TECHNISCHE
UNIVERSITÄT
WIEN

VIENNA
UNIVERSITY OF
TECHNOLOGY

Web Engineering

Datenbankprogrammierung Teil 1 (Persistenz)

Studienbrief DBPR1

Version 1.2 (Juli 2005)

Karl M. Göschka

Inhaltsverzeichnis

0. Übersicht ... 3
0.1. Lehrziele ... 3
0.2. Lehrstoff ... 3
0.3. Aufgaben, Übungen ... 3
0.4. Voraussetzungen ... 3
0.5. Literatur ... 3
1. Objektorientierung und Persistenz ... 4
1.1. Objekt-Serialisierung ... 7
1.2. Spezifische Persistenz ... 8
1.3. Gekapselter Datenbankzugriff ... 8
1.4. Persistenz-Frameworks ... 10
1.5. Orthogonale Persistenz in objektorientierten Datenbanken ... 11
1.6. Integration des Schichtenmodells mit den Persistenzansätzen ... 12
2. Lehrzielorientierte Fragen ... 14

0. Übersicht

0.1. Lehrziele

Das primäre Lehrziel der Studieneinheit **Datenbankprogrammierung Teil 1 (Persistenz)** ist die Vermittlung des Verständnisses um die Grundprobleme der Kombination von objektorientierter Programmierung mit relationalen Datenbanken.

0.2. Lehrstoff

Dieser Studienbrief gibt einen Überblick über verschiedene Persistenzansätze.

Abschnitte, die mit einem (*) gekennzeichnet sind, dienen der freiwilligen Stoffergänzung und werden nicht geprüft.

0.3. Aufgaben, Übungen

Der letzte Abschnitt enthält eine Sammlung lehrzielorientierter Fragen zur Selbstüberprüfung. Die Beantwortung sollte nach Durcharbeiten des Studienbriefes möglich sein. Treten bei einer Frage Probleme auf, so versuchen Sie diese mit ihren Studienkollegen zu lösen. Ist das nicht möglich, können Sie mich per eMail direkt kontaktieren oder Sie stellen die entsprechende Frage in der nächsten Übungsstunde.

0.4. Voraussetzungen

Der vorliegende Kurs setzt Kenntnisse des Software-Engineering und der Netzwerkdienste voraus, sowie Objektorientierte Methoden.
Die Studieneinheit **Datenbankprogrammierung Teil 1 (Persistenz)** baut auf den Studieneinheiten GRUN, RELAT, SQL1 und SQL2 auf.

0.5. Literatur

[1] Heuer, A.; Saake, G.: „Datenbanken: Konzepte und Sprachen“, International Thomson Publishing, Bonn, 2.Auflage, 2000.

[2] Heuer, A.; Saake, G.; Sattler, K: „Datenbanken kompakt“, mitp, Bonn, 2001.

1. Objektorientierung und Persistenz

Objektorientierung ist Stand der Technik bei der Softwareentwicklung. Dennoch sind relationale Datenbanken[1] Stand der Technik in vielen Bereichen. Aufgrund der Langlebigkeit von Datenbanksoftware wird sich daran auch nicht allzu schnell etwas ändern. Damit bleibt die Frage offen, wie mit der Diskontinuität im Design zwischen der objektorientierten Welt und dem klassischen semantischen Datendesign umgegangen werden soll.

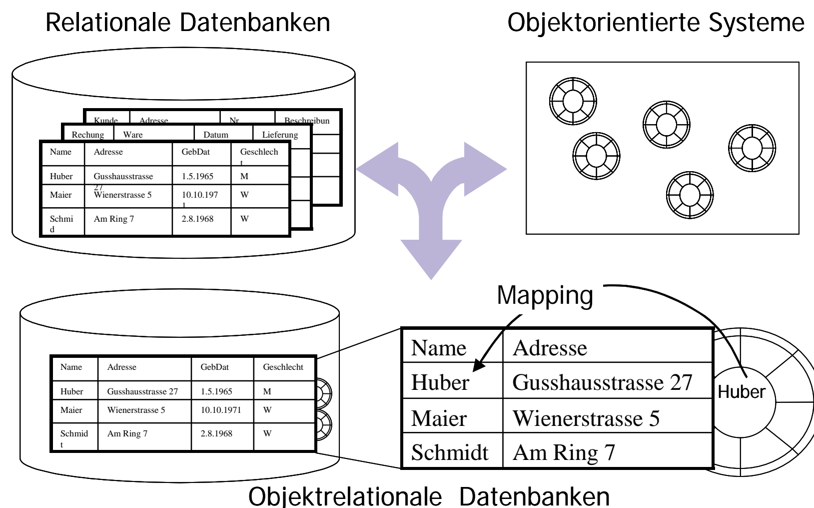

Bild 1.1: ***Objektorientierte Persistenz in relationalen Datenbanken***

Das Grundproblem besteht nun darin, die objektorientierte Struktur auf die relationale Struktur abzubilden (Object Relational Mapping, siehe **Bild 1.1**). Einfache Klassen könnten direkt auf Tabellen – sogenannte Relationen – abgebildet werden, wobei jedes Attribut auf eine Spalte abgebildet wird und eine Instanz auf eine Zeile (Tupel). Der Primärschlüssel der Klasse kann auch als Primärschlüssel in der Datenbank verwendet werden.

Für komplexere Strukturen gibt es folgende strukturell unterschiedliche Ansätze:

1. *Schwache Struktur* mit Dependent-value-Classes: Objekte abhängiger Werte-Klassen werden im Binärformat[2] in einem einzigen Attribut in der Datenbank abgespeichert mit

---

[1] bzw. die objektrelationalen Nachfolger der relationalen Systeme, deren wesentlicher Vorteil darin besteht, dass die Anbindung objektorientierter Software erleichtert wird. Im Datenbankkern handelt es sich bei den meisten Systemen aber immer noch um relationale Datenbanken.

[2] BLOB Binary Large Object

dem Nachteil, dass die Datenbank diese Daten nicht durchsuchen kann und keine Beziehungen auf diese Spalten zeigen kann.

2. *Starke Struktur*: Sind mehrere Klassen miteinander in Beziehung, wird jede Klasse in einer eigenen Tabelle abgelegt. Die Tabellen werden, genauso wie die Klassen, über Attribute in Beziehung gesetzt. Die Datenbankstruktur spiegelt also die Klassenstruktur wider.
3. *Hybride Struktur*: Die Instanzen der Objekte werden zwar serialisiert, die Referenzen werden jedoch in eindeutige Schlüssel-Beziehungen umgeformt, sodass die Strukturinformation zwar enthalten ist, sich jedoch nicht in der relationalen Datenbankstruktur wiederfindet. Besonders interessant in diesem Zusammenhang sind *semi-hybride* Strukturen mit XML.

Darüber hinaus muss man noch unterscheiden, wie stark die Objektstruktur und die Datenbankstruktur voneinander entkoppelt sein sollen (Kapselung und Geheimhaltung). Dies führt zu verschiedenen Ansätzen für die Persistenz von Objekten. Unter **Persistenz** versteht man dabei folgendes: Die Lebensdauer eines persistenten Objektes übersteigt die Lebensdauer der Applikation, die das Objekt erzeugt hat. Das persistente Objektmanagement speichert den Zustand der Objekte auf einem nicht-flüchtigen Medium, sodass das Objekt weiterexistiert, auch wenn die Anwendung beendet wird.

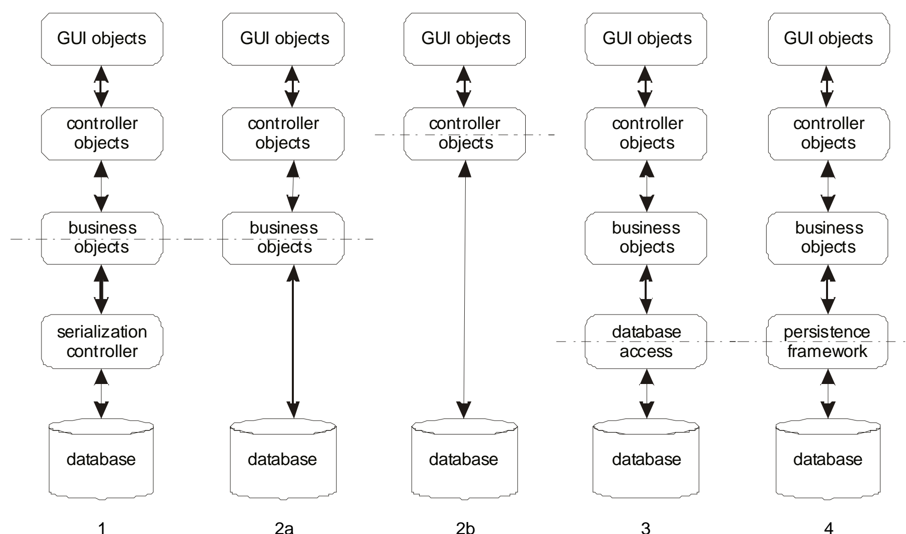

Bild 1.2: ***Verschiedene Ansätze für Persistenz***

Bild 1.2 gibt einen kurzen Überblick über verschiedene Persistenz-Ansätze in Bezug auf die üblichen Schichten bei objektorientierter Software: User Interface Objects, Controller

Objects, die Business Objects, die die Geschäftslogik implementieren (also die eigentliche Funktionalität der Software) und die Objekte für den Zugriff auf die Datenbank. Die strichpunktierte Linie markiert dabei die Lage des Bruches zwischen der objektorientierten und der relationalen Welt. Auf diese Diskontinuität muss von der Analyse über das Design bis hin zur Implementierung besondere Rücksicht genommen werden. Dabei ergeben sich folgende Möglichkeiten:

1. Objekt-Serialisierung: Der gesamte Zustand des Objektes wird in einen linearen Bytestrom serialisiert („*marshalling*"), der dann im File-System oder in einer Datenbank abgelegt werden kann. Problematisch bei diesem Ansatz ist vor allem die starke Vernetzung der Objekte.
2. Jedes Business-Objekt unterhält seine eigene Verbindung zur Datenbank. Wenn die Applikation im Wesentlichen nur aus den klassischen Datenbank-Operationen besteht und es darüber hinaus kaum Funktionalität gibt, dann ergibt sich der Spezialfall (2b), der typisch für Informationssysteme mit sehr dünner Middleware ist.
3. In diesem Fall wird der Zugriff auf die Datenbank in einigen dafür speziell vorgesehenen Controller-Objekten zusammengefasst.
4. Für große und komplexe Projekte empfiehlt sich ein *Persistenz-Framework*. Dabei wird der Datenbankzugriff für die übrigen Objekte völlig transparent, die Struktur der relationalen Datenbank bleibt vor der objektorientierten Welt verborgen.

Während Variante 1 stets schwach oder hybrid strukturiert ist, wird Variante 3 i. Allg. stark strukturiert sein. Lediglich für die Varianten 2 und 3 stehen alle drei Varianten sinnvoll zur Auswahl. Im Sinne der Entkoppelung wird man bei Variante 3 eher zur hybriden Struktur greifen, während man bei Variante 2 – die ja ohnehin nur geringere Entkoppelung bietet – eher die Vorteile der Konsistenzüberwachung durch die Datenbank nutzen und daher zur starken Strukturierung greifen wird.

Alle diese Ansätze haben dennoch ein Problem gemein: die strukturelle Diskontinuität zwischen den Objektbeziehungen im Klassendiagramm und den Relationen zwischen den Entities im ER-Diagramm. Darüber hinaus bewirken manchmal harte Vorgaben an die Performance, dass einige prozedurale Elemente bereits in der Datenbank programmiert

werden müssen, wodurch ein weiterer Bruch im Design herbeigeführt wird[3]. Welchen Weg man auch immer wählt, solange man sich dieses Design-Bruches bewusst ist und stets besondere Sorgfalt darauf verwendet, ist noch nichts verloren.

1.1. Objekt-Serialisierung

Einer der einfachsten Wege, um Objektpersistenz zu erreichen, ist die Serialisierung. Objekt-Serialisierung unterstützt die Umwandlung eines Objektes sowie aller von diesem Objekt erreichbaren Objekte in einen Bytestrom und natürlich umgekehrt die vollständige Rekonstruktion der Objektstruktur aus diesem Bytestrom. Serialisierung kann für Persistenz im kleinen Bereich genutzt werden. Sie wird aber auch für Kommunikation über Sockets, RMI oder auch bei CORBA benutzt. Serialisierung ist bei Java bereits konzeptionell „eingebaut" und in Form von Interfaces verfügbar.

Und natürlich kann man serialisierte Objekte nicht nur über das Netzwerk transportieren oder im Dateisystem ablegen, sondern auch in einer Datenbank speichern. In diesem Fall hat die Datenbank keine Struktur im eigentlichen Sinn, sie dient nur als Lager für die serialisierten Objekte. Der Nachteil dieses Ansatzes ist auch unter dem Namen „Banane-Gorilla-Problem" bekannt: Um – z. B. mittels Schlüssels oder Suchfunktion – auf die Banane zugreifen zu können, muss erst die Gesamtheit aller assoziierten Objekte – der Gorilla – instanziiert werden. Daher kann Serialisierung nur bei solchen Applikationen eingesetzt werden, die nicht allzu oft auf nicht allzu große Datenbestände zugreifen.

Jedenfalls wird von Serialisierung abgeraten bei Applikationen, die

- mehrere hundert Megabyte[4] oder mehr an Daten verwalten: Es muss stets ein ganzer Objektgraph gelesen und geschrieben werden,
- die Objekte oft verändern: gleicher Grund wie oben,
- zuverlässige Persistenz benötigen: Bei einem Systemabsturz während der Serialisierung bzw. während des Dateizugriffes können Daten verloren gehen.

---

[3] Zwar gibt es schon Datenbanken, die die Java Virtual Machine JVM im Datenbankkern integriert haben, um in der Datenbank objektorientiert implementieren zu können. Es bleibt aber abzuwarten, wie der Performance-Vergleich bei großen Systemen ausfällt.

[4] Für relationale Datenbanken ist das immer noch sehr klein. Die größten Datenbanken weltweit haben bereits zweistellige Terabyte gespeichert.

Alternativ gibt es auch die Möglichkeit, die Art der Serialisierung von Objekten vom Programmierer selbst festlegen zu lassen[5]. In diesem Fall könnte man bei der Serialisierung eine SQL-Insert-Anweisung generieren, die dann in der Datenbank ausgeführt wird. Umgekehrt könnte man die Rekonstruktion durch Select-Anweisungen bewerkstelligen. Auf diese Weise kann ein Objekt als ein Tupel in der Datenbank abgelegt werden, die Schlüssel und die übrige persistente Information kann in Form von Attributen gespeichert werden statt nur als Bytestrom. Man kann so auch die Objektreferenzen als Fremdschlüsselbeziehungen abbilden und erhält somit in der Datenbank ein viel strukturierteres Abbild der Objektstruktur als beim reinen Bytestrom. Der wesentliche Nachteil ist, dass man diesen Vorgang spezifisch für jedes einzelne Objekt implementieren muss. Daher eignet sich dieser Ansatz zwar durchaus für größere Datenmengen, aber nur für einfachere Strukturen mit nicht allzu vielen Objekten und einer möglichst einfachen und übersichtlichen Struktur.

1.2. Spezifische Persistenz

Bei diesem Ansatz wird der Designbruch zwischen der objektorientierten und der relationalen Welt weitgehend ignoriert: Jedes Objekt unterhält seine eigene Verbindung zur Datenbank für seine eigenen Zwecke – nicht nur, um seinen eigenen Zustand zu speichern, sondern um alle Datenbank-Operationen auszuführen, die lokal im Objekt benötigt werden. Dieser Ansatz kann für kleine Projekte durchaus sinnvoll sein, aber er bedingt eine starke gegenseitige Abhängigkeit jedes Objektes von der gesamten Datenbankstruktur: Eine Änderung an der Datenbank wird i. Allg. Anpassungen bei vielen Objekten nach sich ziehen. Erweiterungen sind dadurch schwierig, der langfristige Wartungsaufwand überproportional.

Am ehesten kann diese Methode noch sinnvoll eingesetzt werden, wenn die eigentliche Funktionalität der Anwendung sehr gering ist, wenn es sich also um ein Informationssystem handelt, dessen wesentlichste Aufgabe die Durchführung der klassischen Datenbankoperationen (Suchen, Ändern, Einfügen, Löschen) über ein modernes User Interface ist.

1.3. Gekapselter Datenbankzugriff

Bei diesem Ansatz, der sich für Systeme mit komplexerer Middleware besser eignet als der vorangehende, wird der Datenbankzugriff in einigen wenigen Controller-Objekten gekapselt. Damit wird die gegenseitige Abhängigkeit zwischen Datenbank und Applikation auf diese

---

[5] In Java in der Form von „externalizable objects" im Gegensatz zu „serializable objects".

wenigen Objekte reduziert. Jeder dieser Controller ist nun verantwortlich für bestimmte Tabellen der Datenbank und bestimmte Klassen im Klassendiagramm. Die Controller greifen auf die Datenbank zu und instanziieren bestimmte Objekte mit den Daten aus der Datenbank. Umgekehrt schreiben diese Controller die Daten aus den Objekten zurück in die Datenbank. Die anderen Objekte der Applikation können so weitgehend von der Last der Persistenz befreit werden, dennoch muss der Vorgang der Persistierung explizit angestoßen werden. Außerdem muss die Datenbank parallel mit den Methoden des semantischen Datendesigns entwickelt werden. Darüber hinaus liegt die gesamte Verantwortung für die Transaktionskontrolle algorithmisch in den Controllern.

Dieser Ansatz wurde in einem früheren Projekt der Autoren in Form eines Prototyps implementiert, um eine Java-Variante mit einer reinen HTML-Version[6] vergleichen zu können. Als großer Vorteil dieses Ansatzes entpuppte sich die Möglichkeit, objektorientierte Design-Methoden mittels UML anwenden zu können.

In Bezug auf Design und Implementierung konnte also abgeleitet werden:

- Wie bei OO-Design üblich, war die Design-Phase aufwändiger als bei der reinen HTML-Lösung, dafür war die Implementierung einfacher, rascher und geradliniger.
- Es hat sich herausgestellt, dass Container-Objekte notwendig sind, um größere Mengen von Objekten handhaben zu können, v. a. beim Suchen. Diese Container-Objekte beinhalten Suchstrukturen für die Schlüssel und die Referenzen der wichtigsten assoziierten Klassen.
- Die so genannte „lazy instantiation“ – also die Instanziierung von Objekten so spät wie möglich – spart Ressourcen auf der Client-Seite. Oft konnten Container-Objekte einfach um ein, zwei Attribute erweitert werden, um sich die Instanziierung ganzer Objektegraphen ausschließlich für komplexe GUI-Funktionen zu ersparen, ohne Abstriche beim Komfort des GUI in Kauf zu nehmen. Diese erweiterten Container beinhalten genug Information, um die vom GUI benötigten Funktionen abdecken zu können, die eigentlichen Objekte werden dafür gar nicht benötigt. Die Container-Objekte sind damit eine besondere – und zugleich besonders wichtige – Form von Datenbank-Zugriffs-Controllern.

---

[6] Bei der reinen HTML-Lösung werden die HTML-Seiten von prozeduraler Datenbanksoftware generiert. Am Client befindet sich keine Funktionalität.

- Die Kapselung der Datenbankzugriffe im Rahmen der Controller-Objekte bewirkt eine ausreichende Entkoppelung zwischen Datenbank und Applikation für mittelgroße Projekte.
- Um das parallele Design von Datenbank und Klassendiagramm zu unterstützen, gibt es Methoden und Tools, wie ein ER-Diagramm in ein Klassendiagramm und vice versa übergeführt werden kann.

Um die Effizienz dieses Mechanismus weiter zu erhöhen, kann man einen Mechanismus für Preloading und Caching einführen. Damit wird nicht nur Bandbreite des Netzwerks gespart, sondern auch die Antwortzeit des Systems reduziert. Bei Informationssystemen wird – im Unterschied zu Multiprozessorsystemen – üblicherweise keine strikte Synchronität gefordert werden müssen. Falls doch, können aktive Datenbankkonzepte (Trigger) herangezogen werden, um die Cache-Konsistenz zu garantieren.

1.4. Persistenz-Frameworks

Für große Projekte bzw. wenn mehrere ähnlich gelagerte Projekte abgehandelt werden, lohnt sich der Aufwand für ein *Persistenz-Framework*. Persistenz-Frameworks erreichen den höchsten Grad an „Geheimhaltung“[7] zwischen Datenbank und Geschäftslogik. Die Grundidee ist, dass das Geschäftsobjekt ohne Rücksicht auf die Persistenz entwickelt wird. Einige zusätzliche Methoden werden eingeführt, um das Objekt zu instanziieren oder um eine Datenbanktransaktion abzuschließen. Der Datenbankzugriff wird damit weitgehend im Framework gekapselt. Implementierungen verschiedener Frameworks, haben folgende Erfahrungen ergeben:

- Die Implementierung ist wesentlich aufwändiger vom Umfang her, aber nicht unbedingt schwieriger.
- Die Datenbankstruktur kann tatsächlich innerhalb gewisser Grenzen verändert werden, ohne die Geschäftsobjekte anpassen zu müssen. Umgekehrt wirken sich Änderungen des nicht-persistenten Teils der Geschäftsobjekte auch nicht auf die Datenbank aus. Die Entkoppelung funktioniert also, lediglich tiefe strukturelle Änderungen können

---

[7] Geheimhaltung – oder „secrecy“, wie sie im Englischen bezeichnet wird – ist neben der Kapselung eine der wichtigsten Forderungen, um die möglichen Vorteile objektorientierter Systeme auch tatsächlich erreichen zu können. Dies hat nichts mit Sicherheit zu tun – weder im Sinne von Safety noch von Security.

durchschlagen, dies ist aber keine Schwäche des Ansatzes, sondern liegt in der Natur solcher Änderungen selbst.

- Gewisse Schwierigkeiten ergeben sich aus der Tatsache, dass Datenbankoperationen grundsätzlich Mengen von Tupeln zurückliefern. Die getesteten Frameworks bieten für solche Fälle keine systematische Unterstützung, der Programmierer muss sich selbst darum kümmern.
- Es ist nicht möglich, einzelne Attribute für reine GUI-Zwecke abzurufen, es müssen stets die kompletten Objektgraphen aus der Datenbank re-instanziiert werden.
- Die Geschäftslogik kann mittels objektorientierter Entwurfsmethoden (z. B. UML) entwickelt werden, ohne von Persistenz-Überlegungen beeinträchtigt zu werden.
- Für Web-basierte Anwendungen ist es von Vorteil, die Objekte an der Schnittstelle (die Datenbankobjekte) auf der Server-Seite zu lassen (z. B. in Form von Servlets), weil Multi-Tupel-Operationen so performanter ablaufen. Die Geschäftsobjekte können hingegen zwischen Client und Server verteilt werden. Ein Thick Client kann hier Last vom Server und vom Netzwerk nehmen.

Eine ganz andere Methode bietet „Java Relational Binding“, wo aus der Klassenbeschreibung das relationale Schema inklusive der benötigten Methoden für das Lesen und Schreiben der Objekte abgeleitet wird. Damit ist die Diskontinuität im Design zwar gänzlich eliminiert, weil alle benötigten Teile generiert werden können, leider eignet sich diese Methode dadurch aber nur schlecht für die Integration von bereits bestehenden Datenbanken.

1.5. Orthogonale Persistenz in objektorientierten Datenbanken

Der beste Ansatz für objektorientierte Applikationen – zumindest in Hinblick auf das Design – ist die orthogonale Persistenz in objektorientierten Datenbanken. Dabei werden drei Hauptströmungen unterschieden:

1. Erweiterung der objektorientierten Programmiersprachen um Persistenzkonzepte,
2. Erweiterung der relationalen Datenbanken um objektorientierte Kernels,
3. komplett neu entwickelte Systeme, hier v. a. auch im Bereich der Multimedia-Datenbanken, der einen primären Anwendungsbereich für objektorientierte Datenbanken darstellt. Der Grad der Unterstützung von Multimedia-Inhalten bietet ein weiteres Unterscheidungskriterium für objektorientierte Datenbanken.

Die Object Data Management Group ODMG arbeitet permanent daran, diese Strömungen zusammenzuführen und zu normen (ODMG2.0). Ebenso wird auch an einer standardisierten Abfragesprache für objektorientierte Datenbanken gearbeitet (OQL).

Für die Verwendung von Thick Clients und – im Vergleich zu relationalen Datenbanken – sehr lange andauernde Transaktionen wurden spezielle Check-in/Check-out-Mechanismen entwickelt. Einen interessanten Ansatz für orthogonale Persistenz mit Java stellt das Pjama-Projekt der Universität in Glasgow dar. Pjama ist ein persistentes Programmiersystem für Java, welches der Notation orthogonaler Persistenz folgt: Die Objekte werden bei minimalem Aufwand für die Applikationsentwicklung von einer speziellen Laufzeitumgebung persistent gehalten.

1.6. Integration des Schichtenmodells mit den Persistenzansätzen

Bild 1.3 fasst die sinnvollsten Kombinationen aus Client/Server-Überlegungen und Persistenzansätzen zusammen. Dabei kommt es wesentlich darauf an, ob der primäre Zweck mehr bei einem Informationssystem oder bei der Implementierung komplexer Geschäftslogik liegt. Unter Informationssystemen in diesem Zusammenhang werden solche Systeme verstanden, deren wesentliche Aufgabe im Speichern und Auffinden von einfach strukturierten Daten in einer großen Datenmenge besteht. Demgegenüber stehen Systeme wie CAD-Anwendungen oder Spiele mit komplex strukturierten Daten und komplexer Funktionalität. Mitunter kann der Server auch eine aktive Rolle spielen.

Die Techniken (3), (4) und (5) unterscheiden sich voneinander v. a. im Bereich der Datenbank-Integration. Bei (3) greifen die Objekte mittels SQL direkt auf die Datenbank zu. Diese Technik kann für mäßig komplexe Applikationen eingesetzt werden, aber auch für Informationssysteme, die mittels CORBA in größere Systeme integriert werden sollen. Die Techniken (4) und (5) hingegen sind für reine Informationssysteme i. Allg. zu aufwändig. Die Techniken (1) und (2) sind eher geeignet für reine Informationssysteme, es handelt sich außerdem um rein passive Systeme mit prozeduralen Elementen im Server-Bereich.

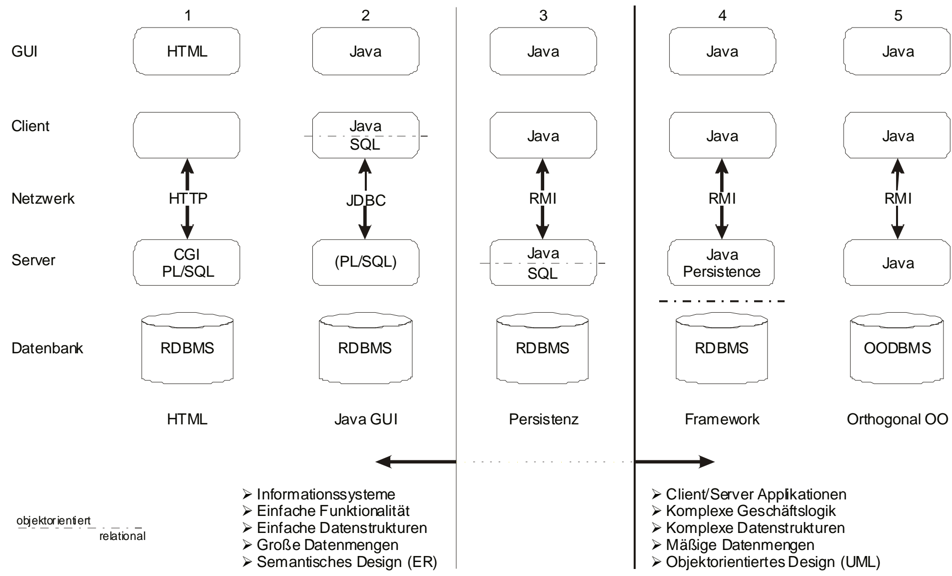

Bild 1.3: ***Persistenz mit verschiedenen Client/Server-Architekturen***

2. Lehrzielorientierte Fragen

1. Welches Grundproblem gibt es bei der Abbildung objektorientierter auf relationale Strukturen? Welche strukturellen Ansätze kennen Sie diesbezüglich?
2. Geben Sie einen Überblick über verschiedene Persistenz-Ansätze und vergleichen Sie diese qualitativ.
3. Welche Gemeinsamkeiten und Unterschiede gibt es zwischen ER und UML?
4. Erläutern Sie das Cursor-Prinzip.
5. Erläutern Sie das Grundprinzip der Call-Level-Schnittstelle (CLI) und geben Sie Beispiele an.
6. Was versteht man unter „static embedded SQL“? Erläutern Sie den Einsatz der Cursor-Technik bei static embedded SQL anhand eines Beispiels.
7. Was versteht man unter „dynamic embedded SQL“? Geben Sie Beispiele an.
8. Was versteht man unter „stored procedures“? Erläutern Sie den Einsatz der Cursor-Technik anhand von PL/SQL.