Erscheint im Tagungsbericht der GI-SI-Jahrestagung 1995

Verteilungstransparenz bei der objektorientierten Spezifikation verteilter Applikationen

Klaus-Peter Löhr

Bericht B-95-08
Mai 1995

Zusammenfassung: Verteilte Applikationen werden heute zunehmend verteilungstransparent programmiert. Wenn man schon bei der Programmierung weitgehend von verteilungsbedingten technischen Details abstrahieren kann, erhofft man sich dies natürlich erst recht für die Spezifikation. Bei genauerem Hinsehen zeigt sich allerdings, daß manche Transparenzdefizite der Implementierung schon bei der Spezifikation berücksichtigt werden müssen, und zwar auch bei sorgfältiger objektorientierter Strukturierung. Die vorliegende Arbeit erläutert die Problematik anhand einer in Object-Z formulierten Beispielapplikation und zeigt auf, was beim Entwurf zu beachten ist.

lohr@inf.fu-berlin.de

Institut für Informatik
Fachbereich Mathematik und Informatik
Freie Universität Berlin
Takustr. 9
D-14195 Berlin

http://www.inf.fu-berlin.de

1 Einführung

Softwaretechnik für verteilte Applikationen umfaßt traditionell die Auseinandersetzung mit Nichtsequentialität, Interprozeßkommunikation und Kommunikationsprotokollen. Dieses Bild beginnt sich zu wandeln: zunehmend gelingt es, die schwierig zu handhabende und fehleranfällige Kommunikation in die Basissoftware zu verlagern; Werkzeuge wie Stub-Generatoren erlauben verteilungstransparente Fernaufrufe, und das dynamische Binden über Rechnergrenzen hinweg wird durch Konfiguratoren, Namensdienste und Dienstvermittlungen unterstützt. Es ist eine generelle Tendenz erkennbar, eine möglichst hochgradige *Verteilungstransparenz* zu erreichen und damit der Entwicklung verteilter Applikationen ihre spezifischen Schwierigkeiten zu nehmen[1].

Vor diesem Hintergrund stellt sich die Frage, ob in Zukunft die Entwicklung verteilter Applikationen überhaupt noch verteilungsbedingte Anforderungen an die Softwaretechnik stellen wird. Dies gilt erst recht für die frühen Phasen der Software-Entwicklung - die ja noch weiter von den technischen Details einer verteilten Realisierung abstrahieren sollten als die Implementierung. Dabei ist es irrelevant, ob ein objektorientierter Ansatz verfolgt wird oder nicht. Spezielle objektorientierte Programmiersprachen mit verteilter Implementierung erreichen eine sehr gute Verteilungstransparenz [Raj u.a. 91], und für offene Systeme zeigt die CORBA-Initiative [OMG 91], daß Verteilungstransparenz nicht an Heterogenität scheitern muß. Selbst Vererbung kann über Rechnergrenzen hinweg praktiziert werden, ohne daß im Programm etwas von der Verteilung sichtbar wird [Wolff 95].

Nun gibt es selbst bei hundertprozentiger Verteilungstransparenz zwei Aspekte, mit denen eine "normale" Softwaretechnik nicht befaßt ist: Nichtsequentialität und Konfigurierung. Zur *Nichtsequentialität* ist zu beachten, daß die Eigenschaft einer Applikation, sequentiell bzw. nichtsequentiell zu sein, nichts mit der unterliegenden - für den Applikationsentwickler nicht sichtbaren - verteilten oder nichtverteilten Implementierung zu tun hat. So wie ein nichtsequentielles Programm nichtverteilt ausgeführt werden kann, kann auch ein sequentielles Programm verteilt ausgeführt werden. Bei verteilten Klienten/Anbieter-Systemen wird man zwar in der Regel nichtsequentiellen Code finden: ein Anbieter fungiert als öffentlicher Dienstleister in einer aus unabhängigen Klienten bestehenden nichtsequentiellen Umgebung und erlaubt häufig die überlappende Ausführung mehrerer Klientenaufträge (mittels Threading). Diese Nichtsequentialität des Anbieters hängt aber *nicht ursächlich* mit der Verteilung zusammen.

Originär verteilungsbedingt ist dagegen die *Konfigurierung*, und es ist kein Zufall, daß die Fortschritte bei der Verteilungstransparenz von einem zunehmenden Interesse an Konfigurierungsfragen begleitet sind [Kramer/Purtilo 94]. Die Konfigurierung legt fest, aus welchen Modulen bzw. Klassen eine lokale Komponente einer verteilten Applikation zusammengesetzt ist, wie die Komponenten im Netz plaziert werden und wie das dynamische Binden zwischen Komponenten erfolgt. Die angestrebte Konfiguration wird eventuell mit Hilfe einer formalen Konfigurationssprache spezifiziert, die von einem Konfigurationswerkzeug

[1] Das englische „distribution transparency“ meint die Unsichtbarkeit, das Verbergen der Verteilung („transparent“ im Sinne von unsichtbar weil durchscheinend). Der Entwickler soll von der Verteilung *abstrahieren* können. *Verteilungsabstraktion* wäre daher im Deutschen ein treffenderer Begriff - ist allerdings kaum gebräuchlich.

(Konfigurator) interpretiert werden kann - womit der Konfigurierungsvorgang automatisiert wird [Löhr u.a. 94].

Sieht man von der Konfigurierung ab und fragt nach dem Einfluß der Verteilung auf die objektorientierte Spezifikation der Applikation, so ist die Antwort zunächst: wegen der Verteilungstransparenz ist die Spezifikation *unabhängig* von der Verteilung. Die Frage wäre demnach genauso unsinnig wie etwa - um ein hardwarenäheres Analogon zu wählen - die nach "objektorientierter Programmierung von RISC-Maschinen".

Bei genauerem Hinsehen werden allerdings Effekte sichtbar, die eine punktuelle Berücksichtigung der Verteilung bereits bei der Spezifikation nahelegen. Nicht alles, was lokal möglich ist, kann verteilt "mit entsprechend geringerer Effizienz" genauso realisiert werden. Dies gilt bereits für den klassischen Fernaufruf [Tanenbaum/Renesse 88]. Die Effizienzhürden können so groß sein, daß man gewisse semantische Abweichungen in Kauf nimmt, d.h. auf eine hundertprozentige Verteilungstransparenz verzichtet. Der springende Punkt ist, daß die Abstriche an der Verteilungstransparenz auf den Entwurf durchschlagen können und somit nicht erst bei der Implementierung, sondern bereits bei der Spezifikation berücksichtigt werden müssen. Wird dies ignoriert, so kann später, wenn die Probleme bei der Implementierung bzw. Konfigurierung zu Tage treten, ein kostspieliges Redesign erforderlich werden.

Die geschilderte Problematik wird in dieser Arbeit anhand einiger Beispiele verdeutlicht. Bei objektorientierten Spezifikationen kann man - wie bei anderen Spezifikationen auch - zwischen *Systemspezifikation* und *Entwurfsspezifikation* unterscheiden. Erstere ist abstrakter; ihre Struktur präjudiziert im Gegensatz zur Entwurfsspezifikation *nicht* die Architektur des implementierten Systems. Häufig sind Spezifikationen nicht eindeutig der einen oder anderen Kategorie zuzuordnen. Im folgenden wird der Entwurfscharakter im Vordergrund stehen. Wir werden uns zunächst auf den Standpunkt stellen, daß dank der Verteilungstransparenz die Spezifikation einer verteilten Applikation sich nicht von der Spezifikation einer nichtverteilten - möglicherweise aber nichtsequentiellen - Spezifikation unterscheidet. Dies wird in Abschnitt 2 anhand eines Beispiels demonstriert. Bei genauerem Hinsehen zeigt sich jedoch, daß dieser Standpunkt nicht unproblematisch ist, weil die Qualität eines Entwurfs letztlich nicht unabhängig von der Konfigurierung beurteilt werden kann. In den Abschnitten 3-5 wird dies für drei verschiedene Problemklassen demonstriert:

- Umlenkung lokaler Objektaufrufe,
- Semantik asynchroner Objektaufrufe,
- Konsistenz verteilter Objekte.

Diese Liste erhebt keinen Anspruch auf Vollständigkeit. Ziel dieser Arbeit ist, einige typische Stellen zu identifizieren, wo bei verteilungstransparentem Spezifizieren eine Berücksichtigung der späteren Verteilungsstruktur angebracht sein kann, will man unpraktikable Entwürfe vermeiden.

2 Verteilungstransparente objektorientierte Spezifikation

Unter dem Postulat der Verteilungstransparenz können wir bei der Spezifikation die Verteilung der Implementierung ignorieren. Als Referenzbeispiel betrachten wir die vereinfachte Version eines typischen Problems der rechnergestützten Gruppenarbeit: so wie mehrere Personen in einem Raum gemeinsam an einer Tafel arbeiten können, sollen mehrere räumlich getrennte Personen an einer gemeinsamen "virtuellen Tafel" arbeiten können, die jede Person

als Fenster auf ihrem Bildschirm sieht. Damit das Problem etwas interessanter wird, soll es mehrere Tafeln geben können, die namentlich identifizierbar sind [Levy/Tempero 91].

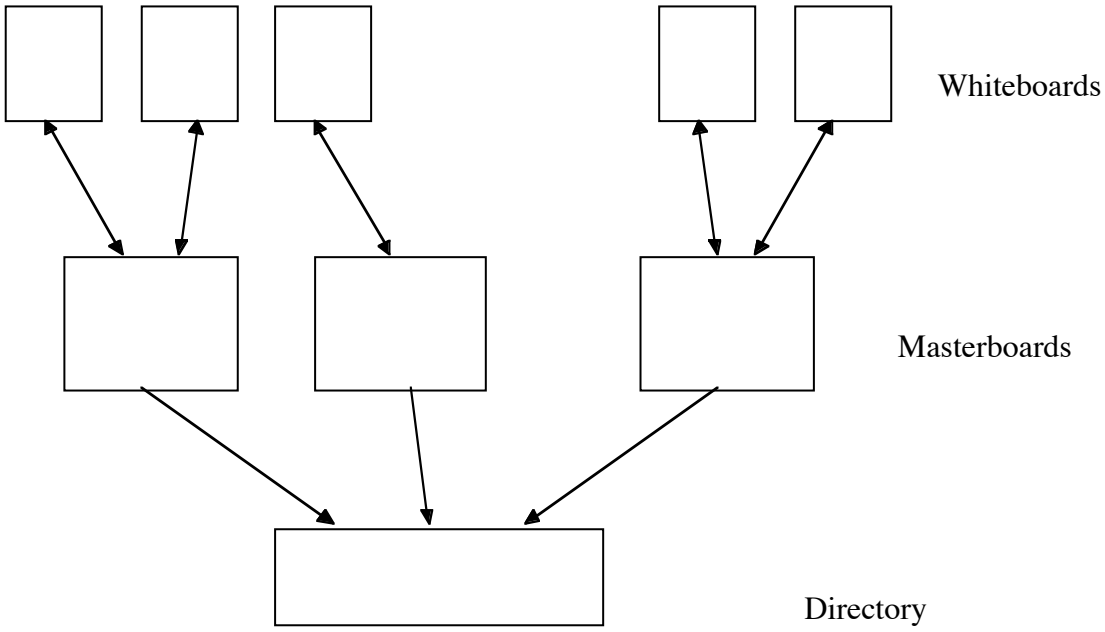

Abb. 1 Architektur einer einfachen verteilten Applikation

Um von den Details eines Fenstersystems zu abstrahieren, setzen wir einen Typ *Item* voraus, der für ein Element textueller oder graphischer Information steht, ferner eine Klasse *Window* mit einer Operation *draw*, die ein *Item* in einem Fenster plaziert; *drawpict* tut dies für eine Folge von *Item*s. Ein Programm verschafft sich Zugang zu einer Tafel mit Namen *name*, indem es ein Objekt der Klasse *Whiteboard* erzeugt und die Operation *initialize* aufruft - unter Beigabe des Parameters *name* sowie eines Parameters *dir*, der auf ein Verzeichnis von Tafeln verweist. *Whiteboard* ist Unterklasse von *Window* und besorgt sowohl die Manipulation des lokalen Fensters als auch die Verteilung der Information auf die anderen beteiligten Fenster (über eine Klasse *Masterboard*). Abb. 1 zeigt ein Szenario mit einigen Objekten; ein Pfeil zwischen zwei Objekten bedeutet, daß es eine entsprechende Aufrufmöglichkeit gibt.

Die Wahl der Spezifikationssprache hat keinen entscheidenden Einfluß auf den hier darzulegenden Sachverhalt. Wir wählen *Object-Z* [Duke u.a. 91] [Rose/Duke 94] [Duke u.a. 94], eine der objektorientierten Erweiterungen von Z [Wordsworth 92]. Object-Z erlaubt die Kapselung der von Z bekannten *Schemata* zu *Klassenschemata*, die Bezugnahme auf *Objekte* über Verweise und die *Vererbung* durch Schema-erweiterung. Bei einer Klasse kann zudem neben der üblichen, die erlaubten Zustände charakterisierenden Klasseninvariante eine *Geschichtsinvariante* angegeben werden, die mittels temporaler Logik die möglichen Zustandsfolgen einschränkt.

Abb. 2 zeigt die Klasse *Whiteboard*. Die von *Window* geerbte Operation *draw* wird in *windraw* umbenannt. Eine neu eingeführte Operation *draw* (letzte Zeile) sorgt dafür, daß der auf der Tafel zu bewirkende Effekt nicht nur im lokalen Fenster, sondern auch in den zugehörigen Fenstern auf anderen Stationen sichtbar wird. Für die Verwaltung des Tafelinhalts und der beteiligten Klienten ist ein *Masterboard* zuständig, das über den Verweis *master* erreicht wird. Der Anfangszustand eines *Whiteboard* wird unter Benutzung eines *Directory* so einge-

richtet, daß *master* auf das *Masterboard* verweist, das *Whiteboard* als Beteiligter beim *Masterboard* verzeichnet ist und im zugehörigen lokalen Fenster der Tafelinhalt sichtbar ist. (|| bedeutet Konjunktion mit Identifizierung der Ausgabe des 1. Operanden mit der Eingabe des 2. Operanden.) Auch eine Finalisierung ist vorgesehen.

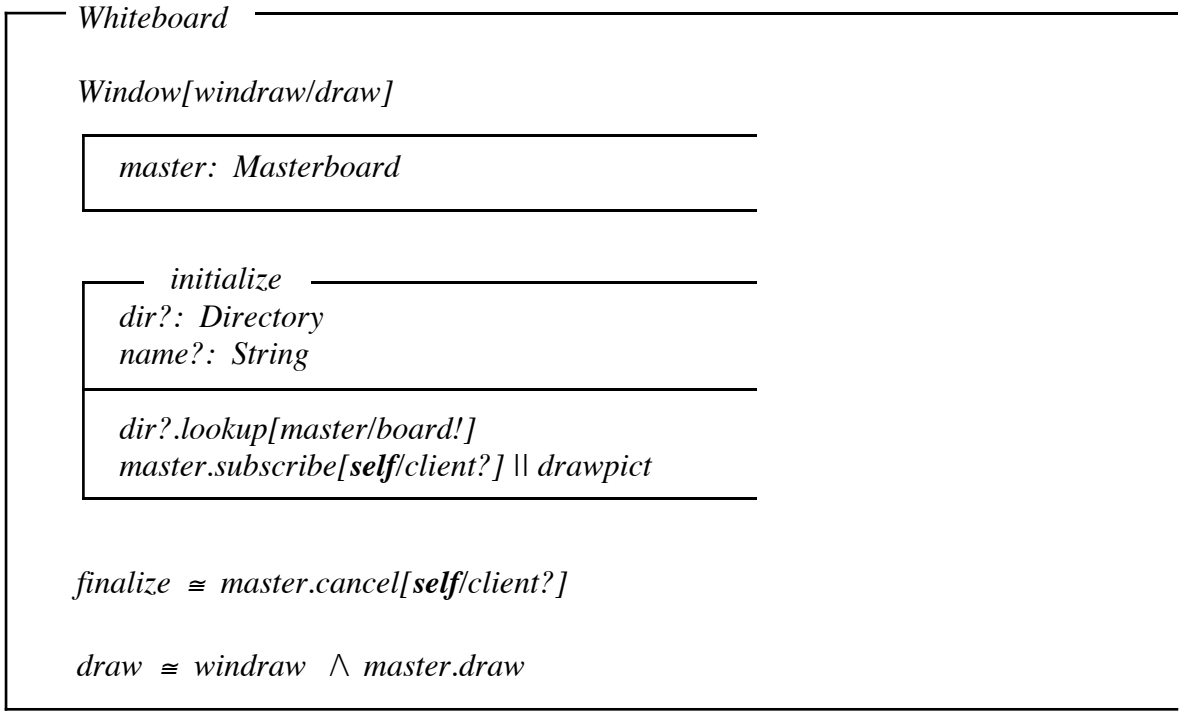

Abb. 2 Klassenschema *Whiteboard*

Abb. 3 zeigt die Klasse *Masterboard*. Sie hat drei Zustandskomponenten: *dir* verweist auf das zugehörige *Directory*, *picture* umfaßt die bisher angezeigten *Item*s, *clients* die Teilnehmer (***F*** ist in Z der Typkonstruktor „set of finite subsets of"). Wenn ein neuer Teilnehmer sich anmeldet (*subscribe*), wird ihm der aktuelle Tafelinhalt übergeben, damit er sein Fenster auf den aktuellen Stand bringen kann (*drawpict* in Abb. 2). Wenn der letzte Teilnehmer sich abgemeldet hat (*cancel*), wird die Tafel aus dem Verzeichnis *dir* entfernt. *draw* sorgt für die Verteilung eines neuen *Item* an alle Teilnehmer.

Die Klasse *Directory* ist in Abb. 4 gezeigt. Der Zustand eines Verzeichnisses ist eine Abbildung *board: String* → *Masterboard,* initialisiert mit „undefiniert" (*board* = ∅). Erwähnenswert bei *lookup* ist: wenn ein Teilnehmer nach einer nicht vorhandenen Tafel fragt, wird automatisch eine erzeugt. *delete* löscht das angegebene *Masterboard* aus dem Verzeichnis.

Die Spezifikation dieser verteilten Applikation macht keine Aussage über die Verteilung der Objekte der Klassen *Whiteboard, Masterboard, Directory*. Für die Funktionalität ist dies auch nicht relevant. Typischerweise wird man die *Whiteboard*-Objekte auf verschiedenen Stationen finden; sie werden dort von den Benutzern der Applikation erzeugt. Wo das *Directory* und die *Masterboard*s liegen, wissen die Benutzer nicht (und wollen sie auch nicht wissen). Bei der Konfiguration wird man natürlich darauf achten, daß eine effiziente Interaktion zwischen einem *Masterboard* und den zugehörigen *Whiteboard*s möglich ist.

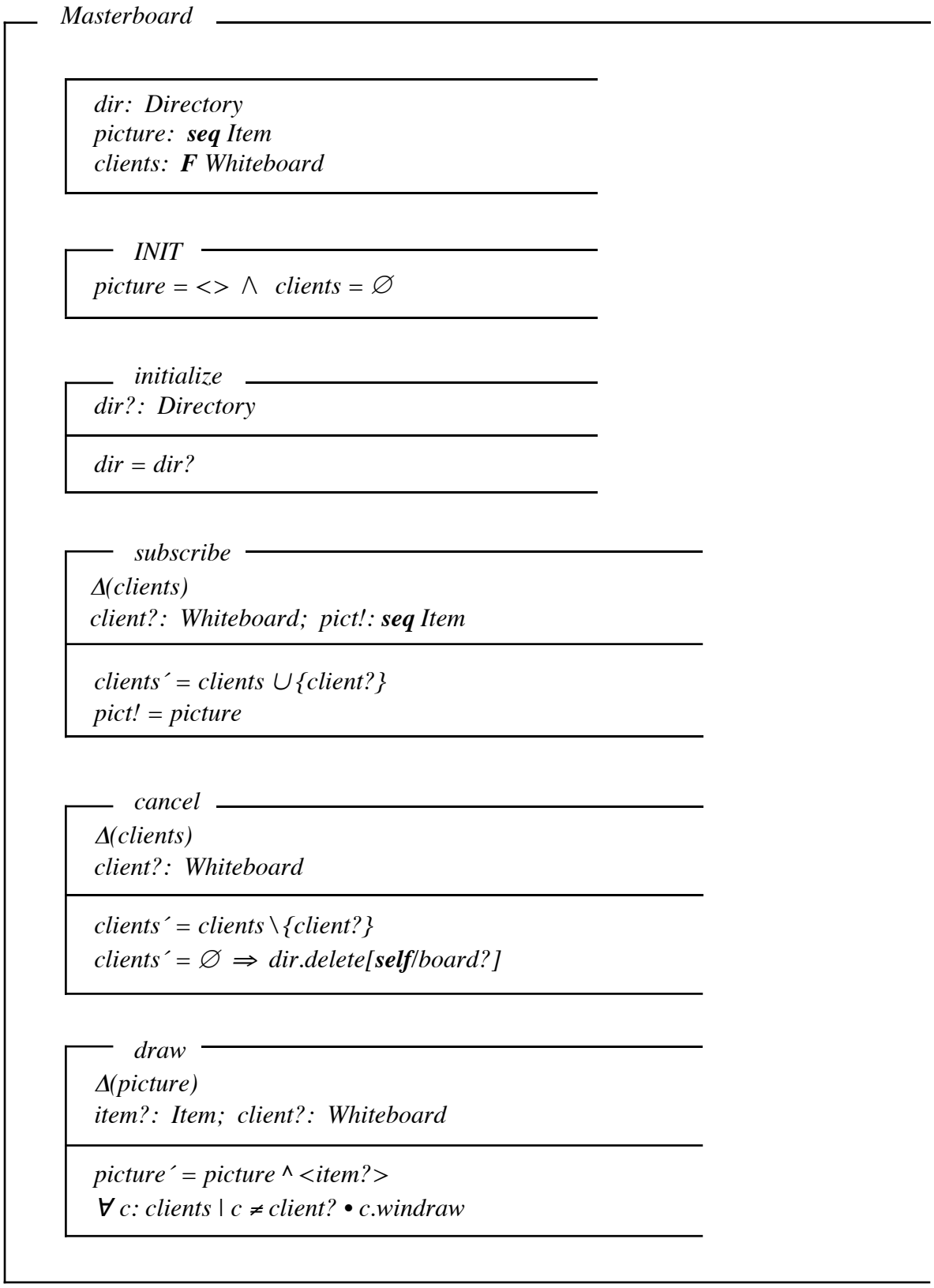

Abb. 3 Klassenschema *Masterboard*

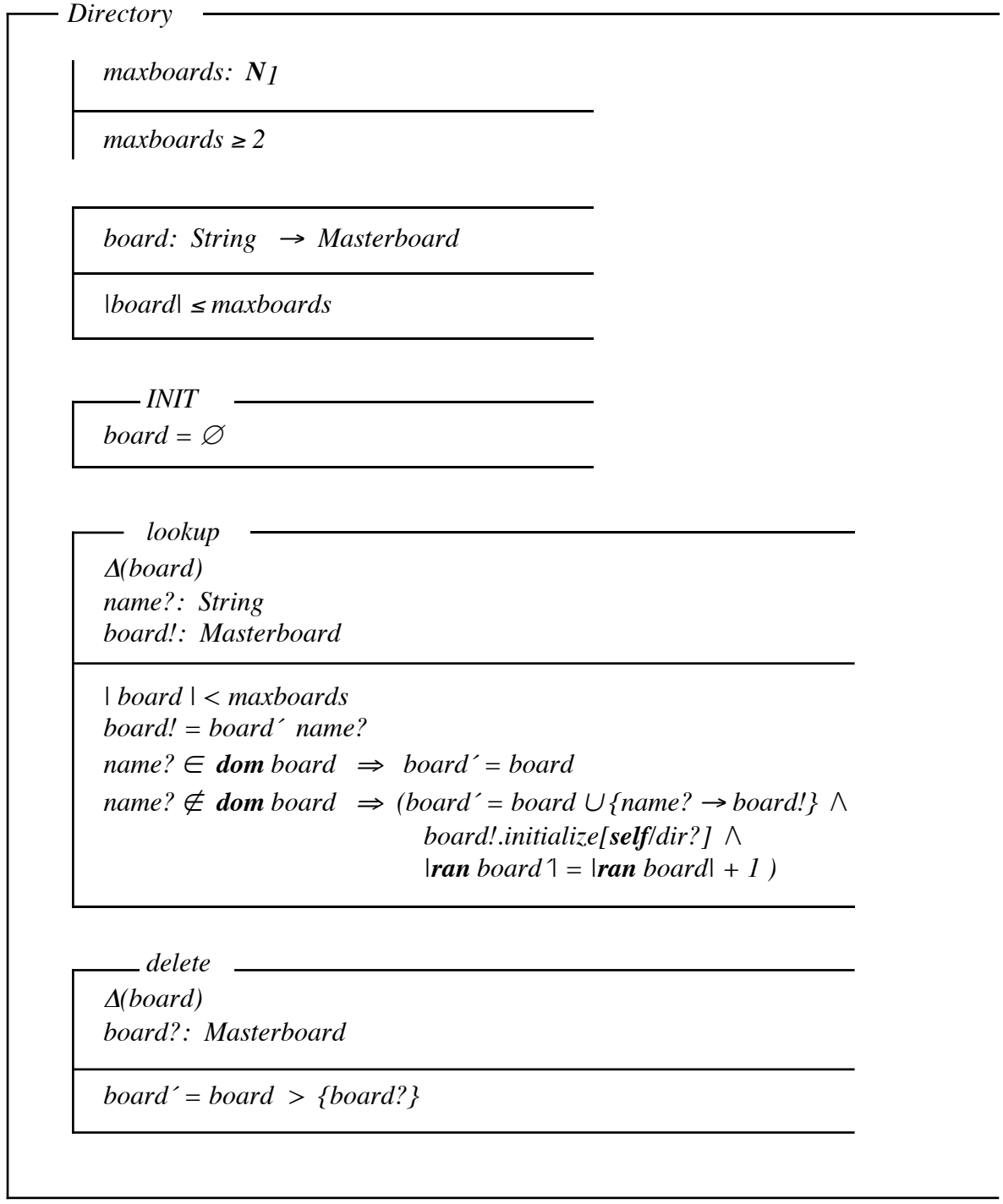

Abb. 4 Klassenschema *Directory*

Unabhängig von der Verteilung ist festzustellen: das System ist nichtsequentiell. Dies ist durch die nebenläufig erzeugten und operierenden *Whiteboard*s bedingt. Die Spezifikation enthält allerdings keine nichtsequentiellen Elemente. Ihre Semantik basiert auf atomaren Zustandsübergängen der Applikation. Diese werden ausgelöst durch Aktivierung von *White-board*-Operationen: Initialisierung, Finalisierung und *draw*. Bei der Implementierung ist man

nun frei, die Ausführung der Operationen nichtatomar zu gestalten, sofern man nur garantiert, daß Abläufe mit überlappenden Ausführungen serialisierbar sind. So werden zwei *Whiteboard*-Objekte, die verschiedenen Benutzern gehören, ihre jeweiligen Operationen grundsätzlich unabhängig voneinander beginnen, später aber beim Betreten eines *Masterboard*- oder *Directory*-Objekts womöglich einer Ausschlußsynchronisation unterworfen werden. Wieviel Spielraum dem Implementierer eingeräumt wird, hängt durchaus vom Geschick des Entwerfers ab. Auf diese Thematik soll hier aber nicht näher eingegangen werden.

3 Umleitung lokaler Objektaufrufe

Eine *verteilte Programmiersprache* zeichnet sich dadurch aus, daß der Übersetzer Kenntnis von der Verteilung hat. Damit kann ein Objektaufruf, wenn der zugrundeliegende Verweis sich auf ein entferntes Objekt bezieht, vom Laufzeitsystem abgefangen und geeignet weitergeleitet werden. Am Aufrufmechanismus für lokale Objekte ändert sich dadurch nichts. Die Situation ist anders, wenn bei einer nichtverteilten Sprache Stellvertreterobjekte zum Einsatz kommen, deren Klassen von einem Stub-Generator erzeugt werden. Da für die Klasse eines entfernten Objekts eine namensgleiche Stellvertreterklasse untergeschoben wird, muß ein Zugriff auf ein *lokales* Objekt der *gleichen* Klasse notwendig auch über einen Stellvertreter umgeleitet werden.

Zwar tritt dieses Problem in vielen Klient/Anbieter-Szenarien nicht auf, weil der Fernzugriff auf einen Anbieter häufig gerade wegen des Fehlens einer entsprechenden lokalen Funktionalität notwendig wird. Im allgemeinen muß man aber, wenn ein Programm verteilt ausgeführt werden soll, auf das Problem vorbereitet sein. Die Umleitung lokaler Objektaufrufe kann derart drastische Effizienzeinbußen zur Folge haben, daß man nach Wegen suchen wird, sie zu vermeiden. Im Grundsatz gilt: wenn für eine Klasse K häufig auf lokale K-Objekte zugegriffen wird, darf es keine Fernzugriffe auf K-Objekte geben. Es gibt dann die folgenden Alternativen:

1. Die Klasse K ist an *eine* Station gebunden, d.h. K-Objekte gibt es überhaupt nur dort (u.U. eine drastische Einschränkung!).

2. Für die entfernten Objekte wird eine Hilfsklasse L mit gleicher Funktionalität wie K verwendet; L kann als Unterklasse von K konzipiert werden.

3. Bei Fernaufrufen wird für K-Parameter nicht Verweisübergabe, sondern ausschließlich Wertübergabe praktiziert - sofern die damit geänderte Semantik akzeptabel ist (und die Programmiersprache das erlaubt!). Fernzugriffe auf K-Objekte entfallen damit.

Offensichtlich betreffen diese Überlegungen nicht nur die Implementierung; sie sind bereits für die Spezifikation relevant. In unserer Beispielapplikation sieht es auf den ersten Blick so aus, als käme *Item* als Problemkandidat in Frage. Glücklicherweise tritt das Problem aber *nicht* auf. Unabhängig davon, ob *Item* als Klassentyp definiert ist oder nicht, muß aus Effizienzgründen bei der Implementierung sowohl von *master.draw* als auch von *client?.windraw* der *Wert* von *Item* und nicht ein Verweis übergeben werden. Somit gibt es zwar Fernaufrufe, bei denen *Item*-Objekte von Station zu Station kopiert werden, aber es gibt keine Fernzugriffe auf solche Objekte selbst.

Bei einer etwas anderen Problemstellung und ungeschicktem Entwurf kann eine Umleitung lokaler Aufrufe leicht eingeschleppt werden. Wir betrachten ein *talk*-Szenario, bei dem lediglich zwei Partner miteinander kooperieren. Wir unterstellen einen Entwurf, bei dem die Klasse *Whiteboard* die Klasse *Window* nicht beerbt, sondern *benutzt*, und zwar für das eigene Fenster und das des Partners; bei *draw* wird für die beiden Fenster jeweils ein *windraw*-Aufruf vorgesehen. Dieser Entwurf führt zwangsläufig zu einer Implementierung, bei der der lokale Aufruf den Umweg über einen Stellvertreter machen muß.

4 Ausführungsreihenfolge bei Objektaufrufen

Nichtsequentielle objektorientierte Programmiersprachen erlauben häufig nicht nur synchrone, sondern auch asynchrone - d.h. nebenläufig zum Aufrufer ausgeführte - Operationen auf Objekten. Es ist auch möglich, in einem Objekt synchrone, asynchrone und spontane[2] Operationen beliebig zu kombinieren und gleichzeitig gewisse Überlappungen von Operationsausführungen zuzulassen [Löhr 93]. Bei der Spezifikation genügt es häufig, die Objekte als atomar zu betrachten (wie wir das oben sogar für das Gesamtsystem getan haben).

Das Verhalten eines solchen Objekts entspricht dem eines zyklischen sequentiellen Prozesses, der wiederholt Aufträge entgegennimmt, bei deren Ausführung eventuell Unteraufträge erteilt und eine Auftragsausführung gegebenenfalls mit der Bereitstellung eines Ergebnisses beendet. Ein Objektaufruf entspricht einer Auftragserteilung. Aufträge und Ergebnisse werden als Nachrichten zwischen den Prozessen ausgetauscht.

Nach [Lamport 78] definiert man für zwei Ereignisse A und B die Relation $A \rightarrow B$, in Worten "A ereignet sich vor B", wie folgt: $A \rightarrow B$ bedeutet, daß eine der folgenden Aussagen gilt:

1. ein Prozeß führt A vor B aus;
2. A ist das Absenden einer Nachricht, B ist das Empfangen dieser Nachricht;
3. Es gibt ein Ereignis C mit $A \rightarrow C$ und $C \rightarrow B$.

Damit ist auch die Bedeutung der Aussage "ein Objektaufruf A erfolgt vor einem Objektaufruf B" definiert, unabhängig davon, ob die aufgerufenen Operationen synchron oder asynchron sind. Wir bezeichnen die zugehörigen Ereignisse "Beginn der Operationsausführung" (= Auftragsannahme) mit a bzw. b. Es gilt $A \rightarrow a$ und $B \rightarrow b$.

Uns interessiert der Fall, daß A und B sich auf dasselbe Objekt beziehen. In diesem Fall ist es wünschenswert, daß mit $A \rightarrow B$ in der Regel auch $a \rightarrow b$ gilt (sofern nicht ein Guard $b \rightarrow a$ erzwingt). Wenn A sich auf eine synchrone Operation bezieht, ist das a priori gesichert. Bei einer nichtverteilten Implementierung ist $a \rightarrow b$ üblicherweise auch im asynchronen Fall gesichert, und zwar durch die unterliegende Implementierung der Interobjektkommunikation.

Bei einer verteilten Implementierung sieht es anders aus. Zu garantieren, daß die Ausführungsreihenfolge von Aufrufen eines Objekts der Aufrufreihenfolge entspricht, kann mit unverhältnismäßig großen Kosten verbunden sein - jedenfalls wenn der Bereich der Lokalnetze verlassen wird. Betrachten wir wiederum unsere Beispielapplikation! Als Kandidaten für Asynchronie bieten sich die *draw*- und *windraw*-Operationen an: sie haben keine Ergebnisse, auf die man warten müßte - was insbesondere bei Fernaufrufen ein wichtiger Aspekt ist.

[2] Ein *aktives Objekt* kann wiederholt *spontane* Operationen ausführen, ohne dazu von anderen Objekten aufgerufen worden zu sein.

Betrachten wir *draw* aus *Masterboard:* eine asynchrone Implementierung wäre sinnvoll; ist sie durch die Spezifikation gedeckt? Mit *wb:Whiteboard* muß eine Sequenz von zwei Operationsausführungen *wb.draw(a)* und *wb.draw(b)* die *Item*s *a* und *b* in dieser Reihenfolge nicht nur in das lokale Fenster, sondern auch in die anderen beteiligten Fenster bringen. Wenn *master.draw* asynchron arbeitet und die Implementierung die Ausführungsreihenfolge nicht garantiert, kann die Reihenfolge der lokalen *windraw*s von der Reihenfolge der entfernten *windraw*s abweichen. Ob sich das auf das Bild auswirkt, hängt von der Spezifikation von *windraw* ab (die wir hier nicht betrachtet haben).

Man sieht demnach, daß man beim Spezifizieren gut daran tut, sich beizeiten Gedanken über die Ausführungsreihenfolge asynchroner Fernaufrufe zu machen. Unter Umständen ist es vertretbar, die Spezifikation so abzuschwächen, daß eine wesentlich effizientere Implementierung möglich wird.

5 Konsistenz verteilter Objekte

Abschließend soll auf ein Synchronisationsproblem eingegangen werden, bei dessen Behandlung tunlichst die spätere Verteilung berücksichtigt wird. Das Problem entsteht, wenn mehrere Objekte andere Objekte *gemeinsam* benutzen. Abb. 5(a) zeigt zwei Anbieter, die von zwei unabhängig arbeitenden Klienten benutzt werden (in Abb. 1 tritt diese Situation nicht auf). Für die Klienten seien Invarianten formuliert, die bestimmte Konsistenzeigenschaften des Anbieterpaars garantieren.

Eine nichtsequentielle Implementierung der Spezifikation könnte die Konsistenz am einfachsten durch wechselseitigen Ausschluß der Klienten sicherstellen. Soll diese Implementierung aber so verteilt werden, daß Klienten auf verschiedenen Stationen liegen, so ist eine *verteilte* Ausschlußsynchronisation unumgänglich - ein Aufwand, den man lieber vermeiden würde!

Die Folgerung daraus ist, daß eine Spezifikation gemäß Abb. 5(a) für eine verteilte Implementierung nicht gut geeignet ist. Sie sollte so geändert werden, daß der Zugang zu den beiden Anbieter-Objekten durch *ein* Objekt reguliert wird, wie in Abb. 5(b) gezeigt.

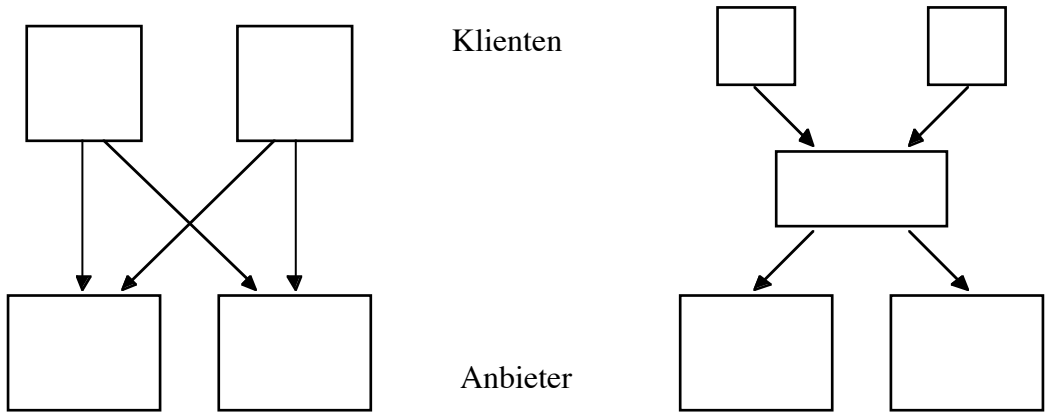

Abb. 5 Klienten mit konsistentem Anbieterpaar

6 Fazit

Die Entwicklung verteilter Applikationen wird beträchtlich vereinfacht, wenn verteilungstransparent *implementiert* werden kann. Man sollte deshalb nicht darauf verzichten, eine mindestens ebenso ausgeprägte Verteilungstransparenz bei der *Spezifikation* anzustreben. An einer Spezifikation aus dem Bereich der rechnergestützten Gruppenarbeit konnte man sehen, daß dies ohne weiteres möglich ist.

Es gibt allerdings Fälle, wo die spätere verteilte Ausführung nicht straflos ignoriert werden kann. Typische Entwurfsschwächen und die damit zusammenhängenden Probleme wurden erläutert. Natürlich überrascht es nicht, daß Entwurfsfehler in der Implementierungsphase oft nicht mehr kompensiert werden können - das gehört zum Grundwissen der Softwaretechnik. Das Fatale ist hier aber, daß sie bei der Implementierung noch nicht einmal bemerkt werden. Von der anschließenden Konfigurierung hängt es ab, ob der Entwurf "paßt"; somit tut man gut daran, trotz Verteilungstransparenz die beabsichtigte Konfiguration von Anfang an im Blick zu behalten.

Danksagung: Die Gutachter der GISI ´95 haben die Arbeit kenntnisreich kommentiert. Ich danke ihnen für eine Reihe treffender Verbesserungsvorschläge, die ich gerne aufgegriffen habe.

Literatur

[Duke u.a. 91] R. Duke, P. King, G. Rose, G. Smith: The Object-Z specification language: version 1. TR 91-1, Dept. of Computer Science, Univ. of Queensland, May 1991

[Duke u.a. 94] R. Duke, G. Rose, G. Smith: Object-Z: a specification language advocated for the description of standards. TR 94-45, Dept. of Computer Science, Univ. of Queensland, December 1994

[Kramer/Purtilo 94] J. Kramer, J. Purtilo (eds.): Proc. 2. Int. Workshop on Configurable Distributed Systems. IEEE 1994

[Lamport 78] L. Lamport: Time, clocks and the ordering of events in a distributed system. Comm. of the ACM 21.7, July 1978

[Levy/Tempero 91] H.M. Levy, E.D. Tempero: Modules, objects and distributed programming: issues in RPC and remote object invocation. Software - Practice & Experience 21.1, Januar 1991

[Löhr 93] K.-P. Löhr: Concurrency annotations for reusable software. Comm. of the ACM 36.9, September 1993

[Löhr 94] K.-P. Löhr: Towards an object-oriented design methodology for concurrent systems. Proc. TOOLS USA ´94, Prentice-Hall 1994

[Löhr u.a. 94] K.-P. Löhr, I.Piens, Th. Wolff: Verteilungstransparenz bei der objektorientierten Entwicklung verteilter Applikationen. OBJEKTspektrum 5/94, November/Dezember 1994

[OMG 91] Object Management Group: Common Object Request Broker: Architecture and Specification. Document 91.12.1, OMG 1991

[Raj u.a. 91] R.K. Raj, E.D. Tempero, H.M. Levy, A.P. Black, N.C. Hutchinson, E. Jul: Emerald: a general-purpose programming language. Software - Practice & Experience 21.1, Januar 1991

[Rose/Duke 94] G. Rose, R. Duke: An Object-Z specification of a mobile phone system. In K. Lano, H. Haughton (eds.): Object-Oriented Specification Case Studies. Prentice-Hall 1994

[Tanenbaum/Renesse 88] A.S. Tanenbaum, R.v. Renesse: A critique of the remote procedure call paradigm. In R. Speth: Research into Networks and Distributed Applications. North-Holland 1988

[Wolff 95] Th. Wolff: Transparently distributing objects with inheritance. Proc. 28. Hawaii Int. Conf. on System Sciences, IEEE 1995

[Wordsworth 92] J.B. Wordsworth: Software Development with Z. Addison-Wesley 1992