Realisierung von verteilten Editoren in Java auf Basis eines aktiven Repositories

Udo Kelter, Marc Monecke, Dirk Platz

Praktische Informatik, Fachbereich Elektrotechnik und Informatik
Universität Siegen, 57068 Siegen
e-mail: {kelter|monecke|platz}@informatik.uni-siegen.de

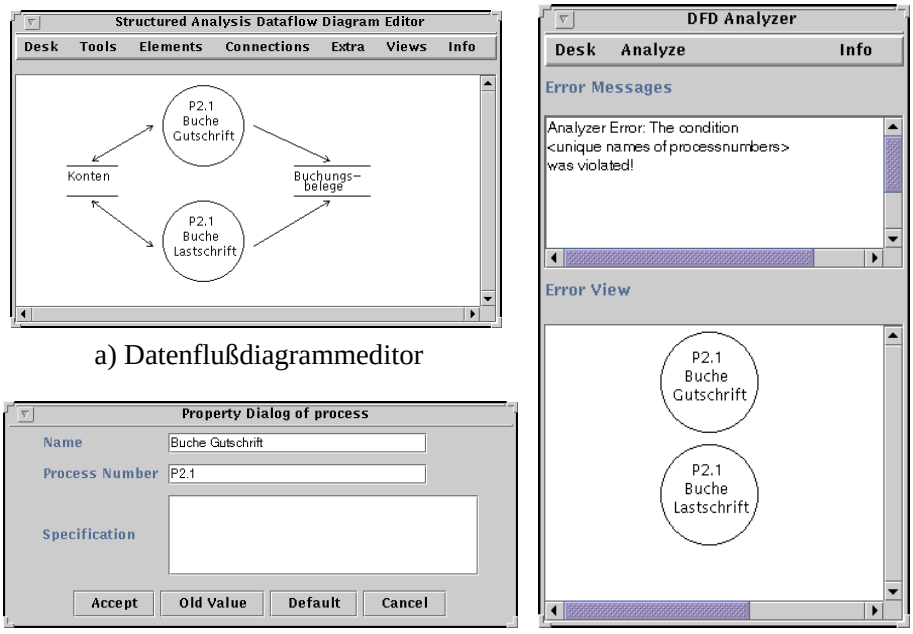
Zusammenfassung Dieses Papier behandelt die Konstruktion verteilter Editier-Umgebungen am Beispiel von Software-Entwicklungsumgebungen. Die vorgeschlagene Architektur basiert auf einem aktiven Repository, das Änderungen an den Daten netzwerkweit und sichtenübergreifend propagiert und diverse andere Dienste (Transaktionen, Zugriffskontrollen) anbietet. Die textuellen und/oder graphischen Benutzungsschnittstellen sind in Java geschrieben und daher (z.B. als Applets) leicht verteilbar. Sie kommunizieren über einen Java-API-Server indirekt mit dem Repository. Wir beschreiben diverse Details dieser Architektur, insbesondere das Prozeßkonzept, das die parallele Ausführung von Java-Anwendungen ermöglicht, die netzwerkweite Notifizierung sowie die relevanten Java-Klassen für den Zugriff auf das Repository.

1 Einführung und Motivation

Java ist nicht alleine eine objektorientierte Programmiersprache, sondern darüber hinaus eine Technologie zur Realisierung verteilter Systeme. In diesem Papier betrachten wir eine spezielle Klasse verteilter Systeme, nämlich Software-Entwicklungsumgebungen (SEU). Eine SEU umfaßt typischerweise eine Vielzahl von graphischen oder textuellen Editoren, mit denen Dokumente angezeigt und ediert werden können. Abb. 1 zeigt drei verschiedene Werkzeuge für Datenflußdiagramme (*DFDs*). Weitere gängige Dokumenttypen sind OOA-, ER-Diagramme, Petri-Netze, u.ä. Integriert in die interaktiven Editoren oder als separate Werkzeuge sind ferner diverse nichtinteraktive Werkzeuge zur Analyse, Prüfung, Übersetzung und sonstigen Verarbeitung der Dokumente vorhanden.

Verteilt ist eine SEU[1] in dem Sinne, daß sie von mehreren Entwicklern parallel benutzt wird – umfangreiche Software wird stets in Teams entwickelt – und daß diese Entwickler an verschiedenen, möglicherweise heterogenen Rechnern arbeiten, die durch ein lokales oder (in zunehmendem Ausmaß) weitverteiltes Netz miteinander verbunden sind. Trotz dieser Verteilung müssen die Entwickler eines Teams auf einem einzigen, logisch zentralen Datenbestand arbeiten.

[1] Genaugenommen müßte von einer Installation der SEU auf bestimmten Rechnern gesprochen werden; da aus dem Kontext klar ist, was gemeint ist, wird auf diese begriffliche Trennung verzichtet.



a) Datenflußdiagrammeditor

b) Spezifikationseditor für Prozesse

c) Analysator

Abbildung 1. Werkzeuge zur Bearbeitung von Datenflußdiagrammen

Ein an einem Rechner sitzender Entwickler arbeitet i.a. in mehreren Fenstern[2] parallel mit der SEU. Hierbei kann es sich einerseits um mehrere interaktive Editoren handeln, die nur quasiparallel ausgeführt werden, da zu einem Zeitpunkt immer nur ein Editor benutzt werden kann. Zusätzlich können nichtinteraktive Werkzeuge, die nur Kontrollausgaben erzeugen, echt parallel ausgeführt werden, wie z.B. ein Analysator für DFDs oder ein Übersetzer / Systemgenerierer (`make`).

Die verschiedenen Fenster bieten bestimmte Sichten auf die Daten an, die oftmals inhaltlich überlappen: wird z.B. die Prozeßnummer vom Prozeß *Buche Gutschrift* im Spezifikationseditor in Abb. 1b in P2.2 geändert, sollte die Anzeige auch im DFD-Editor in Abb. 1a umgehend und automatisch aktualisiert werden. Des weiteren sollte auch der Analysator in Abb. 1c auf die Änderung reagieren, da die Konsistenzverletzung durch die Änderung behoben worden ist.

Allgemein sollten Änderungen immer auf *alle* betroffenen Fenster, auch auf solche, die auf anderen Rechnern laufen, propagiert werden, damit alle Fensterin-

[2] Um anschaulich zu bleiben, wird i.f. der Begriff Fenster als Synonym für Werkzeug benutzt. Manche Werkzeuge verwenden zusätzliche Fenster für Parametereingaben u.a.; solche Fenster sind hier nicht gemeint.

halte immer dem aktuellen Stand entsprechen. In vielen Fällen wird es höchstens einem Fenster erlaubt sein, die angezeigten Daten zu verändern. Es kommen auch Dokumente vor (z.B. eine zentrale Dokumentenliste), die von mehreren Rechnern unabhängig modifiziert werden können, ferner sind fallspezifische Zwischenformen denkbar. Systemseitig ist zur Lösung dieser Probleme ein geschachteltes Transaktionskonzept erforderlich.

Diese Probleme bzw. Anforderungen treten auch in anderen Anwendungen auf, insbesondere bei den verschiedensten Buchungs- oder Datenerfassungs- und -auswertungssystemen. Insgesamt bilden derartige Anwendungen eine Problemklasse, die durch folgende Merkmale charakterisiert wird:

- verteilter Zugriff auf Dokumente in heterogenen Umgebungen
- Anzeige- und Editierfenster, die unterschiedliche Sichten auf gleiche oder überlappende Daten anbieten
- automatische netzwerkweite, sichtenübergreifende Änderungspropagation
- Änderungen können von interaktiven und nichtinteraktiven Werkzeugen verursacht werden
- echt parallele und quasiparallele Werkzeugausführungen

Diese Anforderungen werden von Notifizierungsmechanismen innerhalb von GUI-Frameworks (z.B. dem AWT) nicht erfüllt, weil diese nur lokal arbeiten.

I.f. wird eine Lösung der vorstehend umrissenen Problemklasse vorgestellt, die auf folgenden grundlegenden Entwurfsentscheidungen basiert:

1. Die Dokumente werden in einer logisch zentralen Datenbank[3] gespeichert. I.f. wird ein objektorientiertes Datenbankmanagementsystem (ooDBMS) unterstellt, da es sich zur Dokumentverwaltung besser eignet als relationale DBMS (s. [4]). ooDBMS bieten hierzu u.a. spezielle Datenmodelle, Transaktions-, Versions- und Verteilungskonzepte an [7, 11, 16]. ooDBMS als Basis für SEU werden auch *Repositories* oder Objektmanagementsysteme (*OMS*) genannt. Statt von einer Datenbank reden wir i.f. von einer *Objektbank*.
   Als konkretes OMS verwenden wir das System H-PCTE [9, 8], das wesentliche Teile des ISO-Standards 13719 PCTE [15, 17] realisiert. Die Ergebnisse dieses Papiers sind aber auch auf andere ooDBMS übertragbar.
2. Eine laufende SEU besteht aus mehreren echt- oder quasiparallelen Werkzeugausführungen, wobei jede Werkzeugausführung aus Sicht der Objektbank einen eigenen Anwendungsprozeß mit eigenem externen Schema und eigenem Recovery bildet.
3. Die Werkzeuge arbeiten "direkt" auf der Objektbank, d.h. alle Editierschritte werden sofort in die Objektbank propagiert. Wir nennen eine solche Werkzeugarchitektur *OMS-orientiert* [3, 6, 14] und werden sie in Abschnitt 2.2 näher erläutern. In OMS-orientierten Architekturen übernimmt das OMS die Aufgabe, laufende Werkzeuge, die auf Änderungen an Daten reagieren müssen (insb. durch Änderung von Fensterinhalten), zu benachrichtigen. Um

[3] Die Datenbank kann physisch verteilt sein, was aber i.f. nicht relevant ist.

die Benutzung eines derartigen Notifikationsmechanismus einfach zu machen, sollte das OMS "*callbacks*" durchführen (vgl. [14]), d.h. technisch gesehen wird aus der API[4]-Bibliothek des OMS heraus eine Funktion aufgerufen, die im Anwendungsprogramm definiert ist.

4. Die SEU ist ein verteiltes System: die graphischen Benutzungsschnittstellen der Werkzeuge werden in Java realisiert und laufen lokal an den Arbeitsplätzen. Sie kommunizieren über Internet-Protokolle mit einem speziellen Java-Server, der ein Java-API zur Objektbank anbietet. Datenintensive und komplexe Anwendungen (z.B. Übersetzungen) werden serverseitig ausgeführt.

Dieser Lösungsansatz wirkt in seiner Grobstruktur naheliegend, im Detail ergeben sich allerdings knifflige Probleme:

Aus Aufwandsgründen kann ein OMS nicht komplett neu in Java implementiert werden; stattdessen muß ein vorhandenes OMS mit einem Java-API versehen werden. Architektonische Alternativen hierfür werden in Abschnitt 3 diskutiert.

Aus technischen Gründen muß der Java-API-Server mehrere Anwendungen (Applets) gleichzeitig bedienen können. Aus Sicht des OMS ist der Java-API-Server nur eine einzige Anwendung. Die von ihm bedienten Applets spielen hier aber die Rolle der parallel laufenden Anwendungen auf der Objektbank, d.h. sie benötigen u.a. eigene Recovery-Logs und externe Schemata. Eine Lösung für dieses Problem wird in Abschnitt 4 angegeben.

Analog tritt dieses Problem bei der Zuordnung von callbacks zu Applets auf. Zusätzlich ist hier die Frage zu beantworten, wie die bei einem callback aufzurufende Java-Operation in einem Applet identifiziert wird. Ein weiteres Problem bei der verteilten Notifikation ist, daß der Java-API-Server an die laufenden Anwendungen asynchron Nachrichten über Änderungen an überwachten Objekten senden muß. Dies widerspricht der üblichen asymmetrischen Kommunikation, wonach nur Anwendungen Dienstanforderungen an einen Server schicken (und dann auf eine Antwort warten) und nicht empfangsbereit für asynchron eintreffende Nachrichten sind. Diese Probleme werden in Abschnitt 5 behandelt.

2 Hintergrund

2.1 H-PCTE

Da die meisten Details von H-PCTE hier nicht relevant sind, beschreiben wir nur die wichtigsten Merkmale; diese sind in dieser oder ähnlicher Form auch in anderen OMS zu finden.

Das Datenmodell von H-PCTE basiert auf dem Entity-Relationship-Modell. Eine Objektbank enthält *Objekte* und *Beziehungen*, hier als *Links* bezeichnet, die die Objekte verbinden. Objekte und Links sind typisiert und können Attribute haben. Die Objekttypen bilden eine Typhierarchie (mit mehrfachem Erben).

[4] API = *application programming interface*

H-PCTE verfügt über einen einfachen, aber effektiven Sichtenmechanismus, bei dem ein *externes Schema* i.w. als Teilmenge aller Typen spezifiziert ist. Jedes Werkzeug benutzt ein externes Schema und "sieht" in der Objektbank nur Instanzen von Typen (incl. Subtypen), die in seinem externen Schema enthalten sind.

2.2 OMS-orientierte Werkzeugarchitektur

Ein zentrales Ziel von OMS ist, die Realisierung von Werkzeugen zu vereinfachen und vor allem den Programmieraufwand zu reduzieren, indem möglichst viele Teilprobleme, die bei der Werkzeugentwicklung auftreten, durch Dienstleistungen des OMS gelöst werden. Es zeigt sich allerdings, daß die Leistungen eines OMS mit konventionellen, dateiorientierten Werkzeugarchitekturen nicht effektiv ausgenutzt werden können, sondern daß hierzu eine *"OMS-orientierte"* Werkzeugarchitektur erforderlich ist.

Konventionelle Werkzeuge speichern Dokumente in Dateien, z.B. je ein DFD in einer Datei. Wir sprechen hier von einer *grobgranularen Datenmodellierung*. Der Dateiinhalt hat eine bestimmte Syntax, durch die die Dokumentfeinstruktur rekonstruiert werden kann. Die in einem Werkzeug angebotenen Funktionen (z.B. das Erzeugen eines Prozesses in einem DFD) können nicht direkt auf dem Dateiinhalt durchgeführt werden; stattdessen muß dieser zunächst in eine transiente Kopie in den Hauptspeicher des Werkzeugs konvertiert werden. Diese transiente Kopie wird erst beim "Sichern" des Dokuments wieder zurückkonvertiert.

Entscheidend an dieser Stelle ist die Beobachtung, daß hier nur *transiente Dokumentkopien* bearbeitet werden und daß sich der aktuelle Zustand der Dokumente infolge von Editieroperationen nur noch in der transienten Kopie darstellt, nicht hingegen im unterliegenden persistenten Speichersystem. Ersetzt man nun in dieser Architektur das Dateisystem durch ein OMS, werden alle Dienstleistungen des OMS, wie z.B. Abfragesprachen, *praktisch wertlos*, da sie auf den veralteten Daten in der Objektbank arbeiten würden. Relevante Leistungen des OMS müssen daher erneut, also redundant, in den Werkzeugen realisiert werden. Diese Redundanz kann man vermeiden, wenn (a) die Werkzeuge alle benutzerveranlaßten Änderungen *inkrementell* und *sofort* in der Objektbank eintragen und (b) die Entwicklungsdaten *feingranular* modelliert werden, d.h. die Dokumentfeinstruktur muß vollständig in der Objektbank nachgebildet werden; eine grobgranulare Modellierung wie in Dateien würde zu hohen Konvertierungsaufwand und diversen weiteren Problemen führen [2, 10].

Unter diesen Annahmen liegt es nahe, die transiente Datenhaltung in den Werkzeugen, die grob geschätzt ein Drittel des Werkzeugquelltextes ausmacht [1], schlichtweg zu vermeiden und *direkt auf der Objektbank zu arbeiten*.

Hierbei ergibt sich die in Abb. 2 gezeigte Struktur: Die graphischen Objekte der Anzeige werden im Werkzeug durch Java-Laufzeitobjekte repräsentiert. Die Laufzeitobjekte enthalten *keine Kopien der Dokumentdaten*, stattdessen enthalten sie Referenzen auf die entsprechenden Objekte und Links in der Objektbank.

Nur eine derartige Architektur erlaubt es, Dienste des OMS auszunutzen, z.B. den Sichtenmechanismus des OMS zur Integration heterogener Werkzeuge

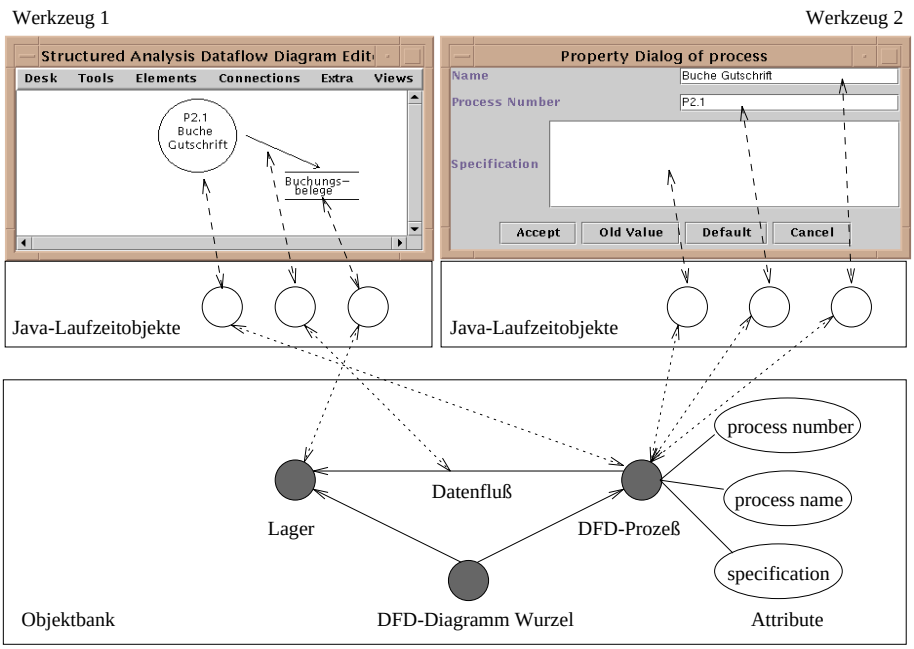


Abbildung 2. Zusammenhang zwischen Graphik, Java-Laufzeitobjekten und Objektbankobjekten

(*multiple view integration* [13, 18]). Die Werkzeuge aus Abb. 2 zeigen beispielsweise beide das gleiche (Objektbank-) Objekt, das den Prozeß P2.1 repräsentiert, an und benutzen hierzu verschiedene externe Schemata.

3 Ansätze zur Realisierung eines Java-API für H-PCTE

Die Dienstleistungen eines OMS sollten durch Werkzeuge, die in verschiedenen Sprachen geschrieben sind, ausnutzbar sein. Die PCTE-Standards definieren daher verschiedene APIs, u.a. in der Sprache C und Ada. H-PCTE bietet nur ein C-API an. Die Frage stellt sich somit, wie Java-Programme Funktionen von H-PCTE technisch aufrufen können[5].

1. Verwendung des *Native Method Interface* (NMI): Das Java-API wird durch *native methods* realisiert, also nicht in Java, sondern durch C-Funktionen, die auf das C-API von H-PCTE zugreifen. Dieser Ansatz scheitert, weil Java und H-PCTE threads einsetzen und sich die verwendeten thread-Konzepte als nicht verträglich erwiesen haben.

[5] Daneben ist natürlich die Gestaltung der Klassenstruktur des Java-API ein eigener Problemkomplex; dieser ist aber nicht Gegenstand dieses Papiers.

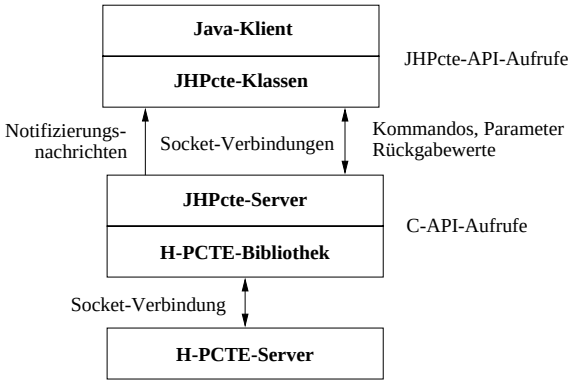


Abbildung 3. Prozeßstruktur

2. H-PCTE in Java: Wäre H-PCTE in Java implementiert, würde sich das obige Problem nicht stellen. Eine Reimplementierung von H-PCTE in Java ist jedoch wegen des enormen Umfangs (über 200 kLOC), dem daraus resultierenden Entwicklungsaufwand und der (noch) eingeschränkten Geschwindigkeit von Java-Anwendungen nicht praktikabel.
3. Verwendung eines Server-Prozesses: Ein in C oder C++ geschriebener Server (*JHPcte-Server*) wird als separater Betriebssystem-Prozeß ausgeführt und wandelt Aufrufe von Java-Klienten in Aufrufe der C-Funktionen von H-PCTE um. Java-Klient und JHPcte-Server kommunizieren über eine Socket-Verbindung. Die Kommunikation zwischen Klient und JHPcte-Server wird durch Java-Methoden realisiert, bei deren Aufruf ein Kommando und die übergebenen Parameter an den JHPcte-Server übertragen werden, die Funktion dort ausgeführt und der Fehlerstatus der ausgeführten Funktion sowie eventuelle Rückgabewerte zurückübertragen werden.

   Probleme mit threads treten hier nicht auf, da Klient und JHPcte-Server in getrennten Betriebssystem-Prozessen ablaufen. Der Aufwand für die Realisierung ist dabei nicht höher als bei der Verwendung von Native Methods: Zwar müssen grundlegende Kommunikationsfunktionen selbst implementiert werden, dafür entfällt die Einarbeitung in das recht umfangreiche NMI.

Offensichtlich ist nur der unter Punkt 3 vorgestellte Ansatz praktikabel. Abb. 3 zeigt die resultierende Prozeßstruktur, einen Java-Klienten, der über den JHPcte-Server auf den H-PCTE-Server zugreift.

Anders als in reinen Client-Server-Architekturen kann ein H-PCTE-Klient vom OMS über Änderungen benachrichtigt werden (Notifizierung). Um Java-Klienten notifizieren zu können, wird zwischen Java-Klient und JHPcte-Server eine zweite Socket-Verbindung etabliert, über die der JHPcte-Server Notifizierungsnachrichten an den Java-Klienten schickt (s. Abschnitt 5).

4 Das Prozeßkonzept

4.1 H-PCTE-Prozesse

Mehrere Anwendungen können parallel auf einer Objektbank arbeiten. Folglich muß der H-PCTE-Server mehrere Klienten-Prozesse parallel bedienen können. Hierzu wird jeder Klienten-Prozeß intern als *H-PCTE-Prozeß* verwaltet. Jeder H-PCTE-Prozeß arbeitet mit einer eigenen Sicht auf den Daten im OMS und besitzt einen eigenen Transaktions-[6] und Rechtekontext.

Eine Besonderheit des H-PCTE-Systems besteht darin, daß ein einzelner Betriebssystem-Prozeß (z.B. der JHPcte-Server) *mehrere* H-PCTE-Prozesse parallel starten kann. Beim Erzeugen eines H-PCTE-Prozesses wird ein POSIX-thread gestartet, in dessen Rahmen die neue Anwendung ausgeführt wird. Alle Aufrufe von C-API-Operationen finden innerhalb derartiger POSIX-threads statt. Der POSIX-thread identifiziert dabei *implizit* den zu verwendenden H-PCTE-Prozeß, d.h. in den C-API-Operationen braucht der jeweilige H-PCTE-Prozeß nicht explizit durch einen Parameter angegeben zu werden.

4.2 Identifizierung von H-PCTE-Prozessen im Java-Klienten

Abb. 4 zeigt Werkzeuge einer in Java realisierten SEU. Bei der in Abschnitt 2.2 vorgestellten Werkzeugarchitektur ist jedem Werkzeug-Fenster ein eigener H-PCTE-Prozeß zugeordnet. Die SEU ist als Java-Klient beim JHPcte-Server angemeldet.

Es liegt nun nahe, für jedes Werkzeug einen Java-thread zu erzeugen und diesem Java-thread einen H-PCTE-Prozeß zuzuordnen. Zugriffe auf die Objektbank innerhalb eines solchen threads werden implizit dem zugehörigen H-PCTE-Prozeß im JHPcte-Server zugeordnet und in dessen Kontext durchgeführt. Dieser Ansatz bedingt, daß der Java-thread während der gesamten Lebensdauer des Fensters existiert und nach Abarbeitung jedes Benutzer-Ereignisses (z.B. Betätigung einer Maustaste) in einen Wartezustand versetzt bzw. nach Eintreten des nächsten Benutzer-Ereignisses wieder aufgeweckt wird. Dies ist aber mit dem AWT nicht mit vertretbarem Aufwand möglich: Der Aufbau des Fensterinhaltes und die Reaktionen auf Benutzer-Ereignisse finden innerhalb eines vom Java-Laufzeitsystem verwalteten threads statt. Um einen anderen thread diese Aufgaben ausführen zu lassen, müßte eine Kommunikation zwischen den threads realisiert werden, was die Handhabung der threads deutlich erschweren würde. Noch gravierendere Probleme entstehen durch die verteilte Notifizierung, bei der asynchron Ereignisse auftreten (s. Kap. 5). Aus diesen Gründen ist der Ansatz, pro Fenster einen thread zu benutzen, nicht praktikabel.

Eine gangbare Alternative besteht darin, jeden H-PCTE-Prozeß durch ein *Prozeß-Objekt* im Java-Klienten zu repräsentieren. Jedem Prozeß-Objekt ist ein

[6] Durch die Verwendung von Transaktionen wird die Konsistenz der Daten bei komplexen Benutzeraktivitäten garantiert. Weiterhin ist die einfache Realisierung einer *Undo*-Funktionalität möglich, indem die im OMS durchgeführten Änderungen bis zu einem vorher zu setzenden *save point* zurückgesetzt werden.

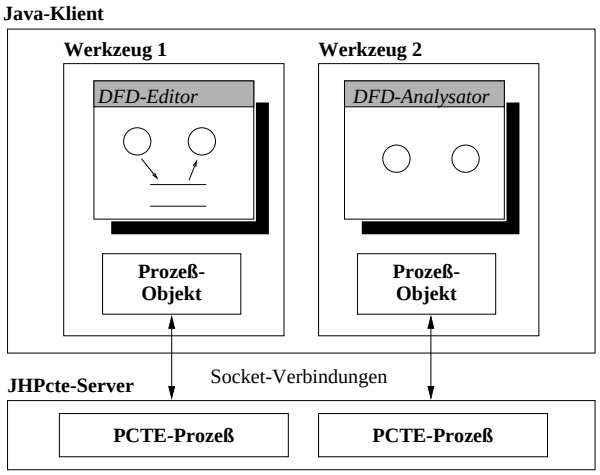


Abbildung 4. Repräsentation von H-PCTE-Prozessen

H-PCTE-Prozeß im JHPcte-Server zugeordnet. Abb. 4 zeigt einen DFD-Editor und einen DFD-Analysator, die über ihre zugeordneten Prozeß-Objekte auf die Objektbank zugreifen. Zugriffe auf die Objektbank werden im Kontext des zum Prozeß-Objekt gehörigen H-PCTE-Prozesses im JHPcte-Server ausgeführt.

4.3 Das Java-API

Im folgenden geben wir die Klassen bzw. Methoden des Java-APIs von H-PCTE an, die für die Prozeßhandhabung relevant sind. Ihre Anwendung ist sehr einfach und wird anhand einiger Beispiele verdeutlicht.

Die Klasse HPcte. Mit den Methoden dieser statischen Klasse kann ein Klient H-PCTE-Prozesse erzeugen.

```
PcteProcess p;

// H-PCTE-Prozeß erzeugen
p = HPcte.HPcte_lwp_login_create_and_start ( user, password );
```

Die Methode liefert ein Objekt, das den erzeugten H-PCTE-Prozeß repräsentiert. Als Parameter werden der Benutzername und das Benutzerpasswort übergeben; diese Angaben werden für die Authentifizierung des Klienten benötigt[7].

[7] H-PCTE bietet Zugriffsrechte auf Objektbasis und eine Benutzer- und Benutzergruppenverwaltung an, auf die aus Platzgründen nicht weiter eingegangen werden kann.

Die Klasse PcteProcess. Instanzen dieser Klasse (*Prozeß-Objekte*) repräsentieren H-PCTE-Prozesse. Der Klasse sind Schnittstellen-Operationen zugeordnet, die den Prozeß betreffen, z.B. zum Setzen und Abfragen seines externen Schemas.

Im folgenden Beispiel wird an einem Objekt, das einen DFD-Prozeß repräsentiert, dessen Name aus dem Attribut `process_name` ausgelesen. Zunächst wird ein externes Schema für den H-PCTE-Prozeß gesetzt, in dem diese Daten sichtbar sind. Danach wird eine Objektreferenz auf das DFD-Objekt erzeugt; Objekte können in H-PCTE durch einen textuellen Pfadnamen identifiziert werden, der als Parameter übergeben wird und dessen Details hier nicht weiter relevant sind.

```
PcteObjectReference DFD_Diagram_Process;

// externes Schema des Prozesses setzen
p.Pcte_process_set_ws ( ¨DFD_document¨ );

// Objektreferenz zu DFD-Prozeß-Objekt erzeugen
DFD_Diagram_Process =
     p.Pcte_object_reference_set_absolute ( ¨/.../...¨ );
```

Die zurückgelieferte Objektreferenz wird für diverse Operationen auf dem Objektbank-Objekt benutzt. Sie beinhaltet intern einen Verweis auf das Prozeßobjekt `p`, in dessen Kontext sie erzeugt wurde, d.h. das Java-Objekt `p` braucht bei diesen Operationen nicht mehr angegeben zu werden.

Die Klasse PcteObjectReference. Instanzen dieser Klasse stellen Referenzen auf H-PCTE-Objekte dar. Auf Objekte und Links in der Objektbank kann ausschließlich über Objektreferenzen zugegriffen werden; daher ist dieser Klasse der größte Teil der Schnittstellenoperationen zugeordnet. Hierzu zählen Operationen zum Erzeugen und Löschen von Objekten und Links sowie zum Lesen und Schreiben von Attributen. Der Wert eines Attributs (hier das Attribut `process_name`) kann wie folgt ausgelesen werden:

```
PcteAttributeValue value;
// Name des DFD-Prozesses auslesen
value = DFD_Diagram_Process.Pcte_object_get_attribute (¨process_name¨);
```

5 Benachrichtigung von Werkzeugen

5.1 Das Notifizierungskonzept von H-PCTE

Das in H-PCTE realisierte Notifizierungskonzept basiert darauf, daß H-PCTE-Prozesse *Notifizierer* an Objekten, Links und Attributen in der Objektbank installieren können, um sich über Änderungen an diesen Ressourcen informieren zu lassen. Tritt eine überwachte Änderung auf, ruft das OMS automatisch den überwachenden H-PCTE-Prozeß auf. Beim Rückruf wird eine detaillierte Ereignisbeschreibung mitgeliefert, die neben der Art des aufgetretenen Ereignisses den neuen Zustand der Objektbank enthält. Mit Hilfe dieses Mechanismus kann auf sehr einfache Art und Weise gewährleistet werden, daß alle Fenster, unabhängig davon, wo sie ausgeführt werden, stets den aktuellen Zustand der Daten in der Objektbank anzeigen [14].

5.2 Notifizierung in JHPcte

Die Umsetzung des H-PCTE-Notifizierungskonzepts in der Java-API von H-PCTE soll anhand eines Beispiels erläutert werden (Abb. 5).

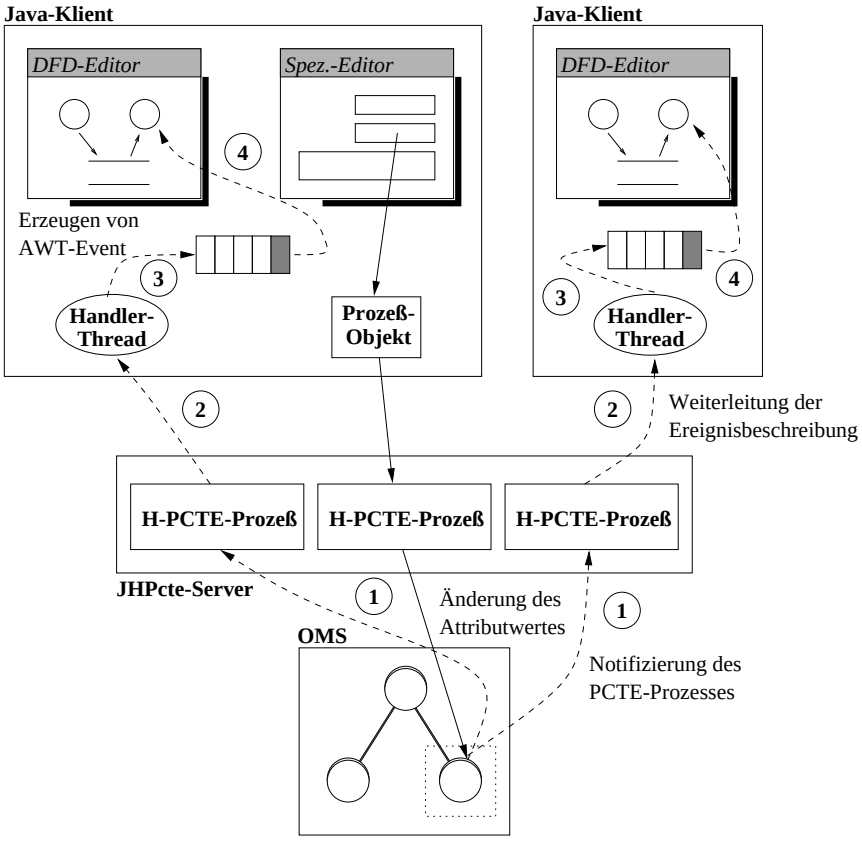


Abbildung 5. Notifizierung

Ausgangssituation ist, daß in einem Java-Klienten ein DFD-Editor und ein Spezifikationseditor ausgeführt werden, mit denen der Entwickler A ein DFD editiert. Auf einem anderen Rechner im Netz läuft ein weiterer Java-Klient, in dem dasselbe DFD durch einen DFD-Editor für Entwickler B angezeigt wird. Einer der DFD-Prozesse in diesem Diagramm besitzt ein zugeordnetes verfeinerndes DFD, an dem Entwickler B arbeitet. B will über alle Veränderungen an dem zu verfeinernden DFD-Prozeß sofort informiert werden.

Hierzu installiert der DFD-Editor Notifizierer an allen vom ihm angezeigten Daten. Das folgende Beispiel zeigt die Anmeldung eines Notifizierers an dem At-

tribut `process_number` des Objekts, auf das die Objektreferenz `DFD_Diagram_Process` verweist.

```
PcteNotifier noti;

// Notifiziererobjekt zur Überwachung des Attributs erzeugen
noti = p.HPcte_notifier_create( DFD_Diagram_Process, ¨process_number¨,
                                HPcte.HPCTE_OBJECT_MODIFY_EVENT );
```

Entsprechend müssen für alle anderen angezeigten Objekte, Links und Attribute passende Notifizierer in der Objektbank angemeldet werden.

Sofern nun eine überwachte Ressource verändert wird, löst das OMS automatisch eine Benachrichtigung aus. Nehmen wir z.B. an, der Entwickler A ändert das Feld *Process Number* in seinem Spezifikationseditor, wodurch das Attribut `process_number` des Objekts, das den DFD-Prozeß in der Objektbank repräsentiert, geschrieben wird. Hierauf folgt:

1. Das OMS stellt fest, welche der installierten Notifizierer von der Änderung betroffen sind. Für die H-PCTE-Prozesse, die zu diesen Notifizierern gehören, wird im JHPcte-Server eine callback-Funktion aufgerufen. Im Beispiel sind dies also die beiden H-PCTE-Prozesse, die für die beiden DFD-Editoren ausgeführt werden[8].
2. Die zu den H-PCTE-Prozessen gehörigen threads übertragen über einen *Notifizierungssocket* (s.u.) die Ereignisbeschreibung an ihre Java-Klienten.
3. Die Ereignisbeschreibung wird im Java-Klienten vom *Notification-Handler*-thread empfangen. Dieser spezielle thread wird bei der Erzeugung eines Prozeßobjekts automatisch gestartet. Er ist nur für den Empfang von Ereignisbeschreibungen und deren Weiterleitung zuständig. Trifft eine Ereignisbeschreibung über den Socket ein, dann erzeugt er ein spezielles AWT-Event, das die Ereignisbeschreibung enthält, und fügt es in die Event-Warteschlange ein.
4. Das Event-Objekt wird an alle angemeldeten *Event-Listener*-Objekte weitergeleitet [5]. Als Listener können sich beliebige Java-Objekte an dem Notifziererobjekt anmelden, die die benötigte Schnittstelle zur Verarbeitung von Notifizierungsereignissen implementieren. Dies können zum Beispiel spezielle Eingabefelder oder Auswahllisten in Dialogfenstern sein, die Daten aus der Objektbank anzeigen und automatisch bei Änderungen aktualisiert werden sollen.
   Im Beispiel muß entsprechend das graphische Objekt, das die `process_number` des DFD-Prozesses innerhalb des Fensters anzeigt, als Event Listener angemeldet sein. Es erhält nun das Event, woraufhin es mittels der Ereignisbeschreibung seine Anzeige aktualisieren kann.

Allgemein können beliebige Objekte auf Notifizierungen reagieren, sofern sie eine entsprechende Methode implementieren, die die Notifizierungsereignisse verarbeiten kann. Das folgende Beispiel zeigt eine Methode, die auf die Änderung des

[8] Tatsächlich kommt noch der Spezifikationseditor von Entwickler A hinzu; dieses Detail ist hier, um übersichtlich zu bleiben, weggelassen worden.

Attributs `process_number` reagiert. Der neue Wert des Attributs wird mit der Zugriffsoperation `getAttributeValue` aus der Ereignisbeschreibung ermittelt.

```
public void pcteNotify ( PcteNotificationEvent e ) {
  PcteEventDescription descr;

  // Beschreibung des Ereignisses
  descr = e.getPcteEventDescription ();

  // Art des Ereignisses ermitteln
  switch ( descr.getAccessEvent() ) {
    case HPcte.HPCTE_OBJECT_MODIFY_EVENT:
      // Attribut process_number geändert?
      if ( descr.getAttributeName().equals(¨process_number¨) ) {
         // Anzeige im Textfeld aktualisieren
         window.setText ( descr.getAttributeValue().getStringValue() );
       }
       ...
    }
}
```

6 Zusammenfassung und Bewertung

Dieses Papier hat einen Ansatz vorgestellt, durch den sich verteilte Editier-Umgebungen leicht realisieren lassen. Die textuellen oder graphischen Benutzungsschnittstellen sind in Java geschrieben und daher leicht verteilbar. Sie kommunizieren mit einem Java-API-Server und indirekt mit einem aktiven Repository, das Änderungen an den Daten netzwerkweit und sichtenübergreifend propagiert und diverse andere Dienste (Transaktionen, Zugriffskontrollen) anbietet.

Die Grundzüge dieser Architektur wurden bereits in [3, 10] beschrieben und in einer SEU implementiert, die auf dem X Window System basierte. Als sehr problematisch erwies sich, daß die meisten X-Implementierungen nicht thread-sicher sind.

Als nachteilig stellte sich heraus (s. Abschnitt 4.2), daß Java-threads nicht unterbrechbar sind und nicht untereinander kommunizieren können; eine etwas elegantere Lösung der Notifizierung wäre möglich, wenn Java-threads ähnlich leistungsfähig wie POSIX-threads wären.

Ferner wäre es bei der Konstruktion von diversen Dialog-Elementen praktisch, wenn man gemeinsame Unterklassen von OMS-Klassen (Beispiele wurden in Abschnitt 4.3 vorgestellt) und GUI-Klassen bilden könnte. Leider ist dies nicht möglich. Insgesamt überwiegen aber die Vorteile der Java-basierten Architektur ganz deutlich gegenüber der X-basierten Architektur.

Literatur

1. Atkinson, M.; Bailey, P.; et al: An approach to Persistent Programming; p.141-146 in: Readings in Object-Oriented Database Systems; Morgan Kaufmann; 1990

2. Bernstein, P.A.: Repositories and Object Oriented Databases; p.34-46 in: Proc. Datenbanksysteme in Büro, Technik und Wissenschaft; Springer-Verlag; 1997
3. Däberitz, D.: Entwurf und Realisierung von Software-Entwicklungsumgebungen unter Berücksichtigung von Funktionen und Merkmalen von Nicht-Standard-Datenbanken; Dissertation, Univ. Siegen; 1997
4. Dittrich, K.; Kotz, A.; Müller, J.; Lockemann, P.: Datenbankunterstützung für den ingenieurwissenschaftlichen Bereich; Informatik-Spektrum 8, p.113-125; 1985
5. Flanagan, D.: Java in a Nutshell, P.; O'Reilly & Associates, Inc.; 1997
6. Henrich, A.; Kelter, U.: Integration von Zugriffsparadigmen in einem Repository; p.307-326 in: Beherrschung von Informationssystemen (Proc. Informatik'96, Klagenfurt, 25.-27.09.1996); Schriftenreihe der ÖCG, Band 88, R. Oldenbourg; 1996/09
7. Heuer, A.: Objektorientierte Datenbanken - Konzepte, Modelle, Systeme; Addison-Wesley; 1992
8. H-PCTE – Release note for Version 3.0; Universität Siegen; 1998 (erhältlich über `http://pi.informatik.uni-siegen.de`)
9. Kelter, U.: H-PCTE – a high-performance object management system for system development environments; p.45-50 in: Proc. COMPSAC '92; IEEE Press; 1992
10. Kelter, U.; Däberitz, D.: An Assessment of Non-Standard DBMSs for CASE Environments; p.96-113 in: Proc. Int. Conf. on Extending Database Technology; LNiCS 1057; Springer-Verlag; 1996
11. Kemper, A.; Moerkotte, G.: Object-oriented Database Management; Applications in Engineering and Computer Science; Prentice Hall; 1994
12. Lea, D.: Concurrent Programming in Java Design Principles and Patterns; Addison-Wesley; 1998
13. Meyers, S.: Difficulties in Integrating Multiview Environments; IEEE Software 8:1, p.49-57; 1991
14. Platz, D.; Kelter, U.: Konsistenzerhaltung von Fensterinhalten in Software-Entwicklungsumgebungen; Informatik – Forschung und Entwicklung 12:4, p.196-205; 1997/12
15. Portable Common Tool Environment - Abstract Specification / C Bindings / Ada Bindings (Standards ECMA-149/-158/-165 und ISO IS 13719-1/-2/-3); 1994
16. Rao, B.: Object-oriented Databases – Technology, Applications, and Products; McGraw-Hill; 1994
17. Wakeman, L.; Jowett, J.: PCTE the Standard for Open Repositories; Prentice Hall; 1993
18. Zdonik, S.: What makes Object-Oriented Database Management Systems different? p.3-28 in: Advances in Object-Oriented Database Systems; NATO ASI Series F, Computer and Systems Sciences, Vol.130; Springer Verlag; 1994