Transparente Programmierung der Anwendungslogik durch attributgesteuerte Konfiguration von Datenbanksperren

Stefan Sarstedt, Alexander Raschke, Jens Kohlmeyer

Fakultät für Informatik
Abteilung Programmiermethodik und Compilerbau
89069 Universität Ulm
{stefan.sarstedt, alexander.raschke, jens.kohlmeyer}@informatik.uni-ulm.de

Abstract: Informationssysteme werden oftmals in einer Schichtenarchitektur realisiert, die Datenbank- und Anwendungscode zur besseren Übersicht, Wartung und Verteilung trennt. Dennoch ist es nicht möglich, den Anwendungscode vollkommen frei von Datenbankaspekten zu halten, da oftmals auch die Synchronisation auf dieser Ebene erfolgen muss. Um eine vollständige Trennung von Synchronisation und Anwendungslogik zu erreichen – und damit die Logik übersichtlicher zu gestalten –, schlagen wir die Verwendung von Metadaten in Form von Attributen vor, mit deren Hilfe der Code annotiert wird. Ein entsprechendes Persistenz-Framework kann dadurch zur Laufzeit sein Verhalten anpassen und die geforderten Synchronisationsaufgaben übernehmen. Unser Ansatz wurde in einer umfangreicheren Anwendung validiert.

1 Einleitung

Beim Entwurf moderner Informationssysteme kommt häufig eine Schichtenarchitektur zum Einsatz. Dabei wird zur besseren Verteilung und Wartung die Anwendungslogik vom übrigen Code (Oberfläche und Datenbankzugriff) getrennt. Üblicherweise werden in einer separaten Schicht allgemein verwendbare Geschäftsklassen zur Verfügung gestellt, die in verschiedenen Anwendungsszenarien verwendet werden können [Ev04]. Um die Implementierung der Persistenz dieser Geschäftsklassen möglichst transparent zu gestalten, können diverse Design Patterns verwendet werden (siehe z. B. *Active Record* oder *Data Mapper* aus [Fo03]).

Zur korrekten Behandlung von nebenläufigen Zugriffen auf die Geschäftsklassen ist die Verwendung von geeigneten Synchronisationsmechanismen notwendig. Deren Semantik ist dabei eher auf Ebene der Anwendungslogik angesiedelt, denn nur dort kann entschieden werden, welche Zugriffe auf Geschäftsklassen sich für einen speziellen Anwendungsfall gegenseitig ausschließen. Andererseits findet die Implementierung der Synchronisation häufig mit Hilfe von Datenbankmechanismen – durch geeignetes Setzen von Sperren – statt, um die korrekte Nebenläufigkeit mit weiteren externen Anwendungen sicherzustel-

len. Dies resultiert allerdings in einer Mischung von Anwendungslogik und Synchronisationsanweisungen, was die Lesbarkeit und Wartung des Codes erschwert.

Im Rahmen dieser Arbeit soll eine neue Lösung vorgestellt werden, die Synchronisationsangaben als Metainformationen in Form von Attributen verwendet, wie sie das Microsoft .NET Framework (und auch Java in der neuesten Version 1.5) anbietet. Diese werden zusammen mit der Methodenimplementierung definiert und wirken sich auf die in ihr implementierte Anwendungslogik aus, welche dadurch übersichtlich bleibt.

Der Aufbau dieser Arbeit ist wie folgt: In Abschnitt 2 gehen wir zunächst auf die Schichtenarchitektur von Informationssystemen und deren konkrete Gestaltung in unserem Beispiel ein. Dort wird ebenfalls die durch die Anwendung geeigneter Design Patterns erreichte Transparenz der Persistenz beschrieben. Probleme durch konkurrierenden Zugriff sowie unser Lösungsansatz zur Synchronisation werden in Abschnitt 3 vorgestellt und danach in Abschnitt 4 mit anderen Arbeiten verglichen. In Abschnitt 5 erfolgt eine Zusammenfassung.

2 Transparenz der Anwendungslogik

Um die bei Einbeziehung von Nebenläufigkeit auftretenden Probleme klarer darstellen zu können, verwenden wir ein Online-Anmeldesystem für Kurse als Beispielapplikation. Die Studenten können über eine Web-Schnittstelle die Veranstaltung, zu der sie sich anmelden möchten, auswählen und sich nach Überprüfung spezieller Einschränkungen (z. B. maximale Teilnehmerzahl, bestimmte zeitliche Ausschlüsse) dafür registrieren.

2.1 Schichtenarchitekturen von Informationssystemen

Wie die meisten neueren Anwendungen wurde auch unsere Applikation gemäß der unter anderem in [Ev04] und [BMR97] vorgeschlagenen „Schichten-Architektur“ entworfen (vgl. Abb. 1). An unterster Stelle findet sich dabei die Datenbank, in der alle Daten der Applikation gehalten werden. Um die Konvertierung der zumeist relationalen Datenmodelle in die entsprechenden Objekte der Klassenhierarchie der Anwendung kümmert sich die „Datenbankschicht”. In der sogenannten „Geschäftslogikschicht“ wird die die gesamte Domäne betreffende Logik realisiert. Die „Applikationsschicht” hält die speziell für diese Anwendung erforderliche Logik vor und gibt die Daten an die „Oberflächenschicht“ weiter, welche diese entsprechend aufbereitet darstellt. Prinzipiell sollten dabei die einzelnen Ebenen nur mit den unmittelbar angrenzenden über entsprechende Schnittstellen kommunizieren. Nur durch diese Kapselung kann eine gewisse Transparenz und Austauschbarkeit gewährleistet werden. Um z. B. aus einer Desktop-Applikation eine Web-Anwendung zu erstellen, genügt es, die Oberflächen-Ebene auszutauschen.

Diese allgemeine Schichtenarchitektur wurde für unser System noch angepasst: Da die Geschäftslogik in diesem Beispiel nicht komplex ist (Anlegen, Lesen, Ändern und Löschen) bietet sich die Verwendung des *Active Record*-Patterns aus [Fo03] an, wodurch die

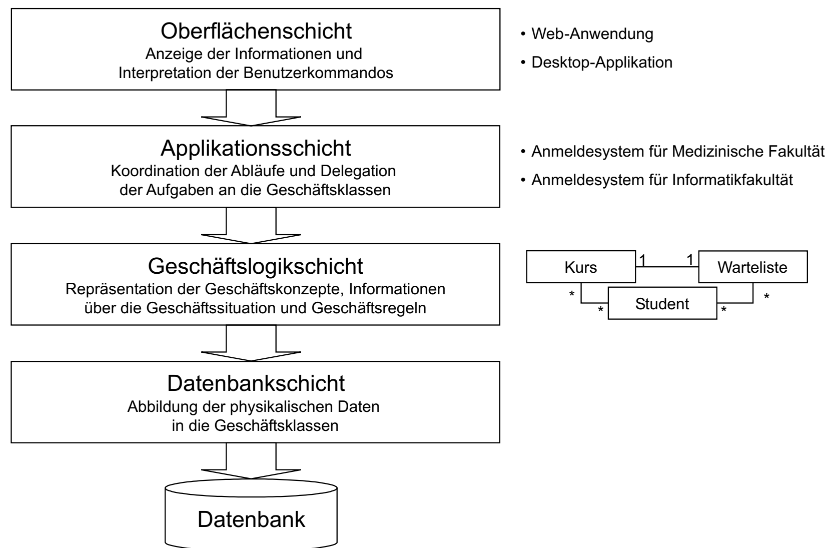

Abbildung 1: Schichtenarchitektur nach [Ev04]

Datenbankaufrufe direkt in der Geschäftslogikschicht stattfinden. Dort werden somit die Daten verwaltet, allerdings keinerlei Anwendungslogik implementiert. Da es für verschiedene Fakultäten bzw. Kursarten verschiedene Anmeldesysteme gibt (mit/ohne Warteliste, Gruppen-/Einzelanmeldung, usw.), werden für diese unterschiedlichen Systeme entsprechend verschiedene Algorithmen benötigt. Diese werden in der Applikationsschicht in jeweils eigenen Anwendungen gekapselt. Im folgenden Abschnitt wird nun zunächst die Realisierung der transparenten Persistenz auf Geschäftslogikebene dargestellt.

2.2 Transparente Persistenz

Unser Ziel ist es, die Anwendungslogik vollkommen frei von Persistenzcode zu halten. Es wäre z. B. wünschenswert, wenn ein konkretes Anmeldeverfahren in der Applikationsschicht folgendermaßen programmiert werden könnte:

```
void Anmelden (Student student, Kurs kurs) {
  if (kurs.Teilnehmer.Count < kurs.MaxTeilnehmerzahl)
    kurs.Teilnehmer.Add(student);
  else
    Fail("Kein freier Platz mehr im Kurs");
}
```

Um dies zu erreichen, haben wir das Verhalten der Klassen der Geschäftslogikschicht (`Kurs`, `Student` und `Warteliste`) entsprechend verändert: Durch Anpassung der

Standard-`ArrayList`-Klasse von .NET und Verwendung von Properties (`Get`- und `Set`-Methoden) wurde erreicht, dass z. B. die Anweisung `kurs.Teilnehmer.Count` die Information über die Anzahl der Kursteilnehmer direkt aus der aktuellen Datenbanktabelle holt:

```
class TeilnehmerFuerKursCollection : ArrayList {
  int kursId; // wird bei Init der Collection gesetzt
  ...
  public override int Count {
    get {
      return (int)ExecSQL("SELECT COUNT(*) FROM "
        + "KursTeilnehmer WHERE kursId=" + kursId);
    }
  }
}
```

Genauso ist nach dem Aufruf von `kurs.Teilnehmer.Add(student)` der Student bereits in der Datenbank eingetragen, womit die zugrundeliegende Persistenz nicht in der Anwendungslogik sichtbar ist. Frameworks für Objektpersistenz (wie z. B. JDO [JDO]) bieten hierfür ähnliche Mechanismen an, haben jedoch auch einige Defizite bzgl. der vollständigen Transparenz (siehe Abschnitt 4).

Der Aufwand zur Erstellung des Frameworks ist nicht unerheblich, der Code könnte aber aufgrund des systematisch strukturierten Aufbaus auch automatisch aus einem UML-Klassendiagramm generiert werden (MDA [KWB03]). Dies würde darüberhinaus die Portierung auf andere Plattformen wie z. B. Java vereinfachen.

3 Nebenläufigkeit und Synchronisation

Um die Probleme der Synchronisation zu erläutern, betrachten wir den folgenden Codeausschnitt aus der `Anmelden`-Prozedur, der für die Kursanmeldung in unserer Beispielanwendung zuständig ist:

```
if (kurs.Teilnehmer.Count < kurs.MaxTeilnehmerzahl)
  kurs.Teilnehmer.Add(student);
```

Wenn sich nun beispielsweise zwei Kursteilnehmer gleichzeitig anmelden wollen, kann eine Situation entstehen, bei der beide zur `Teilnehmer`-Liste hinzugefügt (und damit in die Datenbank eingetragen) werden, auch wenn der Kurs nur noch einen freien Platz enthält. Dies passiert, wenn die Bedingung der `if`-Abfrage von beiden Anmelde-Threads quasi parallel ausgeführt wird und somit beide zu dem Schluss kommen, es sei noch ein Platz im Kurs frei. Dies ist eine Art “Lost-Update”-Problematik [GR92] auf Listenebene.

Damit nun die durch parallele Kursanmeldungen auftretende massive Nebenläufigkeit in den Griff zu bekommen ist, müssen geschickte Synchronisationsmechanismen eingesetzt werden. Hierfür sind mehrere Möglichkeiten denkbar:

1. *Gegenseitiger Ausschluss der Applikationslogik*
   In der Applikationslogik müssen an entsprechenden Stellen gegenseitige Ausschlüsse der Methoden definiert werden (z. B. mit Hilfe von `lock` in .NET bzw. `synchronized` in Java für unsere `Anmelden`-Prozedur):

   ```
   void Anmelden (Student student, Kurs kurs) {
     lock(this) { ... }
   }
   ```

   Dies wäre zwar sehr performant, in unserem Fall allerdings undenkbar, da verschiedene Anmelde-Applikationen auf die gleiche Datenbank zugreifen können sollen, wodurch die Synchronisation mit Hilfe von Datenbanktechniken (Sperren) erfolgen muss.

2. *Verwendung von Datenbank-Transaktionen*
   In unserem Fall muss bei der Abfrage der aktuellen Anzahl der Kursteilnehmer eine Sperre bis zum Transaktionsende auf die entsprechende Tabelle gesetzt und somit jeder weitere Anmelde-Thread daran gehindert werden, dieselbe Abfrage zu machen, bevor der erste Thread seine Transaktion beendet hat. Dies wäre möglich

   a) mit impliziten DBMS Sperren durch einen hohen Isolationslevel.
      Durch Verwendung eines entsprechend hohen Isolationslevels (z. B. *Serializable* [GR92]) kann zwar eine korrekte Abarbeitung sichergestellt werden, die Parallelität wird jedoch auch an Stellen eingeschränkt, an denen es nicht nötig wäre, weil dann auf *allen* in der Transaktion verwendeten Objekten Lese- und Schreibsperren gesetzt werden.

   b) mit expliziten DBMS Sperranweisungen in der Applikationslogik.
      In der Applikationslogik werden die notwendigen Datenbanksperren innerhalb einer Transaktion mit niedrigerem Isolationslevel direkt gesetzt. Dazu müssen datenbankspezifische Operationen in den Code eingefügt werden, wodurch die gewünschte logische Trennung der einzelnen Schichten nicht mehr gegeben wäre:

      ```
      kurs.Teilnehmer.Lock();
      if (kurs.Teilnehmer.Count <
            kurs.MaxTeilnehmerzahl)
        kurs.Teilnehmer.Add(student);
      ```

      Diese Sperre muss bis zum Ende der Transaktion aufrecht erhalten werden, damit konkurrierende Zugriffe bis zur Beendigung der aktuellen Anmeldung blockiert bleiben. Das DBMS gibt diese manuell gesetzten Sperren dann automatisch frei.

      Problematisch ist bei diesem Ansatz, dass diese Mischung aus Anwendungs- und Synchronisationscode das Programm schwerer les- und wartbar macht. Selbst wenn die Geschäftslogikschicht entsprechende Methoden (wie im obigen Beispiel) zur Verfügung stellen würde, würde dies zumindest die geforderte Transparenz der Persistenz zunichte machen.

3. *Anmeldelogik auf Ebene der Geschäftslogik oder Datenbankschicht*
   Es wäre schließlich auch denkbar, die entsprechenden Abschnitte der Anwendungslogik auf die Geschäftslogikschicht oder sogar in die Datenbankschicht (z. B. mit „Stored Procedures") herunter zu ziehen. In diesem Fall findet dort die explizite Synchronisation (wiederum durch DBMS Sperranweisungen) statt.

   Dies würde aber die Schichtenarchitektur komplett auflösen und die Wiederverwendbarkeit der einzelnen Teile (insbesondere der Geschäftslogik für mehrere separate Anwendungen) wäre unmöglich, da hier nun *alle* Arten von Anmeldungen implementiert werden müssten. Jedes neue Anmeldeverfahren müsste in der Geschäfts- und nicht in der Applikationslogik realisiert werden.

Um dennoch die Kombination von Synchronisation mit einer transparenten Applikationslogik zu ermöglichen, setzt unsere Lösung auf die Angabe von Sperrhinweisen in Form von Metadaten, die mit Hilfe der typisierten Attribute des Microsoft .NET Frameworks [BB03] umgesetzt wurden. Attribute können beispielsweise vor Methodenimplementierungen eingefügt und zur Laufzeit per Reflection abgefragt werden, um den Programmablauf zu beeinflussen. Für unseren Zweck definieren wir ein eigenes Attribut vom Typ `Lock`, welches als Argument die zu sperrende Klasse bekommt. Dieses Attribut kann nun zu dem Anwendungscode vor der Methodenimplementierung hinzugefügt werden, beispielsweise für die Funktion `Anmelden`:

```
[Lock(typeof(Teilnehmer))]
void Anmelden (Student student, Kurs kurs) { ... }
```

Diese Anweisung weist den in dieser Methode benutzten Code des Persistenz-Frameworks an, vor dem Zugriff auf die `Teilnehmer`-Liste des Kurses die entsprechende Datenbanktabelle zu sperren, um konkurrierenden Zugriff darauf zu unterbinden. Hierzu wird bei Bedarf ein DBMS-spezifischer Sperrbefehl generiert und vor Initiierung der impliziten Datenbankabfrage in `Count` ausgeführt.

```
class TeilnehmerFuerKursCollection : ArrayList {
  ...
  public override int Count {
    get {
      if (MustBeLocked(typeof(Teilnehmer)) == true)
        LockTable("KursTeilnehmer");
      return (int)ExecSQL(...); // wie oben
    }
  }
}
```

Die Methode `MustBeLocked` prüft zunächst, ob ein entsprechendes Sperrattribut in der aktuellen Methode der Anwendungslogikschicht (siehe z. B. `Anmelden`) für diese Tabelle definiert ist. Dazu wird der Aufrufstack nach einem `Lock`-Attributvorkommen durchsucht dessen Argument dem als Parameter von `MustBeLocked` angegebenen entspricht.

```
public bool MustBeLocked(Type type) {
  StackTrace stackTrace = new StackTrace(false);
  object[] attrs;
  for(int i = 0; i < stackTrace.FrameCount; i++) {
    StackFrame f = stackTrace.GetFrame(i);
    attrs = f.GetMethod().GetCustomAttributes(false);
    foreach(Attribute attr in attrs) {
      if ((attr as LockAttribute).TableToLock == type)
        return true;
    }
  }
  return false;
}
```

Die Methode `LockTable` generiert dann einen DBMS-abhängigen Sperrbefehl, der vor der eigentlichen Abfrage der Teilnehmerzahl in `Count` ausgeführt wird:

```
public bool LockTable(string table) {
  ExecSQL("SELECT * FROM " + table + " WITH (TABLOCKX)");
}
```

Diese Anweisung erzeugt eine Tabellensperre für den MS SQL Server. Bei Oracle würde die entsprechende Syntax `LOCK TABLE <tabelle>` lauten. Da im Falle des MS SQL Servers die eigentliche Sperranweisung `WITH (TABLOCKX)` sogar Teil einer `SELECT`-Abfrage ist, kann sie alternativ direkt in die Abfrage der Anzahl der Kursteilnehmer integriert werden. Dadurch kann eine SQL-Anweisung entfallen.

In einigen Fällen sind komplette Tabellensperren nicht angemessen, da sie nur eine grobe Kontrolle über die Synchronisation erlauben. Wir führen deshalb zwei verschiedene Arten von Sperren ein, sogenannte "SingleLocks" (welche nur einzelne Tabellenzeilen sperren) und "MultiLocks" (welche ganze Tabellen sperren). Dies ermöglicht uns beispielsweise eine Aktualisierung der maximalen Teilnehmerzahl eines Kurses (`MaxTeilnehmerzahl`-Attribut der Klasse `Kurs`) zu verhindern, solange sich noch jemand anmeldet. Hierzu muss die Methode `LockTable` entsprechend angepasst werden.

4 Diskussion

Neben dem von uns vorgestellten Ansatz gibt es eine Reihe von Arbeiten, die sich mit der transparenten Implementierung von Nebenläufigkeit und/oder Persistenz in einer verteilten Anwendung auseinander setzen. Auch diese Ansätze versuchen, die Anwendungslogik vom Persistenzcode zu trennen und diesen transparent in den Code einzubinden.

Hierzu bedienen sich die Arbeiten u.a. der Technik der Aspektorientierten Programmierung (AOP [KLM+97]). Diese bietet sich an, um Aspekte wie z. B. technische Anforderungen vom fachlichen Code zu trennen. Im Folgenden werden Arbeiten vorgestellt, die sich speziell mit den Aspekten Persistenz und/oder Nebenläufigkeit befassen.

In [SLB02] werden die Erfahrungen beschrieben, die bei der Neuimplementierung einer bereits existierende Java-Anwendung mit AspectJ gemacht wurden. AspectJ ist eine aspektorientierte Erweiterung von Java. Hierzu wurde der für die Nebenläufigkeit und Persistenz notwendige Java-Code entfernt und mittels Aspekten neu implementiert. Die Autoren kommen zu dem Schluss, dass AspectJ zwar geeignet ist, Aspekte wie die Persistenz und die Nebenläufigkeit von der eigentlichen Anwendungslogik zu separieren, allerdings traten hierbei Probleme auf: so ist eine völlige Transparenz bei der Umsetzung mit AspectJ nicht zu erreichen, da z. B. Namenskonventionen eingehalten werden müssen, um die Aspekte in den Anwendungscode einzuweben. Darüber hinaus kann es zu unerwünschten Seiteneffekten kommen. Schwierigkeiten bereitet hierbei vor allem die Gleichzeitigkeit von Persistenz und Nebenläufigkeit, da z. B. für die Nebenläufigkeit manche persistente Vorgänge geändert werden müssen. Diese beiden Aspekte können folglich nicht vollständig unabhängig voneinander betrachtet werden [SLB02]. Von ähnlichen Problemen berichten auch die Autoren von [RC03], die ebenfalls eine möglichst transparente und wiederverwendbare Persistenz unter Einsatz der aspektorientierten Programmierung erzielen möchten. In beiden Arbeiten wird daher die Nebenläufigkeit lediglich oberflächlich bzw. überhaupt nicht berücksichtigt. Vor allem bei der beispielhaften Umsetzung der Anwendung in AspectJ aus der Arbeit [SLB02] wurden Einschränkungen gemacht, die den Einsatz nur in einer Umgebung mit nicht-konkurrierendem Zugriff erlauben, weswegen auch keinerlei Datenbanksperren gesetzt werden.
In [KG02] wird der Fragestellung nachgegangen, inwieweit sich im Allgemeinen Nebenläufigkeit über Transaktionen in Form von Aspekten realisieren lässt. Die Autoren ziehen den Schluss, dass Nebenläufigkeit und Aspekte nur auf Ebene der Synchronisationsmechanismen sinnvoll miteinander verknüpfbar sind. Dieses Vorgehen sollte jedoch mit gebotener Vorsicht verwendet werden, da sich bei einer Änderung der Anwendungslogik unter Umständen der die Synchronisation behandelnde Code in den entsprechenden Aspekten angepasst werden muss. Dies ist insbesondere deswegen problematisch, da sich der Aspektcode normalerweise in eigenen Dateien befindet. Diesem Problem wird in unserer Lösung dadurch begegnet, dass sich die Attribute unmittelbar vor den Methoden befinden.

Ein anderer Ansatz, um einen transparenten persistenten Anwendungscode zu erreichen, wird in [AJ98] verfolgt. „PJama“ ist eine Persistenzplattform für Java, die das Prinzip der orthogonalen Persistenz umsetzen soll. Hierfür wird der Java-Interpreter angepasst, so dass alle Datentypen von Klassen über Variablen bis zu Threads persistent gehalten werden können. Allerdings wurde die volle orthogonale Persistenz noch nicht erreicht. Insbesondere wird an einem flexiblen Transaktionsmodell mit konkurrierendem Zugriff gearbeitet.

Eine Spezifikation für ein Persistenz-Framework in Java, die sich mit der transparenten Verwendung von Persistenzcode in einer Anwendung beschäftigt, wird unter [JDO] (Java Data Objects) näher beschrieben. Die Persistenz der Daten wird dem Framework überlassen, der Anwendungsprogrammierer kann sich auf die Anwendungslogik konzentrieren. Allerdings sind Datenbankoperationen nicht vollständig transparent gehalten, da z. B. Verbindungen zur Datenbank explizit in der Anwendungslogik geöffnet und geschlossen werden müssen. Datenbanksperren können im Rahmen von Transaktionen explizit im

Anwendungscode gesetzt werden, es ist aber auch ein implizites Sperren durch Anpassung des Isolationslevels der Transaktionen möglich, was allerdings (wie schon in Abschnitt 3, unter Punkt 2a erwähnt) die an einigen Stellen durchaus gewollte Parallelität unnötig einschränkt. Ähnliches gilt auch für ausgereifte OODBM-Systeme wie beispielsweise ObjectStore [OOD].

Unser Ansatz hat noch Schwächen, die weiterer Überlegungen und Arbeit bedürfen: So werden z. B. durch das transparente Setzen der Sperren über die Attribute im Anwendungscode Sperren unter Umständen früher als nötig gesetzt und dadurch länger (bis zum Transaktionsende) beibehalten. Deswegen ist die Performanz unseres Systems nicht so hoch wie bei einer Anwendung, bei der die notwendigen Tabellen direkt in der Geschäftslogik gesperrt werden. Im Falle massiver Nebenläufigkeit führt dies zu Performanzeinbußen, zu der darüberhinaus auch die bei uns implementierte transparente Persistenz der Daten beiträgt, da momentan noch keine Cachingstrategien implementiert sind.
Für eine Anmeldephase wurde eine Vergleichsimplementierung (ein gekapseltes Anmeldesystem) mit „Stored Procedures" umgesetzt, mit der ein Referenzwert für die Performanz unseres Systems ermittelt werden sollte. In der Datenbankschicht wurden große Teile der Anwendungslogik und explizit die Datenbanksperren für die Nebenläufigkeit realisiert (siehe Abschnitt 3, Punkt 3 der Synchronisationsmechanismen). Aus Log-Dateien konnten wir ablesen, dass dieses System eine etwa vierfach höhere Performanz bei gleicher Last im Vergleich zu unserem Ansatz hatte.
Wir mussten auch feststellen, dass durch die Transparenz bei der Persistenz und der Nebenläufigkeit erhebliche Schwierigkeiten beim Debuggen der Anwendung entstehen. Fehler (z. B. Deadlocks) zu lokalisieren ist bei Nebenläufigkeit in einer verteilten Anwendung im Allgemeinen nicht trivial, wird aber aufgrund des transparenten Codes in unserem System noch erschwert. Von ähnlichen Problemen wird auch in Arbeiten berichtet, die ein System mit der Aspektorientierten Programmierung umsetzen (siehe [SLB02]).
Trotz dieser Schwierigkeiten wird in unserer Anwendung eine transparente Persistenz der Daten erreicht, die auch durch die nötige Nebenläufigkeit und dem daraus resultierenden konkurrierenden Zugriff nicht verletzt wird. Die Datenbanksperren beeinträchtigen in keiner Weise das Persistenz-Framework und sind mit Hilfe der Attribute des Microsoft .NET Frameworks [BB03] transparent im Anwendungscode integriert. Auch Java bietet in der neusten Version 1.5 an, Klassen und Methoden mit Attributen zu versehen, weshalb unser Ansatz nicht auf das .NET Framework begrenzt ist.
Um unsere Lösung zu validieren, wurde ein Anmeldesystems für Praktika und Prüfungen für ca. 1750 Studenten und 200 Veranstaltungen an der Universität Ulm realisiert. In diesem Zusammenhang konnten die zuvor erwähnten Erfahrungen gesammelt und der Ansatz bestätigt werden. Zu Lastzeiten wurden pro Stunde ca. 3600 Anmeldungen bearbeitet.

5 Zusammenfassung

Wir haben in diesem Beitrag gezeigt, wie Metadaten in Form von typisierten Attributen genutzt werden können, Synchronisationsanweisungen deklarativ für Methoden der Applikationsschicht zu spezifizieren. Ein entsprechendes Persistenz-Framework kann diese

Angaben nutzen, um die Synchronisation von Datenbankzugriffen zur Laufzeit zu steuern, indem Datenbanksperren explizit vom Framework gesetzt werden. Die Transaktion kann auf einem niedrigeren Isolationslevel ablaufen und somit eine höhere Parallelität der Anwendung sicherstellen.

Durch die Trennung von Anwendungscode und Synchronisationsanweisungen wird der Code frei von Datenbankaspekten gehalten, was ihn verständlicher und leichter wartbar macht. Hierdurch kann eine vollständige Transparenz der Persistenz erreicht werden. Nachteile des Ansatzes bestehen in der geringeren Performanz im Vergleich zu ausprogrammierten Sperranweisungen (bedingt auch durch die Architektur unseres Persistenz-Frameworks) sowie in der erschwerten Fehlerlokalisierung, beispielsweise im Fall von Deadlocks.

Unser Ansatz wurde in einem Kursanmeldesystem für die Universität Ulm realisiert und im Rahmen einer Anmeldephase für Praktika und Prüfungen im Wintersemester 2003/04 validiert.

Literatur

[AJ98] Atkinson, M. und Jordan, M.: Providing Orthogonal Persistence for Java. In: *ECOOP, Springer-Verlag Lecture Notes in Computer Science 1445*. S. 338–359. 1998.

[BB03] Bock, J. und Barnaby, T.: *Applied .NET Attributes*. Apress. 2003.

[BMR97] Buschmann, F., Meunier, R., und Rohnert, H.: *Pattern-Oriented Software Architecture, Vol.1 : A System of Patterns*. John Wiley & Sons. 1997.

[Ev04] Evans, E.: *Domain Driven Design*. Addison Wesley. 2004.

[Fo03] Fowler, M.: *Patterns of Enterprise Application Architecture*. Addison Wesley. 2003.

[GR92] Gray, J. und Reuter, A.: *Transaction Processing. Concepts and Techniques*. Morgan Kaufmann Publishers. 1992.

[JDO] Java Data Objects, `http://java.sun.com/products/jdo/`.

[KG02] Kienzle, J. und Guerraoui, R.: AOP: Does It Make Sense? The Case of Concurrency and Failures. In: *ECOOP, Springer-Verlag Lecture Notes in Computer Science 2374*. S. 37–61. 2002.

[KLM+97] Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., und Irwin, J.: Aspect-Oriented Programming. In: *ECOOP, Springer-Verlag Lecture Notes in Computer Science 1241*. S. 220–242. 1997.

[KWB03] Kleppe, A., Warmer, J., und Bast, W.: *MDA Explained*. Addison Wesley. 2003.

[OOD] ObjectStore, `http://www.objectstore.net`.

[RC03] Rashid, A. und Chitchyan, R.: Persistence as an Aspect. In: *2nd International Conference on Aspect-Oriented Software Development, ACM*. S. 120–129. 2003.

[SLB02] Soares, S., Laureano, E., und Borba, P.: Implementing Distribution and Persistence Aspects with AspectJ. In: *ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. 2002.