13 Grobkonzept

In diesem Kapitel werden die wesentlichen Entwurfsentscheidungen des Grobkonzeptes dargestellt. Sie umfassen die getrennte Entwicklung der Software für Schnittstelle und visuell-interaktive Bereitstellung von Raumbezügen (Kap. 13.1), die Umsetzung der Schnittstelle über eine internet-basierte dreischichtige Client-Server-Architektur (Kap. 13.2), die Verwendung der objektorientierten Programmiersprache Java für die Erstellung der Software für Client und Server der Schnittstelle (Kap. 13.3), die Realisierung eines anwenderseitig ablauffähigen ‚Fat' Client (Kap. 13.4), die Datenbank-Anbindung über Java Database Connectivity (JDBC, Kap. 13.5) sowie Vorgaben über die Bereitstellung von Datenräumen in Form einzelner Datenbanktabellen (Kap. 13.6).

13.1 Visuell-interaktive Bereitstellung von Raumbezügen

Die visuell-interaktive Bereitstellung von Raumbezügen in Form von Karten für Datenselektion und Ergebnisvisualisierung wurde als integraler Bestandteil der Schnittstelle konzipiert. Dabei wurde es als sinnvoll eingestuft, die hierfür erforderliche Software unabhängig von der Software der eigentlichen Schnittstelle zu entwerfen. Für die visuell-interaktive Bereitstellung von Raumbezügen wurde daher eine autarke Komponente realisiert, die sowohl für Datenselektion wie Ergebnisvisualisierung in die Schnittstelle eingebettet werden kann. Dieser Ansatz wurde insbesondere aus zwei Gründen gewählt. Zunächst erfordert eine geeignete Bereitstellung von Raumbezügen die Entwicklung von Lösungen, die unabhängig von den eigentlichen Aufgaben der Schnittstelle zu sehen sind und entsprechend getrennt von diesen behandelt werden sollten. Ferner erlaubt es die Realisierung einer eigenen, autarken Komponente, diese nicht nur innerhalb der Schnittstelle, sondern zusätzlich auch unabhängig von dieser zu betreiben und somit den Anwendungshorizont deutlich zu erweitern. Dies ist insbesondere von Interesse für mögliche Nachnutzungen, bspw. im Kontext weiterer georeferenzierter, hier nicht adressierter Daten oder zur komfortablen Präsentation ausgewählter wissenschaftlicher Ergebnisdaten des Institutes. Konzeption und Realisierung der Komponente für die visuell-interaktive Bereitstellung von Raumbezügen werden in Kap. 17 (IDA – Interactive Digital Atlas) ausführlich beschrieben.

13.2 Internet-basierte Client-Server-Architektur

Die geforderte Netzwerkfähigkeit der Schnittstelle wurde durch Aufsetzen auf die Infrastruktur des Internet (vgl. Kap. 5.1.1) sichergestellt, die für alle Aspekte der Kommunikation - von den Nutzerschnittstellen auf den Rechnern der Anwender bis zu den anzusprechenden RDBMS - verwendet wird. Die Nutzerschnittstellen wurden so realisiert, dass sie alternativ auch über das auf dem Internet aufsetzende World Wide Web (vgl. Kap. 5.1.2) betrieben werden können. Die Kommunikationsinfrastruktur der Schnittstelle setzt damit auf dem Transmission Control Protocol / Internet Protocol (TCP/IP) auf; eine webbasierte Bereitstellung erfolgt unter Nutzung des Hypertext Transfer Protocol (HTTP). Die Schnittstelle wurde ferner in mehreren logisch voneinander getrennten Schichten entworfen. Die Aufteilung eines Systems auf mehrere Schichten ist dann als sinnvoll anzusehen, wenn die Dienstleistungen einer Schicht auf dem selben Niveau von Abstraktion angesiedelt sind und sich die einzelnen Schichten nach ihrem Abstraktionsniveau in einer Rangfolge befinden, so dass eine Schicht jeweils nur Dienstleistungen, die von tieferen Schichten bereitgestellt werden, benötigt [Balzert 1999, 447]. Eine verbreitete Aufteilung ist eine Grobtrennung in drei miteinander über Netzwerk kommunizierende Schichten (*Drei-Schichten-Architektur*), die zwischen

- Anwendungsschicht (*Client Layer*),
- Datenzugriffsschicht (*Server Layer*) und
- Datenschicht (*Data Layer*)

unterscheidet. Diese Aufteilung wurde auch dem gewählten Systementwurf zugrunde gelegt. Während die einzubindenden Datenbanken die Datenschicht bilden, gliedert sich die Software der Schnittstelle in einen Client Layer und einen Server Layer. Der Client Layer beinhaltet dabei diejenigen Systemkomponenten, die auf dem Rechner des Anwenders ablaufen und diesem die Interaktion mit der Schnittstelle erlauben. Die Komponenten des Server Layer werden hingegen zentral betrieben und dienen primär zur Bereitstellung der erforderlichen Datenzugriffsfunktionalität für den Client Layer. Eine deutliche Trennung von Client Layer und Server Layer bietet dabei im vorliegenden Kontext eine Reihe von Vorteilen:

Kleinerer Client
- Eine Aufteilung der Gesamtfunktionalität auf Client- und Serverkomponenten erlaubt die Realisierung von Clients mit geringerem Umfang, da dort weder die Logik des Datenbankzugriffs implementiert noch die hierfür erforderlichen Treiber bereitgestellt werden müssen.

Funktionale Trennung
- Der Client kann unabhängig von den Details der Datenbankzugriffe realisiert werden und diese als Dienste über den Server abrufen. Diese klare logische Trennung unterstützt Wartbarkeit, Erweiterbarkeit und Wiederverwendbarkeit sowohl von Client- wie Serverkomponenten.

Skalierbarkeit
- Während die individuellen Clients auf den jeweiligen Rechnern der Anwender ablaufen, wird der Server zentral bereitgestellt, so dass für diesen je nach Anforderungen Möglichkeiten zur Performance-Skalierung, bspw. durch die Verwendung besonders leistungsfähiger Hardware, bestehen.

Ferner ist gerade beim Betrieb von Anwendungen über die potentiell unsichere Infrastruktur Internet *Sicherheit* ein zentrales Thema (vgl. hierzu bspw. [Knudsen 1998]). Eine Realisierung des eigentlichen Datenbankzugriffs über eine eigene Serverschicht ermöglicht hier die Etablierung von Kontrollmechanismen, die im Falle von Angriffen schwerer zu umgehen sind – so verbleibt die Software des Servers beim Betreiber und wird nicht über Netzwerk versandt, die vom Client übersandten Anforderungen können dort zudem validiert, auf Berechtigungen überprüft und gegebenenfalls zurückgewiesen werden. Schließlich gelangen sicherheitskritische Informationen, die zum Zugriff auf Datenbankmanagementsysteme erforderlich sind - hierzu zählen unter anderem Netzwerkadressen, Nutzernamen und Passwörter - nicht zum Client und sind entsprechend besser vor Missbrauch geschützt.

13.3 Realisierung mit Java

Für die Realisierung sowohl von Client- wie Server-Software wurde die Programmiersprache *Java* verwendet. Dies erlaubte neben der Nutzung der Vorteile der Objektorientierung (vgl. Kap. 13.3.1) für den Systementwurf die Realisierung von weitgehend plattformunabhängig ablauffähigen Komponenten (vgl. Kap. 13.3.2), so dass Clients und Server der Schnittstelle *ohne Portierungsaufwand* auf der überwiegenden Mehrheit der gängigen Rechner- und Betriebssystemplattformen betrieben werden können.

Java wurde 1991 von James Gosling bei der US-amerikanischen Firma Sun Microsystems[238] als Teil des Projekts *Green* entwickelt, bei dem Software zur Steuerung „intelligenter" elektronische Kleingeräte realisiert werden sollte. Hierfür wurde zunächst die objektorientierte Programmiersprache C++ verwendet; Gosling war allerdings mit einigen

[238] www.sun.com

Eigenschaften dieser Sprache nicht zufrieden und entwickelte eine daran angelehnte, neue objektorientierte Programmiersprache, die er zunächst Oak (engl. für Eiche) nannte. Als sich später herausstellte, dass dieser Name bereits vergeben war, wurde der Name Java[239] gewählt. Zur Veranschaulichung der Fähigkeiten der neuen Sprache entwickelte Sun in Java einen Web-Browser, der zudem über das World Wide Web herunterladbare Java-Programme (sog. Applets, vgl. Kap. 13.3.2 und 13.3.3) ausführen konnte. Der Browser, zunächst als WebRunner bezeichnet und dann in HotJava umgetauft, wurde 1994 vorgestellt und machte die Sprache bekannt; der Durchbruch für Java begann mit der Lizensierung durch Netscape[240] im August 1995.

13.3.1 Exkurs: Vorteile objektorientierter Programmierung

Objektorientierte Programmierung (OOP) ist ein modernes Programmierkonzept, das die Wiederverwendbarkeit und Erweiterbarkeit von Software unterstützt (vgl. hierzu ausführlich [Meyer 1997]). An dieser Stelle sollen kurz einige der zentrale Aspekte dieses Konzeptes skizziert werden. Objektorientierte Programmierung basiert auf *Klassen* und aus diesen erzeugbaren *Objekten*. Eine Klasse definiert sowohl die Art der Daten, die ein Objekt verwaltet, wie die Schnittstellen, über die extern auf die Daten zugegriffen werden kann. Jede Klasse definiert den Typ einer bestimmten Art von Objekten, so dass Klassen auch als Blaupausen für die Erzeugung von Objekten bezeichnet werden können.

Neue Klassen können dabei basierend auf Objekten anderer Klassen erstellt werden (*Aggregation*) und auf deren Funktionalität aufbauen (vgl. Abb. 13.1). Objektorientierung erlaubt so die Konstruktion komplexerer Funktionalität basierend auf jeweils einfacheren Elementen. Diese „Baukastenprinzip" unterstützt die Wiederverwendbarkeit von einmal erstellten Modulen und erleichtert die Realisierung fehlerfreier Software durch die Entwicklung jeweils kleiner, überschaubarer Komponenten.

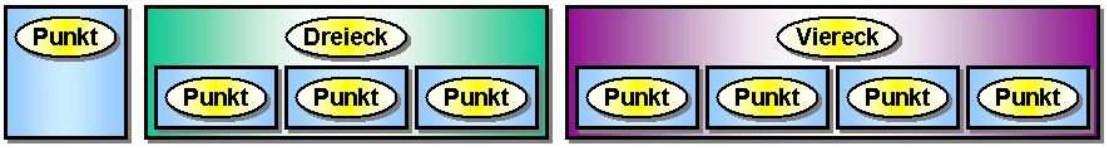

Abb. 13.1 - Ein Beispiel für Aggregation: Aufbauend auf eine Klasse, die es erlaubt, Objekte zur Verwaltung eines Punktes zu generieren, können komplexere Klassen zur Verwaltung etwa von Dreiecken oder Vierecken erstellt werden.

Von wesentlicher Bedeutung ist dabei das Konzept der *Kapselung* (*Encapsulation*) der in den Objekten verwalteten Daten, das als Umsetzung des Konzeptes der Abstrakten Datentypen[241] aufgefasst werden kann. Klassen können - und sollten - so entworfen werden, dass sowohl lesende wie schreibende Zugriffe auf die intern verwalteten Daten eines Objektes ausschließlich über dafür vorgesehene Zugriffsschnittstellen (sog. *Methoden*) erfolgen können. Relevant für einen externen Zugriff sind dabei nur die sog. Signaturen der einzelnen Methoden, die sich aus dem jeweiligen Methodennamen, einer Parameterliste sowie der Art des Rückgabewertes zusammensetzen (vgl. Tab. 13.1).

---

[239] Der Begriff Java ist ein amerikanischer Slangausdruck für Kaffee. Reminiszenzen an dieses Getränk werden von Sun beim Marketing für diese Sprache immer wieder eingesetzt (vgl. die Bezeichnung des Browsers HotJava). So ist das Logo von Java eine dampfende Kaffeetasse, ihr Maskottchen Duke eine stilisierte Kaffeebohne, die Komponententechnologie von Java heißt Java Beans etc.
[240] Die Web-Browser von Netscape unterstützten Java-Applets ab Version Netscape 2.0.
[241] Abstrakte Datentypen (ADT) sind ein zentrales Konzept zur Entkopplung von Softwarebestandteilen. Ein ADT erlaubt es, Datenstrukturen und Operationen auf diesen in einer Weise zur Verfügung zu stellen, die alle Details der internen Realisierung versteckt. Ein typisches Beispiel ist die Bereitstellung einer First-In-First-Out- (FIFO-) Speicherstruktur, auf die nur mittels der Operationen *füge-einen-neuen-Wert-hinzu* und *gib-den-nächsten-Wert-heraus* zugegriffen werden kann (zum Konzept der ADT vgl. ausführlich [Meyer 1997, 121ff.]).

Signatur	Methodenname	Parameter	Rückgabewert
`int getX()`	getX	keine Parameter	Integer-Zahlenwert
`void setX(int)`	setX	Integer-Zahlenwert	kein Rückgabewert

Tab. 13.1 - Beispiele für Methodensignaturen: Lesen und Verändern eines Zahlenwertes.

Stellt bspw. eine Klasse A zur Erzeugung von Objekten, die jeweils die Koordinaten eines Punktes verwalten, die Methode *getX* aus Tab. 13.1 bereit, um einen lesenden Zugriff auf den aktuellen Wert der X-Koordinate eines Punktes zu erlauben, so ist extern nur bekannt, dass diese Funktionalität über eine Methode mit Namen *getX* abgerufen werden kann, die keine Parameter entgegennimmt und einen Integer-Zahlenwert zurückgibt. Die Details der internen Umsetzung hingegen - etwa die konkrete Implementierung der Methode sowie die in der Klasse verwendeten Bezeichnungen und Strukturen zur Datenverwaltung - bleiben von außen unsichtbar. Auf diese Weise ist es möglich, die interne Organisation einer Klasse im Bedarfsfall im Lauf der Zeit zu verändern, ohne andere Klassen zu tangieren – solange Bedeutung und Signaturen der Zugriffsschnittstellen von den Veränderungen unberührt bleiben, sind keine Anpassungsarbeiten in den zugreifenden Klassen erforderlich.

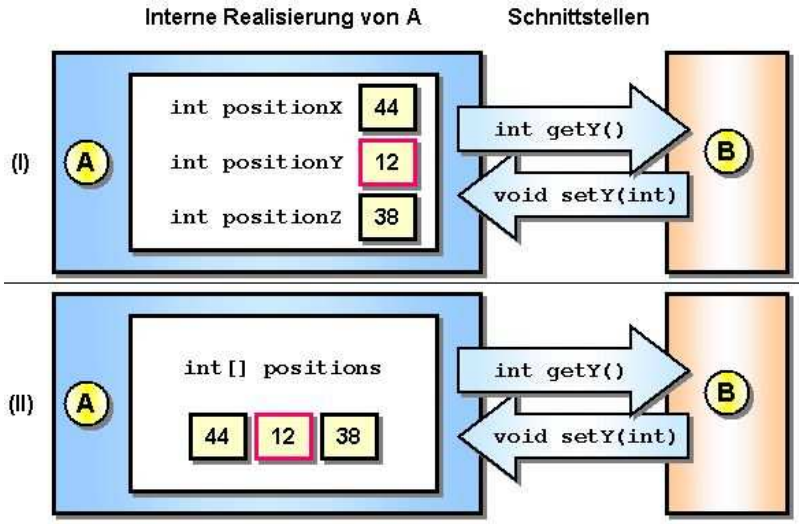

Abb. 13.2 - Entkopplung von Implementierungsdetails durch Kapselung.

Abb. 13.2 verdeutlicht dieses Prinzip an einem einfachen Beispiel. Dargestellt ist die Kommunikation zwischen Objekten zweier Klassen A und B. Klasse A dient zur Erzeugung von Objekten, die die drei Koordinaten eines Punktes im Raum verwalten; Objekte der Klasse B greifen auf diese Daten ausschließlich über die von A bereitgestellten Methoden zu. Angedeutet ist der Zugriff über zwei Methoden *getY* und *setY*, die ein externes Lesen bzw. Verändern des Wertes der Y-Koordinate eines Punktes erlauben. Abb. 13.2(II) stellt eine Veränderung der internen Datenverwaltung von Klasse A dar; die Signaturen der Zugriffsmethoden sind hingegen unverändert geblieben. Nach Abschluss der Änderungen in Klasse A sind daher in Klasse B, die ja nur über die Methoden der Klasse A Zugriff auf die verwalteten Daten erhält, keinerlei Modifikationen erforderlich. Das Konzept der Kapselung ermöglicht so eine weitreichende Entkopplung und Unabhängigkeit einzelner Klassen und Objekte voneinander und unterstützt so die Erstellung modifizierbarer, erweiterbarer und wartbarer Software.

Zu den bekanntesten Konzepten der Objektorientierung zählt das Prinzip der *Vererbung*, auch als *Spezialisierung* bezeichnet (vgl. Abb. 13.3a). Vererbung erlaubt es, neue Klassen (bezeichnet als *Kindklassen*) von einer bestehenden Klasse (*Elternklasse*) abzuleiten. Die in der Elternklasse definierte Funktionalität wird so von den Kindklassen übernommen (*geerbt*) und kann dort sowohl erweitert wie angepasst werden. Bedeutsam ist hierbei die sog.

echte Polymorphie durch *Überschreiben* geerbter Funktionalität. So kann etwa eine Elternklasse (geometrische) *Figur* eine Methode „verschiebe die Figur um den Wert X" bereitstellen, die von entsprechenden Kindklassen wie *Dreieck*, *Viereck* und *Polygon* jeweils geeignet überschrieben werden kann. Einer der Vorteile dieses Konzeptes ist es, dass auf diese Weise auf einer höheren Ebene *Objekte der Kindklassen* als *Objekte der Elternklasse* verwaltet werden können. Dies erlaubt es im Beispiel der geometrischen Figuren, Dreiecke, Vierecke und Polygone einheitlich als Objekte vom Typ *Figur* anzusehen, ohne dabei auf spezifische Details wie etwa die Anzahl der jeweils zu verwaltenden Punkte eingehen zu müssen (vgl. Abb. 13.3b).

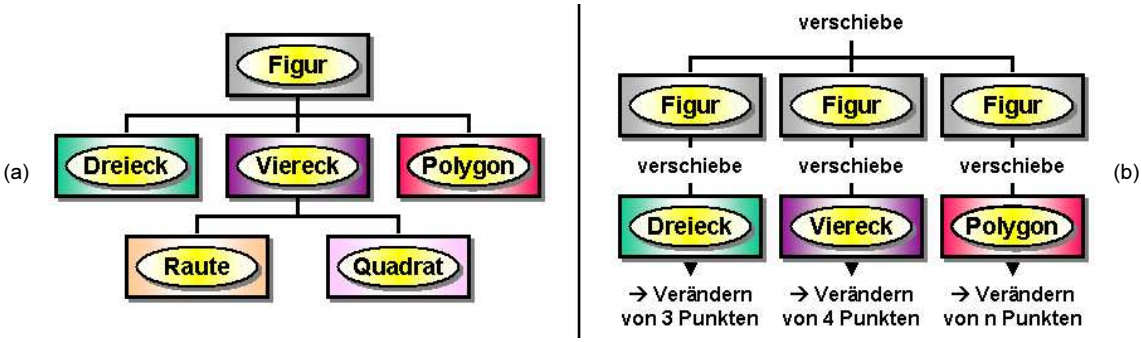

Abb. 13.3 - Vererbung: (a) Vererbungshierarchie; (b) einheitliche Interaktion mit Objekten spezialisierter Kindklassen über Objekte der Elternklasse.

13.3.2 Exkurs: Plattformunabhängige Bereitstellung

Neben der Unterstützung objektorientierter Konzepte erlaubt es Java, weitgehend plattformunabhängige Software zu schreiben, also Programme, die ohne Anpassungsaufwand auf unterschiedlichen Hardware- und Betriebssystemplattformen ablauffähig sind. Bei vielen Programmiersprachen wird aus dem Programmtext (*Quellcode*, engl. *Sourcecode*) ein plattformspezifisch ausführbares Programm (*Executable*) erzeugt, so dass für eine Anpassung an andere Plattformen (die sog. Portierung) zumindest eine Neuübersetzung, oft jedoch auch zusätzlich entsprechende Anpassungen am Quellcode erforderlich sind.

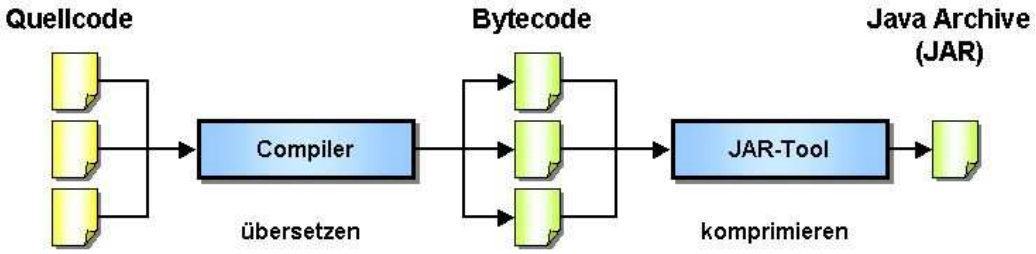

Abb. 13.4 - Erzeugung und Komprimierung von Java-Bytecode.

Um eine Plattformunabhängigkeit zu erreichen, werden aus Java-Programmen *keine* Executables erstellt. Anstelle dessen wird aus dem Quellcode durch einen Compiler ein maschinenunabhängiger Zwischencode - der sog. *Bytecode* - erzeugt, der für eine nachfolgende Verarbeitung optimiert wird. Die entstehenden Bytecode-Dateien können fakultativ ferner noch zu einem Java-Archiv (JAR) zusammengefasst werden, so dass der Bytecode in Form einer komprimierten Datei bereitgestellt werden kann (vgl. Abb. 13.4).

Die Abbildung des Bytecodes auf die einzelnen Plattformen übernimmt jeweils ein spezieller Interpreter, der als *Virtuelle Maschine* (VM) bezeichnet wird. Virtuelle Maschinen setzen den Bytecode in Maschinenbefehle einer konkreten Plattform um und simulieren dabei eine überall identische Plattform. Die Erstellung des Quellcodes sowie seine Übersetzung kann dabei auf einer beliebigen Plattform erfolgen; der so entstehende Bytecode ist automatisch und ohne Neuübersetzung auf alle anderen Plattformen portabel, für die eine entspre-

chende Virtuelle Maschine zur Verfügung steht. Zu den unterstützten Plattformen zählen gegenwärtig Microsoft Windows (3.x, 95, 98, NT, 2000, XP), Sun Sparc / Solaris, AIX (IBM-Unix) und Macintosh ebenso wie Linux. Für die Softwareentwicklung entsteht dabei der Vorteil, dass plattformspezifische Details (etwa unterschiedliche interne Repräsentationen von Zahlenwerten) nicht berücksichtigt werden müssen.

Ein Java-Programm kann dabei sowohl als autonome Applikation (*Java Application*) wie als *Java Applet* bereitgestellt werden (vgl. Abb. 13.5). Eine in Java realisierte Applikation unterscheidet sich - bis auf ihre plattformunabhängige Umsetzung - im Prinzip nicht von in anderen Programmiersprachen entwickelter Software. Java Applets können hingegen gemeinsam mit einer HTML-Seite von einem Web-Server über das World Wide Web (vgl. Kap. 5.1.2) bereitgestellt werden.

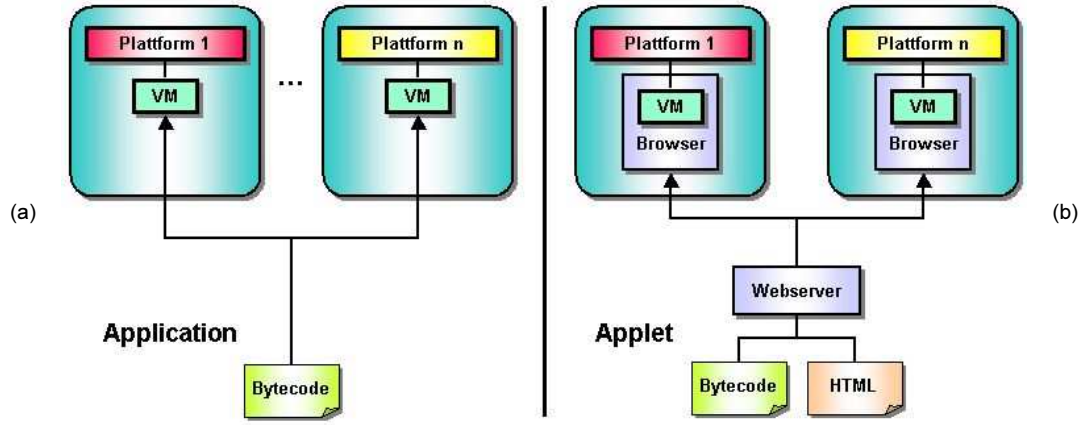

Abb. 13.5 - Plattformunabhängige Umsetzung von Java-Programmen: (a) Applikation; (b) Applet.

Hierfür genügt es im Prinzip, zusätzlich zu einem Applet eine HTML-Seite bereitzustellen, die auf dieses verweist. Gängige Web-Browser besitzen eine eigene Virtuelle Maschine, die die Ausführung der Applets auf dem Client-Rechner ermöglicht. Wird über einen solchen Browser eine HTML-Seite aufgerufen, an die ein Applet gebunden ist, wird dessen Bytecode auf den Rechner des Anwenders heruntergeladen und von der Virtuellen Maschine des Browsers interpretiert.

13.3.3 Exkurs: Applets vs. Applikationen

Ein Java-Client kann damit dem Anwender entweder als Applet oder als Applikation zur Verfügung gestellt werden. Eine Bereitstellung als Applet bietet dabei insbesondere den Vorteil, dass der Bytecode des Client bei jedem Start neu auf den Anwenderrechner geladen werden kann, so dass eine zentrale Wartung der Software und eine unaufwendige Aktualisierung aller Clients im Falle erforderlicher Änderungen möglich ist[242]. Nachteile dieser Variante liegen hingegen in den folgenden Punkten:

- **Einschränkungen der Funktionalität**

Um zu verhindern, dass Applets, die als ausführbare Software aus dem potentiell unsicheren Internet geladen werden, als Viren missbraucht werden können, besitzen Applets aus Sicherheitsgründen nur eingeschränkte Zugriffsrechte. So dürfen Applets zunächst weder Daten auf den Rechner des Anwenders schreiben noch Daten von diesem lesen; ferner

[242] Mitte der 90er Jahre wurde in diesem Zusammenhang u.a. von Sun und Oracle sog. Network Computer (NC) propagiert, die als preiswerte schlanke Internet-Terminals mit Java-basiertem Frontend Java-Programme direkt aus dem Internet laden und so den von Microsoft dominierten PCs Konkurrenz machen sollten. Oracle-Chef Larry Ellison prognostizierte zu dieser Zeit, dass bis zum Jahr 2000 die Zahl der weltweit verkauften NCs mit 100 Millionen die der PCs bereits übertreffen könnte (vgl. [Nollert 1996]). Diese Vision hat sich bis heute nicht erfüllt.

dürfen Netzwerkkontakte nur zu dem Rechner, von dem ein Applet heruntergeladen wurde, aufgenommen werden[243]. In Version 1.0 der Programmiersprache waren diese Bedingungen unumgänglich; ab Version 1.1 können individuelle Erweiterungen der Zugriffsrechte durch den Einsatz sog. signierter Applets[244] erreicht werden, was aber erhöhten Aufwand des Bereitstellers erfordert.

- **Längere Startzeit**

Im Vergleich zu Applikationen ist für den Start eines entsprechenden Java Applets in der Regel ein erhöhter Zeitaufwand erforderlich. Dies ist zum einen darin begründet, dass der entsprechende Bytecode zunächst zum Client geladen wird; zudem muss er dort vor einem Start verifiziert werden, um die Einhaltung der Java-Sicherheitsvorgaben für Applets zu gewährleisten.

- **Spätere Verfügbarkeit aktueller Virtueller Maschinen**

Es gibt in der Regel deutliche Verzögerungen, bis neue Versionen Virtueller Maschinen, die bereits für Applikationen verwendet werden können, auch für Web-Browser zur Verfügung stehen. Durch diese Diskrepanz entstehen für den Betrieb von Applets mehrere bedeutsame Nachteile. Zunächst erlauben neuere Virtuelle Maschinen in der Regel aufgrund interner Optimierungen die Ausführung von Java-Programmen mit einer höheren Geschwindigkeit. So wurden seit VM 1.0 deutliche Performance-Steigerungen durch die Integration von *Just in Time* (JIT)- sowie *Hotspot*-Compilern[245] in neueren Virtuellen Maschinen erreicht, die zudem bei späteren Versionen durch Verbesserungen der Compiler weiter erhöht wurden[246]. Während neue, performantere Virtuelle Maschinen für Applikationen bereits eingesetzt werden können, stehen sie für den Betrieb von Applets zumeist zunächst noch nicht zur Verfügung. Noch schwerwiegender ist der Umstand einzustufen, dass mit neuen Virtuellen Maschinen zumeist jeweils auch neue, verbesserte oder erweiterte Klassenbibliotheken zur Verfügung stehen. Anwendungen, die Klassen aus Bibliotheken einer höheren Version verwenden, sind in älteren Virtuellen Maschinen in der Regel nicht ablauffähig. Sofern Kompatibilität zwischen älteren Java-Versionen und neuen Klassenbibliotheken besteht, können diese zwar prinzipiell mit einem Applet zum Anwenderrechner geladen werden, dies ist aber wegen der Größe der zu übertragenden Klassen oft wenig praktikabel. Erschwerend kommt die je nach Anwendergemeinde sehr heterogene Landschaft aus verwendeten Browsern und Virtuellen Maschinen hinzu: Selbst wenn die neuesten Versionen von Browsern eine Virtuelle Maschine in der erforderlichen Version enthalten, kann nicht in jedem Fall davon ausgegangen werden, dass alle Anwender über diesen Browser verfügen oder bereit sind, diesen zu installieren.

Bei Java Applikationen entfallen diese Nachteile. Hier ist allerdings zu berücksichtigen, dass im Interesse eines geringen Wartungsaufwandes sichergestellt werden muss, dass die Software der jeweiligen Clients im Falle notwendiger Änderungen unaufwendig aktualisiert werden kann. Da es möglich ist, eine Java-Anwendung gleichermaßen als Applikation

---

[243] Auf diese Weise soll verhindert werden, dass Applets, die von außen in ein Intranet und damit hinter einen Firewall geladen wurden, unbemerkt weitere dort befindliche Rechner kontaktieren, schützenswerte Informationen ausspähen und zurück zum Heimatrechner transferieren können.

[244] Dabei wird durch eine externe Zertifizierung sichergestellt, dass der Programmcode eines Applets nicht während des Netzwerktransports vom Bereitsteller zum Anwender verändert wurde.

[245] Just in Time-Compiler wandeln den Bytecode einer Java-Anwendung während der Laufzeit in perfomanteren, plattformspezifischen Code um. Hotspot-Compiler optimieren diesen Ansatz zusätzlich durch eine Vorab-Analyse zur Identifikation und Unterstützung der besonders performance-relevanten Teile einer Java-Anwendung.

[246] So erreichte etwa die im Dezember 1998 vorgestellte Java-Version 2 mit Just in Time-Compiler im Vergleich zu Version 1.1 (ebenfalls mit Just in Time-Compiler) eine bis zum Faktor 4,2 höhere Performance [Müller 1999].

und als Applet zu konzipieren, können beide Ansätze miteinander kombiniert und nach Bedarf eingesetzt werden. Dieser Weg wurde auch für den Client der zu realisierenden Schnittstelle gewählt (vgl. Kap. 13.4). Die überwiegende Mehrheit der Klassen kann dabei in identischer Weise unabhängig vom Betrieb als Applikation oder als Applet eingesetzt werden. Unterschiedliche Vorgehensweisen sind hingegen bei Funktionalitäten erforderlich, die je nach Betriebsart eine jeweils andere Implementierung erfordern (so erfolgt etwa der Zugriff auf vom Heimatrechner bereitgestellte Dateien bei Applets über URLs). Um die Unabhängigkeit der übrigen Programmlogik von diesen Spezifika sicherzustellen, wurde eine Klasse entworfen, die derartige Funktionalitäten unabhängig von der jeweiligen Betriebsart bereitstellt und eine transparente Umsetzung vornimmt (vgl. Abb. 13.6).

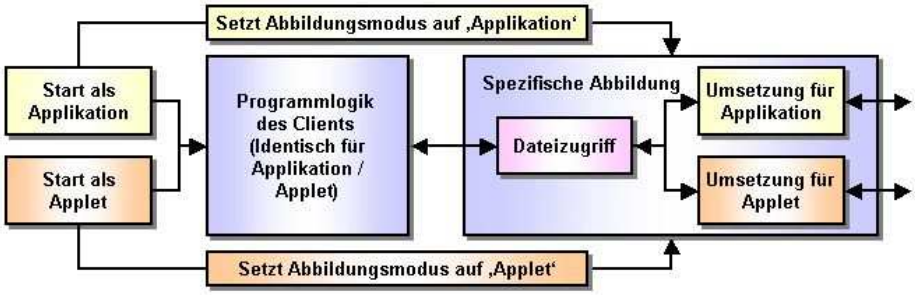

Abb. 13.6 - Entkopplung der Programmlogik von applet- bzw. applikationsspezifischen Details.

13.4 Anwenderseitig ablauffähiger ‚Fat' Client

Der *Client* der Schnittstelle wurde als auf dem jeweiligen Rechner des Anwenders ablauffähiges Programm (sog. ‚*Fat' Client*) realisiert, das sowohl als autonome Anwendung (Java Application) wie über einen gängigen Web-Browser (Java Applet) betrieben werden kann. Die Entscheidung für einen ‚Fat' Client wurde getroffen, da auf diese Weise ein hoher clientseitiger Interaktionsumfang realisiert werden kann, der es erlaubt, zentrale Funktionalitäten autonom auf dem Rechner des Anwenders auszuführen, ohne hierfür jeweils mit dem Server Kontakt aufnehmen zu müssen. Hieraus resultieren im gegebenen Anwendungskontext zwei wesentliche Vorteile:

- **Clientseitige Interaktionsverarbeitung**

Interaktionen mit der Schnittstelle können auf diese Weise in hohem Maße direkt auf dem Rechner des Anwenders verarbeitet und entsprechend schnell in Feedback umgesetzt werden. Auf diese Weise kann eine unmittelbar reagierende Software erstellt werden, die dem Anwender das Gefühl vermittelt, sie zu jeder Zeit unter Kontrolle zu haben.

- **Clientseitige Informationsgenerierung**

Ein anwenderseitig ablauffähiges Programm bildet zudem die Basis für eine schnelle, flexible und interaktive Umsetzung von Ergebnisdaten auf dem Rechner des Anwenders. Auf diese Weise können bspw. interaktive Visualisierungen mit hohem Feedbackumfang realisiert sowie ohne Serverkontakte unterschiedliche visuelle und textuelle Darstellungen aus vorliegenden Ergebnisdaten generiert werden.

13.5 Datenbank-Anbindung über JDBC

Der Server der Schnittstelle wurde als Java-Anwendung (Java Application) realisiert. Für den Zugriff auf die einzelnen Datenräume stand damit *Java Database Connectivity* (JDBC)[247] zur Verfügung. JDBC stellt die Java-Schnittstelle zu relationalen Datenbankmanagementsystemen (RDBMS) dar. Dabei wird ein gleichartiger Zugriff auf

[247] http://java.sun.com/products/jdbc/

eine Vielzahl von RDBMS unterschiedlicher Anbieter unterstützt, für die jeweils entsprechende Treiber verwendet werden. Auf diese Weise können Anfragen, die auf Standard-SQL basieren, die also keine proprietären, produktspezifischen Erweiterungen beinhalten, auf unterschiedliche RDBMS abgebildet werden. Da zudem von JDBC ein netzwerkfähiger Datenbankzugriff unterstützt wird, konnte auf diese Weise die geforderte Möglichkeit zur Anbindung verschiedenartiger und verteilter RDBMS realisiert werden. Jeder einzubeziehende Datenraum ist damit über ein RDBMS bereitzustellen, für das eine JDBC-Schnittstelle existiert. Da diese Voraussetzung für die überwiegende Mehrheit gängiger RDBMS gegeben ist[248], stellt sie keine nennenswerte Einschränkung dar; eine gegebenenfalls erforderliche Ausweitung auf nicht über JDBC direkt zugängliche RDBMS kann überdies durch Verwendung einer sog. JDBC-ODBC-Bridge, der Kopplung von JDBC und der verbreiteten Datenbank-Zugriffsschnittstelle Open Database Connectivity (ODBC), erreicht werden.

13.6 Abbildung von Datenräumen auf Tabellen

Jeder einzubeziehende Datenraum ist in Form *jeweils einer Datenbanktabelle* bereitzustellen[249]. Da relationale Datenbanken primär zur redundanzfreien Datenspeicherung entworfen werden und daher in der Regel jeweils aus *mehreren* Tabellen bestehen, sind hier also Transformationen zur Abbildung eines Datenraumes vor seiner Einbindung in die Schnittstelle erforderlich[250]. Ausschlaggebend für diese Entwurfsentscheidung waren folgende Faktoren:

- **Vereinfachung**

Vor dem Hintergrund der Komplexität der gegebenen Datenräume stellt die Betrachtung jedes Datenraumes als einzelne Datenbanktabelle eine wertvolle Vereinfachung für die Konzeption der Schnittstelle dar. Auf diese Weise ist es möglich, von den vielfältigen Ausprägungen möglicher Tabellenstrukturen in den einzelnen Datenräumen zu abstrahieren.

- **Etablierung einer zusätzlichen Abstraktionsschicht**

Durch die Abbildung der Ausgangsdaten wird die geforderte zusätzliche Datenschicht zwischen diesen und der Schnittstelle etabliert. Durch diese Abstraktionsschicht findet eine sinnvolle Entkopplung zwischen Schnittstelle und Ausgangsdaten statt, die es den einzelnen Datenbereitstellern erlaubt, jeweils bestehende Datenstrukturen aufrechtzuerhalten und es zugleich ermöglicht, erforderliche Homogenisierungen bei der Abbildung der Ausgangsdaten durchzuführen.

---

[248] Vgl. die unter http://servlet.java.sun.com/products/jdbc/drivers bereitgestellte Datenbank der verfügbaren Treiber.

[249] Diese Bedingung wurde vor Beginn von Betriebsphase II gelockert (vgl. Kap. 16.2.1, Unterstützung von Tabellenkaskaden). Ausgenommen ist ferner die Datenbank zur integrierten Bereitstellung von Zeitreihen (vgl, Kap. 18.1.2), auf die die Schnittstelle über eine hierfür von der Scientific Data Management Group entwickelte Extraktionssoftware zugreift (vgl. Kap. 22.2).

[250] Der Aufwand hierfür kann durch Verwendung sog. *Views* auf RDBMS verlagert werden. Views sind virtuelle Datenbanktabellen, die von RDBMS aus vorhandenen Datenbanktabellen dynamisch generiert werden; sie bilden eine gängige Technik zur Bereitstellung unterschiedlicher Nutzersichten auf komplexe Tabellenstrukturen. Dieses Konzept wurde in Betriebsphase I bspw. zur Bereitstellung unterschiedlicher Ausschnitte der CERA-2-Metadatenbank genutzt, während in Betriebsphase II generell zur Bereitstellung „materialisierter" Datenbanktabellen übergegangen wurde.