sind **quasi-transaktional**. Obwohl Entitys zu jeder Zeit erzeugt, bearbeitet und gelöscht werden können, müssen diese Operationen eigentlich in Transaktionen ausgeführt werden. Am Ende einer Transaktion erhält man einen eindeutigen Zustand. Entweder hat die Änderung an der Datenbank Erfolg gehabt oder ist Fehlgeschlagen. Wird eine Entity im Speicher geändert und nicht direkt persistiert, so entsteht in der Zwischenzeit ein undefinierter, inkonsistenter Zustand. Dieser kann später im Programm zu einem *Rollback* in der Datenbank führen.

Entities sind ganz normale (plain old) Java Objekte (POJOs) mit **speziellen Annotationen**. Annotationen beginnen in Java mit einem `@` und dienen dazu Metadaten einzubinden. In diesem Fall beschreiben die Metadaten die Tabelle einer Datenbank und wie die Klasse darauf abzubilden ist.

5.1.2 Aufbau

Damit eine Entity fest in der Datenbank gespeichert wird, muss ein API-Aufruf gemacht werden. Für die verschiedenen Operationen auf die Entities sind sogar viele verschiedene API-Calls notwendig. Die gesamte API wird von dem *EntityManager* implementiert und gekapselt. Er übernimmt die Lese- und Schreibvorgänge und sorgt dafür das aus gewöhnlichen Java-Objekten persistierbare Einheiten werden.

Übergibt man dem *EntityManager* eine Referenz auf ein persistierbares Objekt, so wird dieses zu einer *managed Entity*. Aus der Sammlung aller kontrollierten Entities ergibt sich der *PersistenceContext*. Dieser Kontext lässt sich nicht direkt, sondern nur über den Manager, ändern. Besser ist es, diesen als Zustand zu betrachten. Eine Entity kann nur einmal im *PersistenceContext* vorhanden sein, da alle Elemente einzigartig sein

müssen. Der *EntityManager* selbst, wird aus der Fabrik *EntityManagerFactory* erzeugt. Diese belegt den Manager mit sinnvollen Voreinstellung und dient somit als Template. Voreinstellungen werden in einer separaten *PersistenceUnit* gespeichert und der Fabrik übergeben. Zwischen den Klassen besteht deshalb eine 1:1-Beziehung, da die Fabrik nur mit einer einzigen Unit konfiguriert werden kann. Betrachtet man Abb. 4, so kommt es sehr überraschend, dass mehrere *EntityManager* auf den gleichen *PersistenceContext* verweisen können.

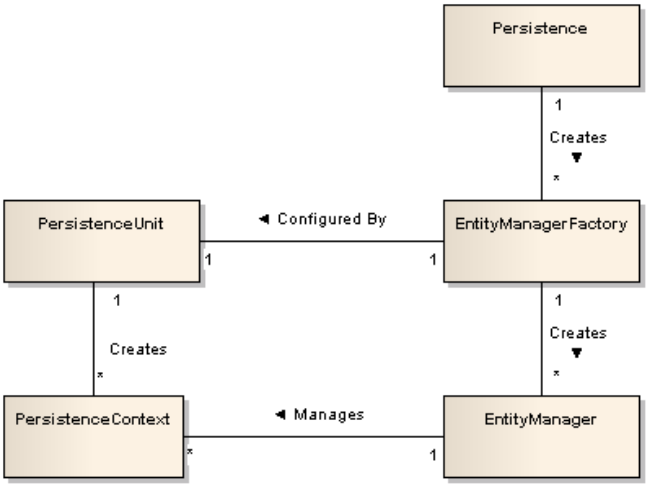


Abbildung 4: Beziehungen zwischen den JPA-Konzepten

5.1.3 Verwendung

Ein *EntityManager* wird sets von der *EntityManagerFactory* geholt. Je nach Name der Fabrik erhält der Manager verschiedene Konfigurations-Parameter, wie z.B. Verbindungsdaten und Datenbank-Login.

```
EntityManagerFactory emf =
        Persistence.createEntityManagerFactory("TraceMonitorConfiguration");
EntityManager em = emf.createEntityManager();
```

Nachdem die Fabrik verfügbar ist, kann der Manager geholt werden. Bereits an dieser Stelle lassen sich schon *Entities* erstellen und persistieren. Als Beispiel soll ein Code-Schnipsel aus dem Trace-Monitor dienen. Er zeigt die Speicherung eines Moduls, nachdem das Gerät den Namen und die ID bekannt gegeben hat.

```
newModule = new Module(); // Entity
newModule.setModuleId(moduleId);
newModule.setName(moduleName);
em.persist(newModule); // write to database
```

Die Entity wird von nun an von dem *EntityManager* auf Änderungen kontrolliert. Wird beispielsweise nach dem letzten Aufruf erneut eine Setter-Methode aufgerufen, werden

die neuen Werte automatisch in die Datenbank geschrieben. Je nach Einstellung der *EntityManagerFactory* schreibt der letzte Aufruf die Daten nicht zwangsläufig sofort in die Datenbank. Standardmäßig arbeitet das DBM-System in dem Modus *lazyFetching*.Daten werden erst geschrieben, wenn eine Transaktion ansteht oder eine Abfrage auf diese Daten von einer anderen Stelle im Code gemacht wird. Die Performance des Systems wird verbessert, da nicht jedes Objekt einzeln geschrieben werden muss.

Soll eine Entity nicht mehr vom *EntityManager* verwaltet werden, so lässt sich diese entkoppeln. Änderungen werden dann nicht mehr mit der Datenbank abgeglichen. Die letzten Änderungen werden nicht synchronisiert.

```
em.detach(newModule); // stop synchronization with db
em.remove(anotherModule); // delete record from db
```

Nicht immer kann ein gesetzter Wert auch in die Datenbank gespeichert werden. Ist die Modul-ID bereits vergeben wird eine *RollbackException* geworfen, da es keine doppelten Primärschlüssel geben darf. Auch kann der Modulname zu lang für das Datenbankfeld sein. Damit solche Fehler direkt abgefangen werden können und nicht erst bei einer Abfrage woanders im Code, muss eine **explizite Transaktion** durchgeführt werden. Dazu muss eine Transaktion geöffnet- und mit einem *commit* beendet werden.

```
try
{
        em.getTransaction().begin();
                em.persist(newModule)
        em.getTransaction().commit();
}
catch( RollbackException e )
{
    LOGGER.log( Level.SEVERE, "Module ID violates primary key constraint or
    module name is too long.");
}
```

Zur Abfrage von Datenbank-Einträgen (*records*) wird ebenfalls der *EntityManager* bemächtigt. Der untere Code-Schnipsel aus dem Trace-Monitor Programm verhindert, dass ein Modul nicht ein zweites Mal registriert wird. Eigentlich sollte das Gerät kein zweites Mal dieses Token senden, aber da es sich um ein zustandsloses Protokoll handelt, kann dieser Fall durchaus eintreten.

```
// Check if moduleId already exists in database
Module moduleMatch = null;
try {
    moduleMatch = em.createNamedQuery("Module.findByModuleId", Module.class)
                                    .setParameter("moduleId", moduleId)
                                    .getSingleResult();
} catch (NoResultException e) {
    LOGGER.log(Level.FINEST, "New ModuleID"); // normal case
```

```
}

if (moduleIdMatch != null) {
    LOGGER.log(Level.INFO, "ModuleID already exists in database");
    return;
}
```

Im Code wird eine benannte Query verwendet (später im Kap. 5.2.3). Der erwartete Rückgabetyp wird als zweiter Parameter angegeben und danach der Platzhalter in der *NamedQuery* durch den Wert der Modul-ID ersetzt. Erwartet wird ein einziges Ergebnis. Findet die Datenbank mehrere Einträge, wird der erste in der Liste zurückgegeben. Falls es zu einer Abfrage keine Einträge gibt, wird eine *NoResultException* geworfen. Diese Exception wird allerdings nur geworfen, wenn ein einzelner Einträg erwartet wird(`getSingleResult()`). Andernfalls wird eine leere Liste zurückgegeben.

5.2 Datenbank

In enSuite läuft das Java-basierte relationale Datenbank-Management-System **Derby**. Das Projekt wurde von der Apache Software Foundation entwickelt um leichtgewichtige Datenbanken in Java zu unterstützen. Das gesamte System ist komprimiert etwa 600 kB groß und hält sich vollständig an SQL92 und SQL99. Im Gegensatz zu großen DBM-Systemen braucht Derby keinen Administrator. Die Programme sprechen *Derby* über eine standardisierte JDBC[5]-Schnittstelle an. Unterstützt wird dabei ein **eingebetter JDBC-Modus** und ein Netzwerk-JDBC-Modus. enSuite verwendet den eingebetteten Modus bei dem der Datenbankserver mit im Prozess der Anwendung läuft. Dies vereinfacht zudem die Auslieferung des Programms, da die Bibliothek nur eingebunden werden muss. Im Netzwerk-Modus hingegen wird ein eigenständigen Server als Dienst gestartet. Während sich im Netzwerk-Modus mehrere Clients zur Datenbank verbinden können, ist die eingebette Datenbank ein **Single-User-System**.

5.2.1 Verbindung zur Datenbank

Um eine Verbindung zur Datenbank herzustellen, muss eine Datei namens **persistence.xml** im Ordner META-INF des Projekts angelegt werden. In der XML-Datei wird eine *Persistence Unit* definiert, welche die Verbindungsdaten speichert. Eine *Persistence Unit* dient zur Konfiguration der *Entity Manager Factory* (Kap. 5.1.2). Neben der Verbindungs-URL werden hier auch die Login-Daten gespeichert.

Die XML-Datei muss nicht per Hand geschrieben werden sondern wird von der IDE generiert, indem man eine neue JDBC-Verbindung anlegt. Allerdings lassen sich einige Optionen (z.B. In-Memory-Database) nur über direktes Editieren erzielen.

[5] Java Database Connectivity ist eine einheitliche Schnittstelle für Datenbanken versch. Hersteller

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <persistence>
3  <persistence-unit name="NbNppTraceMonitorPU">
4  <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
5      <validation-mode>NONE</validation-mode>
6      <properties>
7        <property name="javax.persistence.jdbc.url"
8                  value="jdbc:derby:memory:TraceMonitor;create=true"/>
9        <property name="javax.persistence.jdbc.password" value="geheim"/>
10       <property name="javax.persistence.jdbc.driver"
11                 value="org.apache.derby.jdbc.EmbeddedDriver"/>
12       <property name="javax.persistence.jdbc.user" value="benutzer"/>
13       <property name="eclipselink.allow-zero-id" value="true"/>
14       <property name="eclipselink.ddl-generation"
15                 value="drop-and-create-tables"/>
16     </properties>
17   </persistence-unit>
18 </persistence>
```

Listing 4: persistence.xml

Die *Persistence Unit* heißt in diesem Fall *NbNppTraceMonitorPU* (Z. 3) und wurde aus dem Projektnamen abgeleitet. Der Name der *Persistence Unit* wird im Programmcode zum Erstellen des *Entity Managers* genutzt:

```
EntityManager em;
em =javax.persistence.Persistence.
    createEntityManagerFactory("NbNppTraceMonitorPU").
    createEntityManager();
```

Bei jedem Programmstart werden alle Tabellen der Datenbank gelöscht und neu angelegt (Z.15), sodass gespeicherte Daten verloren gehen. Über die Verbindungs-URL (Z.8) lässt sich das Verhalten des Treibers steuern. Das Schlüsselwort `memory:` sagt aus, dass die Datenbank im Speicher gehalten werden soll. Falls noch keine Datenbank mit dem Namen „TraceMonitor“ exisitiert sorgt `create=true` dafür, dass diese vorher angelegt wird.

5.2.2 Tabellen erstellen

Nachdem Erstellen des *Entity Managers* ist man bereits automatisch zur Datenbank verbunden. Diese ist allerdings noch leer. Es müssen erst noch Tabellen erstellt werden.

Tabellen lassen sich

- im Java-Programm mittels *Data Definition Language(DDL)* erzeugen oder
- über die Netbeans-IDE anlegen.

Da das Domänen-Modell frei von ralationalen Sprachen wie SQL sein sollte, ist es sinnvoller die Tabellen über die IDE zu erzeugen. Nebeans bietet eine grafische Oberfläche oder eine SQL-Konsole an, wobei letztere vorgezogen werden sollte. Nur über die Konsole ist es möglich, den Primärschlüssel automatisch zu inkrementieren (Z. 2) oder einen zusammengesetzten Primärschlüssel (*Compound Key*) anzugeben (Z. 14). Auch Fremdschlüssel(Z. 12) können so erstellt werden.

```
1  CREATE TABLE "FILE" (
2  "file_id" INT not null primary key
3          GENERATED ALWAYS AS IDENTITY
4          (START WITH 1, INCREMENT BY 1),
5  "file_index" INT not null,
6  "file_name" VARCHAR(256),
7  "log_level" INT
8  );
9
10 CREATE TABLE "MODULE_FILE"
11 (
12 "module_id" INT not null references MODULE("module_id"),
13 "file_id" INT not null references FILE("file_id"),
14 PRIMARY KEY("module_id", "file_id")
15 );
```

Listing 5: DDL zur Tabellen-Erzeugung

5.2.3 Entity-Klassen erstellen

Entity-Klassen, also annotierte Klassen um persistierbare Objekte zu erzeugen[6], können von Netbeans aus einer JDBC-Verbindung heraus **automatisch generiert** werden. Dabei spielt es keine Rolle, um was für eine Datenbank es sich handelt. In Netbeans klickt man dazu mit der rechten Maustaste auf den Paketnamen und wählt **New** → **New Entity Classes from Database**.

Besteht also schon eine Datenbank, kann der Java-Programmierer sehr schnell Entities erzeugen und diese nutzen. Sollte sich doch mal eine Tabelle auf dem Datenbank-Server ändern, so können die Entities über die IDE aktualisiert werden. Besteht dagegen noch keine Datenbank, kann der Entwickler auch die Entity-Klassen selber schreiben und daraus Datenbank-Tabellen erzeugen lassen. Je nach persönlicher Präferenz fängt man also mit der Erstellung der Datenbank oder Entity-Klassen an. Die meisten Entwickler werden vermutlich erst eine Datenbank aufsetzen und daraus die Entitäten erzeugen als umgekehrt, da mehr Wissen in diesem Gebiet vorhanden ist.

Aus den fünf Tabellen der `TraceMonitor`-Datenbank werden insgesamt vier Entitäten erstellt. Die Role-Table `MODULE_FILE` ist nicht mehr nötig, da *n:m-Beziehung* im

[6] Vgl. Kap. 5.1.1

Domänen-Modell problemlos möglich sind. `EclipseLink`, ebenfalls zuständig für die Generierung der Entity-Klassen, analysiert die Beziehungen und nimmt ggf. sinnvolle Rationalisierungen vor.

Anhand der erzeugten Entity `File` sollen die Annotationen und Beziehungen erläutert werden. Das Listing 6 wurde dazu auf die relevanten Stellen gekürzt.

Jede Entity muss das Interface `java.io.Serializable` implementieren, da die *Java Persistence API* ausschließlich mit diesem Typ arbeitet. Das Interface besitzt weder Attribute noch Funktionen. Es gibt an, dass ein Objekt serialiserbar ist und hat somit nur eine semantische Bedeutung. Um aus `File` eine Entity zu machen, muss die Klasse zunächst mit `@Entity` annotiert werden. Diese Annotation dient EclipseLink primär als Marker. Jede Entität benötigt einen einzigartigen Primärschlüssel um die Attribute der Tabelle zu identifizieren. Dieser Schlüssel wird in Zeile 23 mit `@Id` festgelegt. Gleichzeitig wird durch Zeile 24 dafür gesorgt, dass der Wert automatisch hochgezählt wird. Im Programm lässt man einfach den Konstruktor leer bzw. ruft die Setter-Methode von fileId nie auf. Über die `@column` Annotation wird jedem Attribut eine Spalte in der Tabelle zugewiesen. Zur Unterscheidung der Variablen und Spaltennamen wandelt Netbeans die Namen leicht ab, sodass aus `file_Id` der Name `fileId` wird. In den Zeilen 11-20 werden so genannte *NamedQueries* definiert, welche von Netbeans automatisch erstellt werden. Dies sind häufig genutzte Abfragen mit einem eigenen Alias Namen. Auf die Art lassen sich leicht alle Entities oder welche mit einem bestimmten Attributswert abfragen ohne dafür selbst eine Query formulieren zu müssen. *NamedQueries* werden effizienter vom EntityManager abgearbeitet als einfache Queries und sollten, wenn möglich, verwendet werden. In der Klasse lassen sich beliebig viele *NamedQueries*, auch über mehrere Tabellen, eintragen. In Zeile 40 wird die Umsetzung der **n:m Beziehung** zwischen `FILE` und `MODULE` veranschaulicht. Eine *Collections*-Klasse wird dazu genutzt die Beziehungen zwischen einer Datei und mehreren Modulen zu speichern. Die Klasse Module hingegen besitzt eine Collection um alle Dateien zu speichern. Somit ist die Verbindung **bidirektional**, da beide Seiten Informationen über ihre Beziehung speichern. Wie in Zeile 42 zu sehen ist, funktionieren **1:n Beziehungen** ganz ähnlich. Allerdings besitzt nur eine der beiden Klassen eine *Collection* und die andere einen *einfachen* Integer-Typ zur Speicherung des Primärschlüssels. In den Zeilen 35-38 wird der Fremdschlüssel annotiert, sodass beim Löschen einer Entität dessen Beziehungen berücksichtigt werden. Der EntityManager kann alle Kinder einer Eltern-Entität autoamtisch mit löschen. Dazu muss die Kaskadierungs-Option, eingeschaltet werden. In Zeile 42 wurde diese Option gesetzt, sodass beim Löschen einer Datei auch alle Traces gelöscht werden. Grundsätzlich ist die **Kaskadierung beim Löschen** nicht einfach, da Kind-Entitäten z.B. Beziehungen zu weiteren Eltern besitzen können. Diese sollten nicht einfach gelöscht werden. Im Zweifel sollten Daten manuell entfernen werden. Bei **n:m Beziehnungen** muss besonders darauf geachtet werden, ob ein Löschvorgang kaskadiert werden sollen. Schnell sind bei dem Vorgang gleich mehrere Tabellen betroffen und eine Kettenreaktion wird ausgelöst. Da der EntityManager bei jeder Transaktion die referentielle Integrität der Datenbank überprüft, führt so ein wildes Löschen schnell zu Fehlern.

```java
package com.elster.nppTraceMonitor.db;

import java.io.Serializable;
import java.util.Collection;
import javax.persistence.*;

@Entity
@Table(name = "FILE")
@XmlRootElement
@NamedQueries({
    @NamedQuery(name = "File.findAll",
            query = "SELECT f FROM File f"),
    @NamedQuery(name = "File.findByFileId",
            query = "SELECT f FROM File f WHERE f.fileId = :fileId"),
    @NamedQuery(name = "File.findByFileIndex",
            query = "SELECT f FROM File f WHERE f.fileIndex = :fileIndex"),
    @NamedQuery(name = "File.findByFileName",
            query = "SELECT f FROM File f WHERE f.fileName = :fileName"),
    @NamedQuery(name = "File.findByLogLevel",
            query = "SELECT f FROM File f WHERE f.logLevel = :logLevel")})
public class File implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Basic(optional = false)
    @Column(name = "file_id")
    private Integer fileId;
    @Basic(optional = false)
    @Column(name = "file_index")
    private int fileIndex;
    @Column(name = "file_name")
    private String fileName;
    @Column(name = "log_level")
    private Integer logLevel;
    @JoinTable(name = "MODULE_FILE", joinColumns = {
        @JoinColumn(name = "file_id", referencedColumnName = "file_id")},
        inverseJoinColumns = {
                @JoinColumn(name = "module_id", referencedColumnName = "module_id")})
    @ManyToMany
    private Collection<Module> moduleCollection;
    @OneToMany(cascade = CascadeType.ALL, mappedBy = "fileId")
    private Collection<Trace> traceCollection;

    public File() {
```

```
45      }
46
47          // Weitere Konstruktoren und
48          // Getter & Setter hier ausgelassen
49
50          // hash()-, equals()- und toString()-Funktion ausgelassen
51
52  }
```

Listing 6: Entity Klasse

5.3 Kommunikation

Der Trace-Monitor ist nun in der Lage Traces zu speichern, aber es fehlt noch die Kommunikation mit dem Gerät um überhaupt Nachrichten zu erhalten. In diesem Kapitel soll es um das Verwalten der Verbindung und die Realisierung des TM-Protokolls gehen.

Zum Öffnen einer TCP-Verbindung wird in Java die Klasse `java.net.socket` verwendet. Da die *read*- und *write*-Methoden blockierend arbeiten, werden die Aufrufe in Threads ausgelagert. Es macht Sinn, einen **Reader- und einen Writer-Thread** zu benutzen. Dadurch blockiert sich das Lesen- und Schreiben vom Socket nicht gegenseitig.

5.3.1 Verwaltung

Der Reader-Thread liest in einer Endlosschleife von dem Socket. Er wartet auf genau zwei Bytes. Das erste Byte ist der Token und identifiziert einen Befehl. Das zweite Byte ist die Länge der nachfolgenden Daten. Das Hauptprogramm registriert sich als *Listener* bei dem Reader-Thread, um über gelesene Daten informiert zu werden. Dazu wurde das *Beobachter*-Muster umgesetzt. Die Benutzeroberfläche bleibt so reaktionsfähig. Die `processToken`-Methode identifiziert das Token und benachrichtigt alle Listener.

```
1  public void run() {
2          in = new DataInputStream( pSocket.getInputStream() );
3
4          while( true ) {
5              buffer = new byte[2];
6              in.readFully(buffer, 0, 2);
7
8              token  = new TMPToken( buffer[0] );
9              length = (int) buffer[1];
10
11             // read following data
12             if(length > 0) {
13                 buffer = new byte[length];
14                 in.readFully(buffer, 0, length);
```

```
15                  dataField = buffer.clone();
16              }
17              else {
18                  dataField = null;
19              }
20
21             // Server disconnect
22              if( token.isFromDevice() && token.isDisconnectCommand() ) {
23                  serverDisconnectedSignal ();
24                  return; // exit thread
25              }
26
27              processToken( token, dataField );
28          }
29  }
```

Der Writer-Thread besitzt eine Warteschlange für Tokens und legt sich schlafen, wenn gerade kein Kommando zu senden ist. Die *commandQueue* ist vom Typ `java.util.LinkedList` und muss synchronisiert werden, da sowohl der aktuelle Thread als auch das Hauptprogramm gleichzeitig darauf zugreifen. Wenn die Warteschlange leer ist, legt sich der Thread schlafen. Aufgeweckt wird der Thread durch ein `notify`-Signal, welches durch Füllen der Warteschlange von außen ausgelöst wird. Der WriterThread darf sich aber nicht die ganze Zeit „schlafen legen“, da in bestimmten Abständen ein Idle-Signal zum Gerät gesendet werden muss. Bleibt dieses Lebenszeichen aus, trennt das Gerät die Verbindung. Um innerhalb der Idle-Periode aufzuwachen, wird bei der `Wait()`-Methode ein Timeout als Parameter gesetzt.

```
1  public void run() {
2      while( !Thread.currentThread().isInterrupted() ) {
3              synchronized( commandQueue )  {
4                  while ( commandQueue.isEmpty() )  {
5                      commandQueue.wait( TMP.IDLE_PERIOD );
6
7                      //Idle-Signal senden
8                      out.write ( TMPToken.PC | TMPToken.IDLE );
9                      out.write ( 0x00 ); }
10             }
11
12             TMCommand command = commandQueue.poll();
13             out.write( command.getCommandId() ); // write command
14             out.write( command.getLength() );
15     }
16 }
```

Das Dekodieren der Daten vor dem Versenden und das Enkodieren beim Empfangen ist Aufgabe des Trace-Monitor-Protokolls. Da der Aufbau der Felder immer gleich ist und sich die Bedeutungen der Bits an einer Position nicht ändern, kann man zur Realisierung eine Reihe von Konstanten nutzen. Um aus einer Byte-Folge mehrere Werte zu extrahieren, verwendet man **Bitmasken**. Eine bitweise Verundung werden die interessanten Bits ausgewählt und mittels Shift-Operation an die passende Stelle geschoben.

```
// Wert extrahieren
int moduleId = (data & TMP.MODULE_ID_BITMASK) >> TMP.MODULE_ID_SHIFT;

// Wert setzen
byte data  |= (moduleId << TMP.MODULE_ID_SHIFT) & TMP.MODULE_ID_BITMASK;
```

Wie in dem Beispiel zu sehen ist, speicher die Java-Klasse *TMP* die Bitmasken und Shift-Längen als statische Konstanten. Weiterhin beinhaltet die Klasse die Port-Nummer für die Verbinung, den Versionsnummer des Protokolls und die Byte-Order.

5.3.2 Byte-Reihenfolge

Das Gerät versendet Daten in der *Little Endian* Byte-Reihenfolge. Das niederwertigste Byte wird also als erstes versendet. Bei der JVM steht gewöhnlich das höchstwertige Byte vorne im Speicher, sodass die Reihenfolge getauscht werden muss. Es ist sinnvoll die Byte-Reihenfolge direkt nach dem Empfang der Nachricht und kurz vor dem Senden einer Nachricht zu setzen. Um die Reihenfolge zu ändern, sollte auf die Java Klassen `java.nio.ByteBuffer` und `java.nio.ByteOrder` aus dem *New I/O Paket* zurückgegriffen werden. Ein ByteBuffer-Objekt kann Speicher allozieren und eine Kopie des Arrays speichern oder direkt auf dem originalen Array arbeiten. Im letzteren Fall umschließt der ByteBuffer sozusagen das Array, wie in dem Beispiel vorgeführt wird.

```
ByteBuffer  buffer = ByteBuffer.wrap( timestampArray );
buffer.order( TMP.BYTE_ORDER );
long utcTime = buffer.getLong(0);
```

Als Rückgabewerte werden alle primitiven Datentypen unterstüzt. Der Parameter in den Getter-Methoden gibt an, wo das erste Byte zu finden ist. Je nach Byte-Reihenfolge wird dieses als niederwertigstes oder höchstwertige Byte interpretiert. Der ByteBuffer holt sich die Anzahl an Bytes aus dem Array, wie für den Datentyp notwendig ist. Ist das Array zu klein wird eine *IndexOutOfBoundsException* geworfen. Zeichenketten (Strings) müssen nicht ungewandelt werden, da der Datentyp `char` genau einem Byte entspricht. Verlangt die `printf`-Funktion einen String kann also auch ein Byte-Array als Parameter angegeben werden. Allerdings ist darauf zu achten, den String mit dem Nullzeichen `\0` zu terminieren.

5.4 Anzeige des Datenbankinhalts

Die performante Anzeige der Trace-Nachrichten aus der Datenbank ist eine wichtige und zugleich schwierige Aufgabe. Wird jede einzelne Nachricht direkt in der Oberfläche

angezeigt, gibt es zu viel Overhead und das Programm reagiert nur noch langsam auf Benutzereingaben. Es muss also ein Lösung gefunden werden Einträge gebündelt aus der Datenbank zu holen und anzuzeigen. Das fertige Konstrukt soll dabei anpassungsfähig bleiben, sodass Spalten in der Tabelle nachträglich umbenannt oder vertauscht werden können. Auch soll der Inhalt aus der Datenbank nicht immer unverändert übernommen werden, wie beispielsweise der Zeitstempel. Der SQL-Datentyp `Date` zeigt unformatiert nur das Datum an, obwohl nur die Uhrzeit von Interesse ist.

Es gibt noch weitere Herausforderungen: aus der Zeile einer Tabelle muss sich die Entität zurückerhalten lassen. Diese Konvertierung muss immer dann vorgenommen werden, wenn Benutzer Eingaben tätigen. Zum Ändern des Trace-Levels etwa wird die File-ID benötigt, welche aber nur in der Entität und nicht in der Tabellenzeile steht.

Betrachtet man noch einmal welche Werte überhaupt in der Trace-Tabelle angezeigt werden sollen (Tab. 2), so stellt man fest, dass sich diese aus einem Verbund von mehreren Datenbank-Tabellen (JOIN) zusammensetzt. Anstatt eine Datenbank-Tabelle eins zu eins auf eine Swing-Tabelle zu mappen, muss also ein JOIN abgebildet werden.

Das Eintragen in der Datenbank soll nicht von Hand programmiert werden, da dies nicht ohne Einbau einer rationalen Sprache wie SQL im Objektmodell funktioniert. Das Ändern oder Tauschen von Spalten ist so auch immer mit Änderungen im Code verbunden, also unflexibel. Für das automatische Anzeigen von Werten in einer Swing-Tabelle auf Basis einer Datenquelle kann *BeansBinding* verwendet werden.

5.4.1 Einführung in BeansBinding

BeansBinding wurde entwickelt um die Eigenschaften von zwei Objekt synchron zu halten. Dabei lässt sich auswählen, welche Eigenschaften synchronisiert werden sollen und in welcher Richtung dies geschieht. Standardmäßig wird bidirektional synchronisiert.

Die Swing-Table benötigt noch eine Datenquelle zur Synchronisation. Für eine Swing-Tabelle kommen nicht beliebige Objekte in Frage. In der Praxis wird häufig eine observierbare Liste herangezogen, welche entsprechende Eigenschaften besitzt. Deshalb müssen alle Einträge einer Datenbank-Tabelle zunächst in einer Liste gespeichert werden. Da alle Traces in der Tabelle `MODULE_TRACE` gespeichert werden, wird die Entität ModuleTrace nach allen Einträgen befragt. Zeilen 1-2 in dem Beispiel unten zeigen diese Abfrage.

Die Liste `moduleTraceList` kann nun als Datenquelle für die Swing-Tabelle verwendet werden (Z. 5). Danach wird den Spalten der Reihe nach eine Eigenschaft (engl. Property) zugewiesen. Der Ausdruck `&{timestamp}` bezieht sich auf die Eigenschaft in der ModuleTrace-Entität. Die Datenbank-Abfrage in Zeile 2 hat ja eine Liste von ModuleTrace-Entitäten zurückgegeben auf welche sich nun diese Ausdrücke beziehen.

Um nun Daten aus anderen Datenbank-Tabellen mit in die Swing-Tabelle aufzunehmen (JOIN), kann man wiederum die Ausdrücke benutzen. So wurde in Z. 18 der Dateiname aus der `FILE`-Tabelle mit in die Swing-Tabelle eingebaut. Dazu werden Properties einfach verknüpft. Im Programm ist dies ja auch ohne weiteres möglich:

```
ModuleTrace moduleTrace = em.
      createNamedQuery("ModuleTrace.findByModuleTraceId",Module.class).
```

```
      setParameter("moduleId", 1).
      getSingleResult();
File f = moduleTrace.getTraceId().getFileId();
```

Ausdrücke sind abstrakter und dienen ganz allgemein zur Beschreibung von Eigenschaften beliebiger Klassen.

```
moduleTraceQuery = em.createQuery("SELECT m FROM ModuleTrace m");
moduleTraceList  =
ObservableCollections.observableList(moduleTraceQuery.getResultList());

jTableBinding = SwingBindings.createJTableBinding(
        UpdateStrategy.READ, moduleTraceList, traceTbl);
columnBinding = jTableBinding.addColumnBinding(
        org.jdesktop.beansbinding.ELProperty.create("${timestamp}"));
columnBinding.setColumnName("Timestamp");
columnBinding.setColumnClass(java.util.Date.class);
columnBinding.setEditable(false);
columnBinding = jTableBinding.addColumnBinding(
        org.jdesktop.beansbinding.ELProperty.create("${moduleId.moduleId}"));
columnBinding.setColumnName("Module Id");
columnBinding.setColumnClass(Integer.class);
columnBinding.setEditable(false);
columnBinding = jTableBinding.addColumnBinding(
        org.jdesktop.beansbinding.ELProperty.create("${traceId.fileId.fileName}"));
columnBinding.setColumnName("File");
columnBinding.setColumnClass(String.class);
columnBinding.setEditable(false);
// usw. für die restlichen Spalten
```

5.4.2 Konvertierung von Tabellenzeilen

Im vorherigen Kapitel wurde beschrieben, wie Entities automatisch auf eine Swing-Tabelle gemapped werden. Der Weg von einer Zeile zurück zur Entität fehlt noch. In der Trace-Monitor GUI (Abb. ??), sieht man insgesamt drei Tabellen. Aus der File-Tabelle (unten links) muss die File-ID extrahiert werden können, um den Trace-Level zu ändern. Aus der Trace-Tabelle (rechts) muss bei einem Doppelklick der Dateiname und die Zeilnenummer extrahiert werden können, um die Datei in einem externen Editor zu öffnen. Statt einfach den Wert in der Spalte zu nehmen, wird auf die Datenquelle selbst zugegriffen. Im Falle der File-Tabelle wäre das auch garnicht möglich, da die ID garnicht angezeigt wird.

Swing-Tabellen basieren auf dem *Model View Controller*-Prinzip, bei dem die Daten von der Darstellung entkoppelt werden. Zum Controller `JTable` gehört die Modell-Klasse `TableModel`. Anzeige und Speicherung der Daten wird so getrennt.

Zur Konvertierung von Zeilen zurück zum Modell kann die Methode `convertRowIndexToModel` aus der Klasse `JTable` verwendet werden.

```
int[] selectedRows = fileTbl.getSelectedRows ();
for(int i : selectedRows ) {
    fileModel = fileList.get( fileTbl.convertRowIndexToModel( i ) );

        // Frage über File-ID die Module-ID ab
        // Sende neuen Trace-Level zum Gerät
}
```

5.4.3 Formatierung von Zellen in der Tabelle

Der Datentyp `java.sql.Date` zeigt in der Standard-Formatierung nur das Datum ohne Uhrzeit an. Um diesen Wert vor der Anzeige in der GUI anzupassen, muss ein *Renderer* für die Zellen der Tabelle geschrieben werden. In dem unteren Beispiel wird die Uhrzeit mit drei Nachkommastellen im Sekundenbereich angezeigt. Der *TableCellRenderer* soll nicht für alle Zellen, sondern nur für Zellen mit dem Datentyp `Date` gelten soll. Um dies zu bewirken, muss bei der Installation des Renderers der Spaltentyp als erster Parameter übergeben werden. Jede Spalte in einer `Swing-JTable` *kann* auf einem bestimmten Datentyp spezialisiert werden. Tut man dies nicht, gilt autoamtisch der Typ `Object`.

```
traceTbl.setDefaultRenderer(Date.class, new MyTableCellRenderer());

public class MyTableCellRenderer extends DefaultTableCellRenderer {
    private SimpleDateFormat df = new SimpleDateFormat( "HH:mm:ss.SSS" );

    @Override
    public Component getTableCellRendererComponent(JTable table, Object value,
            boolean isSelected, boolean hasFocus, int row, int column) {
        if(value instanceof Date)
        {
            df.setTimeZone ( TimeZone.getTimeZone ( "GMT+0:00")); // London
            value = df.format(value);
        }
        return super.getTableCellRendererComponent(table, value,
                        isSelected, hasFocus, row, column);
    }
}
```

Listing 7: Renderer für die Zellen einer Swing-Tabelle

Um die Nachricht zusammen zu bauen, muss der Trace-Monitor die Parameter in den Formatierungsstring einsetzen. Für diese Aufgabe gibt es in Java die Klasse `java.util.Formatter`,

welche stark an die C-Funktion `printf` angelehnt ist. Obwohl die Implementierung nicht ganz vollständig ist, reicht die `format`-Funktion für die meisten Anwendungsfälle in der Praxis aus. Anstatt den fertigen String auf die Standardausgabe zu drucken, wird dieser als Parameter zurückgegeben. Die Signatur der Methode sieht so aus:

```
String java.util.String.format (String formatString, Object[] args);
```

Bei der Anzeige von fertigen Traces traten vereinzelt Konvertierungsfehler auf, welche auf den Platzhalter `%lld` zurückzuführen waren. Dieser steht für den Datentyp `long long` und muss in Java wie ein Integer behandelt werden. In allen Formatierungsstrings wird dieser Platzhalter deshalb durch ein einfaches `%d` ersetzt.

6 Anhang

6.1 Abgrenzung Logging und Tracing

Mit Logging wird allgemein eine Form der Ausgabe von Informationen eines Programms zur seiner Laufzeit bezeichnet. Im Gegensatz zu einer einfachen Ausgabe in einer Konsole oder auf einem Terminal werden Logs im Allgemeinen länger gespeichert, z.B. als Dateien. Die enthaltenen Informationen sind oft in einem standardisierten Format, welches z.B. einen Zeitstempel enthält. Es ist üblich, dass sich die Art bzw. die Menge an Informationen die geloggt werden soll einstellen lässt. So können beispielsweise nur Fehler aufgezeichnet werden, während Warnungen nicht ausgegeben werden.

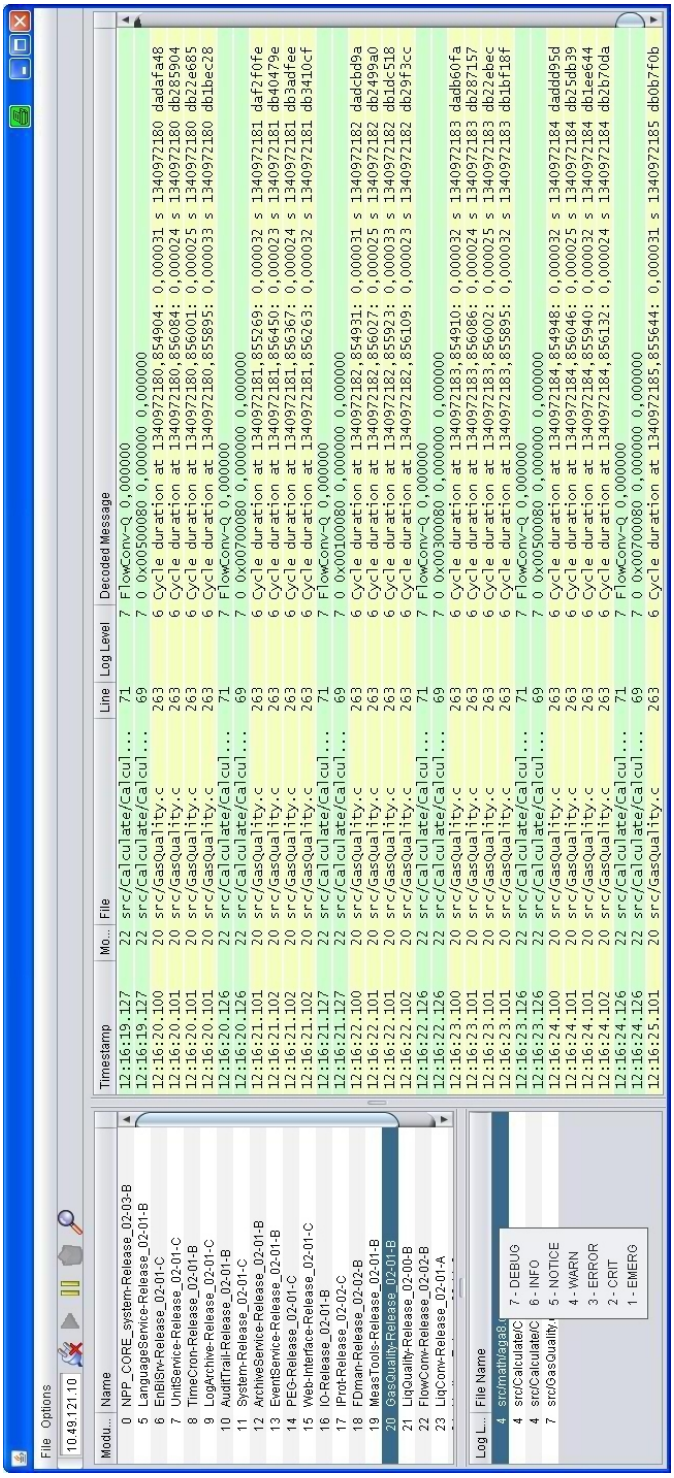
Bei dem Begriff des Tracing liegt der Schwerpunkt auf der Verfolgung des Ablaufs eines Programms. Es soll nachvollziehbar sein, wann welche Funktion mit welchen Eingangsdaten zu welchem Ergebnis geführt hat. Es gibt hierbei zwei verschieden Ansätze. Zum einen kann das Programm zur Laufzeit durch einen bestimmen Prozess überwacht und manipuliert werden. Dabei können neben Systemaufrufen vom Benutzer definierte Trace-Points gesetzt werden. Zum anderen kann das Programm mit zusätzlichen Funktionen übersetzt werden, welche die entsprechenden Informationen von sich aus erzeugen. Anstatt die Ausgabe des Logging-Dienstes in einer Datei auf der lokalen Festplatte zu schreiben, können Nachrichten auch parallel über das Netzwerk oder zu einer anderen Schnittstelle gesendet werden.

Wie man erkennt, haben Logging und Tracing etliche Überschneidungen. Es müssen jeweils Informationen zur Laufzeit erstellt werden, welche nicht zu der eigentlichen Funktionalität des Programms gehören. Beide benötigen einen Zeitstempel, auch wenn für Traces eine höhere Genauigkeit erforderlich ist. Die generierten Informationen sollen speicherbar sein um sie später auswerten zu können. Wird das Tracing in das Programm integriert und nicht von außen erzeugt, besteht somit kaum noch ein Unterschied zum Logging. Auf Grund dieser Gemeinsamkeiten wird im folgenden Logging und Tracing gleichgesetzt. Durch die Trace-Nachrichten soll es möglich sein eine genaue Übersicht über den Ablauf des Programms zu erhalten. Alle Trace-Messages zusammen ergeben den äußeren Zustand des Geräts. Der Programmierer muss dieses Konzept aber auch konsequent nutzen, sodass ausreichend Informationen zur Verfügung stehen.

6.2 Tracing

Oberstes Ziel ist es den äußeren Zustand des Geräts durch die Menge aller Trace-Nachrichten abzubilden. Je konsequenter die Entwickler das Tracing-Konzept umsetzen, desto genauer ist auch dieser Zustand. Im Wesentlichen geht es beim Tracing um das Nachvollziehen des Programmablaufs. Jede Nachricht wird dazu mit einem Zeitstempel versehen und muss natürlich auch in der richtigen Reihenfolge vom Monitor angezeigt werden. Der Trace-Monitor wird so zur Überwachungsschnittstelle für die Entwickler. Man möchte gerne erfahren zu welchen Ergebnis ein Aufruf mit gegebenen Parametern geführt hat. Tracing ist eng verwandt mit dem Logging, reicht jedoch etwas weiter. Logging dient in erster Linie dem Administrator dazu Dienste zu überwachen. Dabei werden auf Anwendungsebene Statusmeldungen festgehalten. Beim Tracing wird auf unterster Ebene geloggt, beispielsweise Exceptions innerhalb von Funktionen. In der Regel ist das Datenaufkommen beim Tracing viel höher als beim Logging und kann sogar dazu führen, dass die Performance des Geräts massiv beeinträchtigt wird.

6.3 Trace-Monitor im Einsatz



Literatur

[MS09] Mike, Keith ; Schincariol, Merrick: *Pro JPA 2 - Mastering the Java Persistence API*. 4. Apress, 2009. – ISBN 978–1–4302–1956–9