Marcus Powarzynski

(E-Mail: mp@maurer-treutner.de)
war viele Jahre als freiberuflicher Trainer, Softwarearchitekt und Entwickler tätig. Seit dem Jahr 2003 arbeitet er für Maurer & Treutner als Coach und unterstützt Kunden der Firma bei der Optimierung von Verfahren und Einführung von Methoden. Seine Beratungstätigkeit hat maßgeschneiderte Architekturen und effizientes Teamwork zum Ziel. Marcus Powarzynski ist Diplom-Informatiker und hat an der Universität Kaiserslautern studiert.

Effiziente Anbindung lokaler Datenbanken in eingebetteten Systemen

Bei der Entwicklung von eingebetteten Systemen spielt die Frage der persistenten Datenhaltung häufig eine wichtige Rolle. Verschiedenartige, zum Teil komplexe Datenstrukturen sollen dauerhaft gespeichert werden. Je nach Komplexität der Daten und Beziehungen der Daten untereinander lohnt sich der Einsatz einer lokalen relationalen Datenbank. Entscheidet man sich für diesen Weg, stellen sich bald eine Reihe von Fragen: Ist die SQL-API des Datenbanklieferanten ausreichend oder ist der dadurch gebotene Abstraktionslevel zu niedrig? Wie kann die Verwaltung, Bereitstellung und Parametrierung der benötigten SQL-Statements architektonisch vernünftig gelöst werden? Wie sollen Ergebnisse aus Datenbankabfragen repräsentiert und an die benötigten Stellen in der Anwendung weitergegeben werden? Wie bildet man das Navigieren im Fachmodell ab? Soll ein Persistenzframework eines Drittanbieters zum Einsatz kommen? In diesem Artikel wird eine einfache Lösung skizziert, die auf eingebettete Systeme zugeschnitten ist und zugleich wichtige nichtfunktionale Anforderungen wie Verständlichkeit, Änderbarkeit und Erweiterbarkeit erfüllt. Die Lösung kann u. a. im Umfeld der Programmiersprachen C und C++ eingesetzt werden.

Motivation

Im Bereich der Informationssysteme haben sich sowohl relationale Datenbanken als auch objektorientierte Programmierung schon seit geraumer Zeit durchgesetzt. Dementsprechend ist das Thema, wie objektorientierte Fachmodelle in Code umgesetzt werden und Fachobjekte in die Datenbank abgespeichert bzw. aus ihr geladen werden können, ausführlich erforscht und diskutiert worden, z. B. in [Fow03], [Amb03]. Es gibt mittlerweile bewährte Konzepte und Vorgehensweisen (u. a. dokumentiert in *Design Patterns*) und darauf beruhende *Persistance Frameworks*, die kommerziell oder als Open Source erhältlich sind. Vor allem im JAVA-Umfeld gibt es eine nahezu unübersichtliche Zahl an Möglichkeiten, ein bekanntes Framework ist beispielsweise Hibernate [Hib]. Untersucht man die Möglichkeit, solche Persistance Frameworks in eingebetteten Systemen einzusetzen, stößt man auf eine ganze Reihe von Punkten, die dagegen sprechen, u. a.:

- Der Footprint sowohl im RAM als auch im ROM ist bei vielen dieser Frameworks sehr groß.
- Das Speichern und vor allem das Laden von Daten ist selbst bei relativ kleinen Datenmengen mit umfangreichen Hauptspeicheroperationen verbunden, wobei intensiv von dynamischer Speicherverwaltung Gebrauch gemacht wird. Die damit verbundene Last überfordert die meisten in eingebetteten Systemen eingesetzten CPUs.
- Viele Persistance Frameworks stellen erhebliche Voraussetzungen an die Umgebung, sie sind beispielsweise nur mit „großen" Datenbank-Engines einsetzbar oder bieten nur eingeschränkte Möglichkeiten für das Mapping auf bestehende Datenbankschemata.
- Viele dieser Frameworks geben zum Teil vielfältige Rahmenbedingungen bezüglich des Entwurfs und der Implementierung der Fachklassen bzw. der gesamten fachlichen Schicht einer Anwendung vor. Es besteht die Gefahr, dass die Gesamtarchitektur vom Framework dominiert wird. Auf die Bedürfnisse von RTE-Systemen wird dabei keine Rücksicht genommen.

Ein alternativer Lösungsansatz

SQL-basierte Datenbanklösungen, die sich auch für den Einsatz in eingebetteten Systemen eignen, bieten eine API, um auf das installierte Datenbankschema mittels SQL zuzugreifen. Diese technische Programmierschnittstelle ist jedoch vom Abstraktionsgrad zu niedrig, um komplexere Fachmodelle ohne zusätzliche Systematik umzusetzen, wenn der Anspruch an Änderbarkeit, Erweiterbarkeit und Wartbarkeit gegeben ist. Unsere in der Praxis bewährte Lösung geht einen Mittelweg zwischen Persistance Frameworks mit vollautomatischem O/R-Mapping und der direkten Programmierung mit SQL in der Anwendungslogik. Sie beruht auf der kombinierten Anwendung bekannter Design Patterns zur Anbindung relationaler

Datenbanken. Die Kernidee ist eine Variante des DAO-Ansatzes (Data Access Object). Auf diese Weise entsteht eine schlanke Library zur Anbindung, die zusammen mit der SQL-API des Datenbanklieferanten die Grundlage für die Programmierung der Fachlogik bietet. Passend zu dieser Library haben wir eine Systematik entwickelt, wie die Fachlogik zu strukturieren ist. So entstehen Zugriffsschichten, deren Struktur und Schnittstellen nach ein und demselben durchgängigen Schema aufgebaut sind.

Erweiterbarkeit, Wartbarkeit und Wiederverwendbarkeit sind also gegeben. Die Lösung gewährleistet hohe Effizienz, hat einen kleinen Footprint und ist so einfach gehalten, dass einer Weiterentwicklung und Wartung auch in kleineren Unternehmen nichts im Wege steht. Man muss kein Datenbankexperte sein, um Fachmodelle umzusetzen.

Einige grundsätzliche Fragestellungen zur Architektur und zum Entwurf

Nachdem wir eingangs die Vision und die Eigenschaften der Lösung skizziert haben, sollen im Folgenden die Grundzüge der Architektur bzw. des Designs umrissen werden.

Wir gehen zunächst davon aus, dass der Anwendung ein Schichtenmodell als Architekturstil zugrunde liegt. Auch in kleinen RTE-Systemen ist die Ausgliederung der Anwendungslogik als eigene Schicht sinnvoll. Unterhalb der Anwendungslogik wird oft die Zugriffsschicht auf Ressourcen jeglicher Art (Geräte, Kommunikation usw.) platziert. Man muss sich darüber Gedanken machen, wie sich in dieses Anwendungsschema die Persistierung eingliedert. Wir formulieren einige konkrete Fragen:

Soll eine Trennung der Anwendungslogik und der Fachlogik in zwei separate Schichten vorgenommen werden?

Diese Trennung in Application Layer und Domain Layer ist in Informationssystemen üblich und kann ohne Weiteres auch für eingebettete Systeme ohne Nachteile übernommen werden. Der Abstraktionsgrad des Domain Layer liegt auf der gleichen Ebene wie beispielsweise der von logischen Geräten. Im Domain Layer sind im Wesentlichen Strukturen und Funktionalitäten angesiedelt, die fachlich motiviert sind und anwendungsübergreifend wiederverwendet werden können. Im Application Layer hingegen findet sich die Implementierung der Systemprozesse (konkrete Workflows, gegebenenfalls nebenläufig, die im Allgemeinen neben dem Zusammenspiel der Elemente der Fachlogik auch die Interaktion mit Sensoren und Aktoren steuern).

Soll bei Strukturierung des Domain Layers die „reine Lehre" der Objektorientierung zum Einsatz kommen, also die vollständige Umsetzung des objektorientierten Paradigmas bis auf Implementierungsebene?

Diese Fragestellung ist wiederholt in der Literatur diskutiert worden und war kürzlich auch wieder Gegenstand in Abhandlungen bezüglich des Für und Wider von serviceorientierten Architekturen. Der Ansatz der vollständigen Umsetzung des objektorientierten Paradigmas geht davon aus, dass das Fachmodell als Klassenmodell entworfen wurde, in dem Klassen neben Attributen auch Operationen enthalten, über Assoziationen und Vererbung in Beziehungen zu einander stehen und eventuell Design Patterns zur Variation von Verhalten (z. B. Strategy Pattern) benutzt werden. Solche Modelle werden dann nahezu 1:1 in die Implementierung umgesetzt. Jede Instanz einer Klasse aus dem Fachklassenmodell wird zu einer echten, dynamisch allokierten Instanz in der Implementierung. Auf diese Weise ist der Domain Layer mit den zu implementierenden Datenstrukturen und Funktionalitäten auch wohl strukturiert. Die meisten Persistance Frameworks mit O/R-Mapping basieren auf diesen Ansatz.

Unsere Erfahrung zeigt jedoch, dass man in vielen eingebetteten Systemen wegen des hohen Overheads durch das O/R-Mappings davon Abstand nehmen sollte. Die Alternative strukturiert den Domain Layer in *DomainComponents*, die sich durch gröbere Granularität im Vergleich zu Fachklassen auszeichnen. Der Bezug zu Fachklassen aus der Analyse ist nicht mehr ganz so direkt wie bei der 1:1-Umsetzung. Der Einsatz von Polymorphie zur Variation von Verhalten bedarf zusätzlicher Elemente. In unseren Projekten haben wir gelernt, dass Fachmodelle mittlerer Komplexität ohne Weiteres mit diesem Ansatz umsetzbar sind und vor allem zu effizienten, kompakten Codes führen – unerlässlich in eingebetteten Systemen. Im Abschnitt *Skizzierung der Lösung* werden DomainComponents näher beschrieben.

Welche Strukturen dienen zum Informationsfluss zwischen den Schichten und zum Austausch mit dem Rest der Anwendung?

Ergebnisse von Datenbank-Queries, Werte für schreibende Zugriffe und Parametersätze müssen innerhalb des Domain Layers und zwischen dem Domain Layer und anderen Schichten ausgetauscht werden. Diese Informationen müssen darüber hinaus kompatibel mit den Datenstrukturen aus dem Rest der Anwendung sein (z. B. strukturierten Messwerten von Sensoren), um die Integration der Datenbankanbindung leicht handhabbar zu machen. Wir empfehlen die Definition einer generischen Struktur (der *AttributeTable*, Erläuterungen folgen im nächsten Abschnitt), die Ergebnismengen und Parametersätze für SQL-Statements in eine Abstraktion zusammenführt. Sie eignet sich auch ohne Weiteres für die Adaption an strukturierte Messwerte, Parametersätze von Geräten, Display-Inhalte u. v. m. Die Stärken des mengenbasierten Ansatzes von SQL werden bei dieser Umsetzung genutzt. Entitäten aus dem Fachmodell spiegeln sich in den Zeilen der generischen Tabellen wider.

Skizzierung der Lösung

Wir betrachten den Lösungsansatz auf konzeptioneller Ebene und möchten einen Überblick über die Zuständigkeiten der wesentlichen Schlüsselabstraktionen vermitteln.

SQLConstructor: Ein ConcreteSQLConstructor kapselt ein SQL-Statement, oder präziser ausgedrückt die Konstruktionsvorschrift eines SQL-Statements. Es kann sich dabei um ein SQL-Statement mit fester Struktur handeln, das nur an einigen Stellen parametriert ist, ähnlich einem Prepared Statement. Eine weitergehende Möglichkeit ist die einer programmatisch hinterlegten Vorschrift, mit deren Hilfe das SQL-Statement in beliebiger Form, von Parametern gesteuert, zur Laufzeit aufgebaut werden kann. Diese Variante ist z. B. bei der Umsetzung von sehr flexiblen Suchkriterien notwendig, die sich in komplexen SQL-WHERE-Klauseln widerspiegeln. Ein ConcreteSQLConstructor ist trotz seiner vielfältigen Möglichkeiten nach

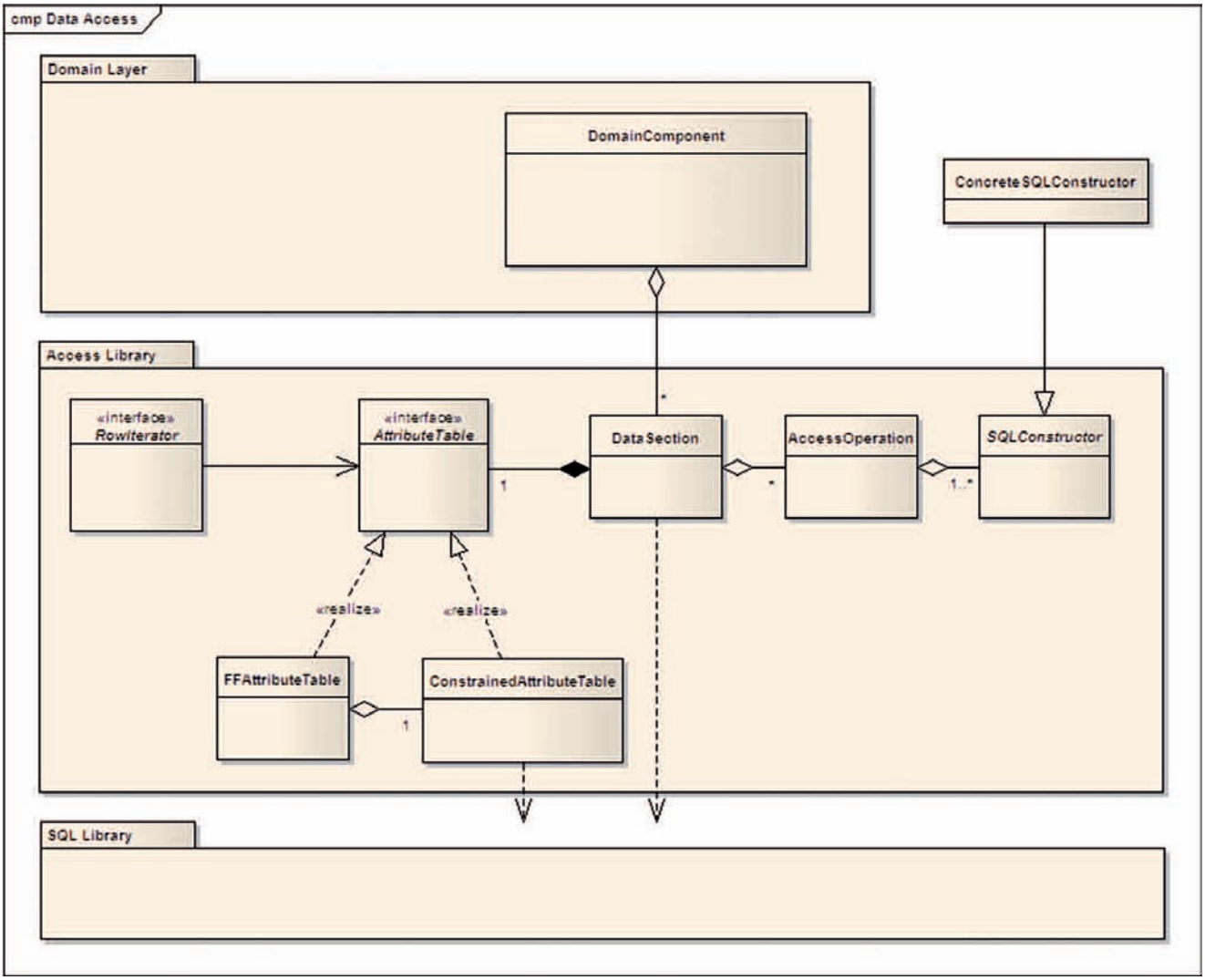

Abb. 1: Konzeptionelle Sicht

einem definierten Schema aufgebaut und damit möglicher Gegenstand der Katalogisierung, Generierung von Code und Wiederverwendung. Die abstrakte Klasse SQLConstructor aus der Access Library liefert eine Basis, der Anwendungsprogrammierer bzw. der Codegenerator erstellt darauf aufbauend eine Reihe von Concrete SQLConstructors (Template Method Pattern).

AccessOperation: Eine AccessOperation fasst mehrere ConcreteSQLConstructors zusammen. Sie repräsentiert eine Zugriffsoperation auf einen kleinen Datenbankausschnitt und wickelt die nötigen SQL-Statements in einer Sequenz ab. Auf diese Weise können z. B. alle nötigen SQL-UPDATE-Statements automatisiert der Reihe nach zur Aktualisierung einer Mastertabelle und einer zugehörigen Detailtabelle mittels eines einzigen Aufrufs einer Methode ausgeführt werden. Diesem Aufruf werden die benötigten Parameter zur Steuerung der Konstruktion der einzelnen SQL-Statements in Form eines RowIterators (s. u.) mit gegeben.

DataSection: Eine DataSection ist für die Bereitstellung der benötigten Zugriffsoperationen auf einen kleinen Datenbankausschnitt (typischerweise eine bis drei Tabellen bzw. ein Datenbank-View) zuständig, ist also eine Zusammenfassung von AccessOperations. Die Ergebnisse lesender Zugriffsoperationen werden in einer AttributeTable (siehe rechts) hinterlegt. Eine DataSection ist eine angemessene Abstraktion der SQL-Ebene und zum Teil auch des technischen Datenbankmodells. Sind alle SQLConstructors und deren Beziehungen zu den DataSections einmal definiert, nutzt der Anwendungsprogrammierer für den Datenbankzugriff nur noch DataSections, kann sich in seinem Denken also auf der entsprechenden Abstraktionsebene bewegen.

AttributeTable: Eine AttributeTable ist die bereits zu Anfang des Abschnittes genannte konzeptionelle Struktur zum Austausch von Informationen. Es handelt sich dabei um eine dynamische, zweidimensionale Tabelle, deren Elemente benannte, typisierte Attribute sind. Sie kann Ergebnisse eines oder mehrerer SQL-SELECTs aufnehmen. Eine Zeile einer AttributeTable kann darüber hinaus auch die Rolle eines Parametersatzes übernehmen, mit dem ein SQLConstructor versorgt wird. Die konzeptionelle Struktur AttributeTable sollte in mindestens zwei konkreten Ausprägungen in der schlanken Access Library zur Verfügung gestellt werden: Die ConstrainedAttributeTable ist ein einfacher, hoch performanter Adapter. Mit ihm wird die konkrete Datenstruktur, die von der SQL-Library des Datenbankherstellers zur Speicherung von Ergebnismengen vorgesehen ist, auf die von der Access Library geforderten Schnittstelle für Attribute

Tables angepasst. Die FFAttributeTable (full featured AttributeTable) ist mithilfe einer eigenständigen Datenstruktur implementiert und speichert zusätzlich zu Attributen unter anderem auch den Modifizierungszustand der einzelnen Zeilen. In einigen Anwendungsfällen lohnt es sich hinsichtlich verbesserter Performance, weitere, zugeschnittene Konkretisierungen der AttributeTable zu implementieren, um dem Anwendungsprogrammierer vielfältige Möglichkeiten zur Optimierung des Lade- und Speicherverhaltens von Daten zu geben.

RowIterator: Ein RowIterator ist ein Iterator auf eine AttributeTable, mit dem diese Zeile für Zeile angesprochen werden kann. Mit einem RowIterator können Parametersätze (repräsentiert durch eine Zeile einer AttributeTable), Ausschnitte aus AttributeTables (durch Angabe eines Intervalls aus zwei RowIterators) oder vollständige AttributeTables referenziert werden. Der RowIterator ist somit ein idealer Kandidat, um ihn als generischen Parameter durch die Anwendung „herumzureichen". Für den RowIterator sind verschiedene Konkretisierungen vorgesehen, es gibt z. B. einen ConstRowIterator, der lediglich lesenden Zugriff auf die Attribute zulässt.

Die Ergebnisse von Queries werden in einer der DataSection eindeutig zugeordneten AttributeTable gespeichert. RowIterators können demnach sowohl auf Ergebniszeilen verweisen, als auch Parameter für die Konstruktion weiterer SQL-Statements sein. In dieser Doppelrolle liegt die Möglichkeit des Navigierens durch das Fachmodell begründet, nach der eingangs gefragt wurde: Schlüssel, die Ergebnis einer Anfrage sind, können unmittelbar als Suchschlüssel für eine weitere Anfrage genutzt werden.

DomainComponent: Die bisher genannten Bausteine sind in ihrer Konstruktion im Wesentlichen bottom-up, von den gegebenen technischen Verhältnissen getrieben. Sie abstrahieren von SQL, dem technischen Datenbankmodell und der konkreten API des Datenbankherstellers. DomainComponents hingegen werden top-down entworfen, ihre Konstruktion ist fachlich getrieben. Eine DomainComponent spiegelt einen Bereich der Domäne wider. Die Schnittstellenmethoden einer DomainComponent gehen mit fachlichen Dingen um, ein Name einer Schnittstellenmethode wäre beispielsweise findMesswerteBySensorName(), wenn es in dem Bereich der Domäne um Messwerte und Sensoren geht. DomainComponents nutzen eine Reihe von DataSections, wenn sie mit fachlichen Daten umgehen, die persistiert werden müssen. Eine solche DomainComponent wird in ihrer Schnittstelle Methoden zum Zugriff anbieten, wie die oben aufgeführte find-Methode. Die Implementierung nutzt zur Realisierung dieser Zugriffe die AccessOperations der aggregierten Data Sections. Die anwendungsübergreifende Fachlogik, die über reine Zugriffsfunktionalität hinaus geht (in der oben angedeuteten Domäne etwa Berechnungsfunktionen über Messwertlisten), ist ebenfalls in den DomainComponents hinterlegt, kann in Ausnahmefällen zum Teil aber auch feingranularer schon konkretisierten Data Sections hinzugefügt werden.

DomainComponents ihrerseits können sich gegenseitig aggregieren und damit Service Layer bilden, falls das in der Architektur grundsätzlich vorgesehen ist. Wichtig ist unter anderem die Zuständigkeit einer DomainComponent, die Datenintegrität der fachlichen Daten für ihren Ausschnitt des Modells zu gewährleisten. Der Aufbau der Schnittstelle einer DomainComponent unterliegt restriktiven Regeln und sichert damit die Orthogonalität des gesamten Domain Layers. Solche Regeln geben den Namensaufbau der Methoden vor (z. B. find<Fachbezeichnung>By<Kriterium>, check<Integritätsbezeichnung>) als auch Reihenfolge und Typen der Parameter.

Bei der Implementierung der Domain Components und der SQLConstructors muss das eventuell zugrunde liegende Komponentenmodell berücksichtigt werden, denn diese Elemente sind fachspezifisch und Gegenstand von Wiederverwendung in anderen Projekten, z. B. in Nachbarprojekten innerhalb einer Produktlinie.

Das in der Abbildung gezeigte Klassendiagramm soll die Zusammenhänge auf konzeptioneller Ebene illustrieren. Die tatsächliche Umsetzung in eine logische Architektur und konkrete Implementierung hängt von der verwendeten Programmiersprache, den genauen Anforderungen und Randbedingungen und dem (falls vorhanden) eingesetzten Komponentenmodell ab.

Typische Codeausschnitte

Folgende Codeausschnitte in JAVA/C++-ähnlichem Pseudo-Code illustrieren typische Anweisungssequenzen, die in Anwendungen vorkommen.

```
(1)  class SensorMesswertAccess extends DataSection
     {
(2)          defineAccessOperations() {
(3)                  AccessOperation(„QueryMesswertBy
                     SensorName").SqlPattern(
(4)                          SELECT wert FROM messwert
                             WHERE sensor_name=?name?);
(5)                  AccessOperation(„Save") ...
                     …
                     }
     }
.....
(6)  DataSection messwertAccess = new Sensor
     MesswertAccess;
.....
(7)  Range(RowIterator, RowIterator)
     findMesswerteBySensorName(string sensorName)
     {
(8)          RowIterator rowIter =
             createAttributeTable(„name", sensorName);
(9)          messwertAccess.execute(„Query
             MesswertBySensorName", rowIter);
(10)         return Range( messwertAccess.get
             AttributeTable().begin(),
(11)         messwertAccess.getAttributeTable().end() );
     }
```

(1) Die Zeile leitet die Definition einer konkreten DataSection ein. Hier wird eine DataSection, bestehend aus zwei AccessOperations mittels Ableiten definert. Die Funktionen *AccessOperation* und *SqlPattern* sind Teil der zur Verfügung gestellten Library.

(3) Diese Zeile zeigt ein einfaches Beispiel: Ein SELECT-Statement, das über das Attribut *name* parametriert wird. Man erkennt dies an den umschließenden Fragezeichen um *name*.

(6) In einer DomainComponent (Definiton nicht dargestellt) werden DataSections instanziiert, im Beispiel *SensorMesswertAccess*.

(7) Die Methode findMesswerteBySensorName() ist Element einer DomainComponent. Sie erwartet als Parameter den Namen eines Sensors und gibt als Ergebnis sämtliche Messwerte des betreffenden Sensors zurück. Die entsprechenden Datensätze werden durch zwei RowIterators referenziert.

(8) Der Parameter *name* wird mithilfe einer temporären AttributeTable zugänglich gemacht.

(9) Die AccessOperation *QueryMesswertBySensorName* wird ausgeführt. Dies beinhaltet die Konstruktion des SQL-Statements, die Ausführung des Statements in der Datenbank-Engine und die (virtuelle) Übertragung der Ergebnisse in die AttributeTable.

(10-11) Anfangs- und Enditerator der soeben gefüllten AttributeTable werden zurückgegeben.

Erweiterungsmöglichkeiten

- Ein ConcreteSQLConstructor kann mit einer Vorbedingung versehen werden, die letztlich über die tatsächliche Ausführung des SQL-Statements entscheidet. Kombiniert mit der Möglichkeit SQL-Statements in Sequenzen zusammenzufassen und dem Zugriff auf den Modifizierungszustand von Attributen können Regeln formuliert werden, die dafür sorgen, dass bei schreibenden Zugriffen nur die nötigen Statements unter Einhaltung der referentiellen Integrität in einer Einheit aufgerufen werden. Einmal programmiert, läuft der gesamte Speicherungsvorgang für den Benutzer von DataSections transparent ab.
- Der Ansatz kann zusätzlich eine Unterstützung für Transaktionen vorsehen. Dazu wird in die Access Library eine Klasse *TransactionScope* integriert, die eine Schnittstelle zur Eröffnung einer neuen bzw. Nutzung einer bereits laufenden Transaktion anbietet. Im Application Layer werden Instanzen dieser Klasse erzeugt und einigen Methoden der DomainComponents mitgegeben. Dort wird entschieden, ob eine neue Transaktion benötigt wird, ob eine bereits bestehende genutzt wird u. ä. Im Falle eines Rollbacks helfen Snapshots der AttributeTables, alte Zustände wiederherzustellen. Selbstverständlich deckt dieses Vorgehen nicht alle Aspekte des komplexen Themas „Transaktionen“ ab (siehe z. B. [Bie10]).
- Die Architektur kann auch auf die Anbindung eines HMI (Human Machine Interface, z. B. eine grafische Benutzeroberfläche) erweitert werden. Wir sehen dabei spezielle Elemente vor, sogenannte *HMIMapper*, die die Übertragung fachlicher Inhalte von AttributeTables in Displays und von Eingabegeräten in AttributeTables übernehmen.
- Die vorgestellte Lösung erlaubt durch ihre Struktur und ihre Systematik den Einsatz von MDSD (Model Driven Software Development) zur Generierung eines großen Teils des Codes.

Fazit

Speicher- und Laufzeiteffizienz und auch die Programmierung mit der Sprache C muss nicht im Widerspruch mit „guter“ Architektur und fachlich getriebenem Schnittstellenentwurf stehen. Der vorliegende Artikel zeigt dies bzgl. der Anbindung von relationalen Datenbanken. Wir haben diese Lösung erfolgreich mehrere Male bei unseren Kunden eingesetzt, z. B. im Bereich der Energiewirtschaft. Wir unterstützen unsere Kunden, die zum Teil sehr anspruchsvolle Anforderungen bzgl. Performance und Footprint haben, bei allen Aspekten der System- und Softwareentwicklung. Dabei spielt das Thema Architektur unserer Erfahrung nach eine ganz wesentliche Rolle. Wenn Sie weitergehende Informationen wünschen - ich freue mich über Ihre Kontaktaufnahme! ■

Referenzen

[Fow03] Fowler, Martin: Patterns für Enterprise Application-Architekturen, mitp, 2003

[Amb03] Ambler, Scott: Agile Database Techniques, Wiley & Sons, 2003

[Hib] Hibernate: www.hibernate.org

[Bie10] Bien, Adam: Enterprise Architekturen, entwickler.press, Neuauflage 2010