

Qualitätssicherung in objektorientierten Client/Server-Architekturen

Matthias Anlauff, Stefan Jähnichen*
Gesellschaft für Mathematik und Datenverarbeitung,
Forschungsinstitut für Rechnerarchitektur
und Softwaretechnik, Berlin
e-mail: {ma,jaehn}@first.gmd.de

3. Oktober 1996

Zusammenfassung

Für die Realisierung von Client/Server-Anwendungen bietet sich eine Vorgehensweise nach dem objektorientierten Paradigma an, da das Lokalisierungsprinzip von Objekten diese Art der Kommunikation optimal unterstützt. Die Grundlage einer objektorientierten Client/Server-Anwendung ist eine Schnittstellenbeschreibung, die die statischen Strukturen der angebotenen Dienstleistungen, in der Regel aber keine Informationen über deren dynamischen Verhalten enthält. Diese Tatsache verhindert effektive Qualitätssicherungsmaßnahmen auf der Seite der Dienstanwender. In diesem Papier wird ein Ansatz beschrieben, wie durch formale Spezifikation der Schnittstelle Qualitätssicherung bei objektorientierten Client/Server-Anwendungen ermöglicht wird.

1 Einleitung

Komplexe Systementwicklungen erfordern neue Konzepte zum Entwurf und zur Strukturierung von Software. Objektorientierung bietet entsprechende Unterstützung zur Zerlegung und Komposition von Systemen und hat wegen dieser Konzepte auch Akzeptanz in breiten Kreisen der Softwareindustrie gefunden.

Die Strukturierung nach diesem Paradigma hat auch spezielle Architekturformen hervorgebracht; eine davon ist bekannt als Client/Server-Architektur und erlaubt eine Zerlegung nach dem Prinzip des Anbieters eines Dienstes und dem Anwender. Trotz der offensichtlichen Vorteile systematischer Vorgehensweise nach objektorientierten Prinzipien, müssen zusätzliche Techniken angeboten oder entwickelt werden, die im Entwicklungszyklus eine systematische und kohärente Qualitätssicherung garantieren.

Dabei handelt es sich nicht notwendig nur um neue Techniken, sondern wesentlich um die Anpassung klassischer Vorgehensweisen an die Notationen und Dokumente, die der Entwicklung von Client/Server-Architekturen unterliegen.

*GMD FIRST, Rudower Chaussee 5, 12489 Berlin

Qualitätssicherung bei Client/Server-Architekturen kann auf unterschiedliche Weise durchgeführt werden. Die meisten Techniken, wie z.B. Testen, Simulation, Testfallgenerierung, sind Aktivitäten, die auf dem fertigen Programmcode durchgeführt werden. In diesem Papier wird ein spezieller Ansatz der Qualitätssicherung vorgestellt, der bereits die Entwicklung der Programmcodes miteinbezieht.

Die Grundlage einer objektorientierten Client/Server-Anwendung ist die Schnittstellenbeschreibung, in der die Klassenstruktur sowie die Attribute, Typdefinitionen und Methodensignaturen der Objektschnittstellen für den angebotenen Dienst angegeben sind. Eine Verhaltensbeschreibung der Routinen ist in der Regel nicht Teil der Beschreibungssprache oder erfolgt auf einer informellen Ebene. Das Fehlen einer präzisen Verhaltensbeschreibung der Dienste macht eine wirksame Qualitätssicherung insbesondere auf der Seite des Dienstanwenders praktisch unmöglich.

In diesem Papier wird ein Ansatz beschrieben, wie durch Verwendung der formalen Spezifikationssprache Object-Z [1, 3, 8, 4] eine solche Qualitätssicherung im Rahmen einer objektorientierten Client/Server-Entwicklung nach dem CORBA-Standard [6, 9] ermöglicht werden kann.

Im folgenden werden zunächst CORBA und Object-Z kurz vorgestellt. Danach wird beschrieben, wie Object-Z als Spezifikationssprache für Client/Server-Anwendungen verwendet wird. Anhand eines kleinen Beispiels wird abschließend der vorgestellte Ansatz veranschaulicht.

2 CORBA als Grundlage für Client/Server-Architekturen

CORBA (Common Object Request Broker Architecture) ist eine Technologie der 1989 gegründeten Object Management Group (OMG) und definiert eine Infrastruktur für die Erstellung von verteilten Anwendungen innerhalb heterogener Umgebungen. Die Standards der OMG ermöglichen die Kommunikation zwischen verschiedenen Softwarekomponenten auf Objektebene. Dabei ist es für den Dienstanwender unerheblich, auf welcher Plattform, welchem Betriebssystem und in welcher Programmiersprache die Zielobjekte auf dem Server implementiert sind.

Die technische Abwicklung der Kommunikation wird durch den im CORBA-Standard definierten Object Request Broker realisiert. Dessen primäre Aufgabe besteht in der Übermittlung von Methodenaufrufen von einer Client-Applikation zum Server.

Damit eine Applikation auf ein CORBA-Objekt zugreifen kann, muß die Interoperabilität zwischen dem Aufrufer und dem entfernten Objekt gewährleistet sein. In CORBA existiert zu diesem Zweck eine Schnittstellenbeschreibungssprache namens IDL (Interface Definition Language). IDL enthält ausschließlich Elemente zur Datenbeschreibung und keinerlei Anweisungskonstrukte.

Die prinzipielle syntaktische Struktur einer IDL-Schnittstellenbeschreibung ist wie folgt:

```

interface <identifier> [:<inheritance>] {
  <type declarations>;
  <constant declarations>;
  <attribute declarations>;
  <exception declarations>;

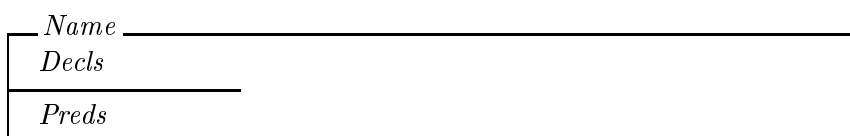
  [<op-type>] op_1 (<parameter>)
    [raises exception];
    .
    .
    .
  [<op-type>] op_n (<parameter>)
    [raises exception];
}

```

Ein IDL-Compiler kann aus einer IDL-Beschreibung zum einen Server-Skeletons, zum anderen Client-Stubs erzeugen. Erstere dienen als Vorgabe zur Implementierung des Servers, letztere kapseln auf der Seite der Client-Applikation die Methodenaufrufe eines entfernten Server-Objekts.

3 Object-Z

Object-Z ist eine objektorientierte Erweiterung der Spezifikationsprache Z [10], die auf einer mathematischen Mengennotation basiert. In Z wird ein Schemakalkül definiert, mit dem man zum einen Zustandsräume und zum anderen Zustandsübergangsope-
 rationen beschreiben kann. Grundkonstrukt in Z ist ein Schema das wie folgt aufgebaut ist:



Decls enthält Deklarationen von Variablen, über deren Eigenschaften in *Preds* prädikatenlogische Aussagen gemacht werden können. Durch die Verwendung von Schemaoperationen, wie z.B. der Schemainklusion, können in Z auch komplexere Sachverhalte übersichtlich dargestellt werden.

Object-Z erweitert Z um Konstrukte und Operationen, die eine objektorientierte Vorgehensweise beim Spezifizieren ermöglichen. Die wesentliche Erweiterung ist die Einführung von Klassenschemata, die wie folgt strukturiert sind (s. [1]):

<i>ClassName</i> [<i>generic parameters</i>] <i>inherited classes</i> <i>type definitions</i> <i>constant definitions</i> <i>state schema</i> <i>initital state schema</i> <i>operation schemas</i> <hr/> <i>history invariant</i>

Das *state schema* enthält die Attribute der Klasse, die darin enthaltenen Prädikate stellen die Klasseninvariante dar. Methoden der Klassen werden in den Operationschemata spezifiziert, wobei die ausgezeichnete Operation *Init* den Anfangszustand des Systems beschreibt.

4 Object-Z als Beschreibungssprache für CORBA

Die IDL-Beschreibung ist nach den Vorgaben des CORBA-Standards, das einzige Dokument, welches der Kommunikation zwischen Client und Server zugrundeliegt. Da IDL keinerlei Möglichkeiten besitzt, das dynamische Verhalten der Objekte zu beschreiben, ist der Dienstonutzer auf die mehr oder weniger gute Dokumentation der IDL-Beschreibung angewiesen. Dadurch entstehen zwangsläufig Probleme bei der Interpretation des Verhaltens von Methodenimplementierungen, da eine Dokumentation entweder nicht vorhanden ist, oder verschiedenen Interpretationsmöglichkeiten zulässt.

Diese Problem von IDL wird auch bei der Definition von TINA-ODL [2], einer Erweiterung von IDL, aufgegriffen. TINA-ODL unterstützt daher auch die Beschreibung des Verhaltens der Objekte. Allerdings geschieht dies hier nur auf einer informellen Ebene durch natürlichsprachliche Kommentare.

Die Idee des hier vorgestellten Ansatzes ist es, Object-Z für die Beschreibung der Schnittstellen zu verwenden und damit eine formale Verhaltensbeschreibung der Objekte zu ermöglichen.

Abbildung 1 skizziert diesen Ansatz. Aus einer Object-Z-Spezifikation wird zunächst eine IDL-Schnittstellenbeschreibung generiert (ozidl). Aus dieser werden mit Hilfe eines IDL-Compilers die Skeletons für die Klassen der Schnittstellenbeschreibung erzeugt. Aus der formalen Verhaltensbeschreibung der Objekte in der Object-Z-Spezifikation wird Qualitätssicherungscode erzeugt (ozqa), der zusammen mit den vom Benutzer erstellten Objektimplementierungen und den Skeletons die Server-Applikation bilden.

Durch diese Vorgehensweise ist es somit möglich, das Verhalten der Objekte formal zu *beschreiben*, und zusätzlich dieses Verhalten in Form des generierten Qualitätssicherungscode in der Server-Applikation *sicherzustellen*.

Im folgenden wird das prinzipielle Vorgehen bei der Generierung der Schnittstellenbeschreibung und des Qualitätssicherungscode erläutert.

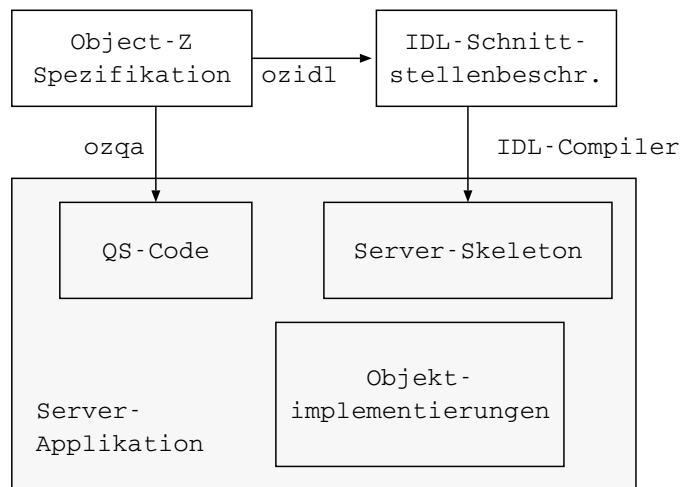


Abbildung 1: Verwendung von Objekt-Z für CORBA-Anwendungen

Zunächst wird ein IDL-Modul „Z“ definiert, das automatisch in alle generierten Schnittstellenbeschreibungen eingebunden wird:

```

module Z {
  interface Object {
    exception preViolation{};
    exception postViolation{};
  };
  interface Set {
    attribute sequence<any> elements;
    boolean isEmpty();
    boolean isIn(in any elem);
    Set setUnion(in Set s);
    Set setCut(in Set s);
  };
};
  
```

„Object“ dient als gemeinsame Oberklasse für die aus Object-Z-Klassen generierten Schnittstellen. In „Object“ werden die Ausnahmen definiert, die bei Verletzung der Vor- bzw. Nachbedingungen der zugrundeliegenden Object-Z-Operation ausgelöst werden. „Set“ wird als allgemeiner Datentyp für Mengenvariablen benötigt. Die Implementierung des Moduls „Z“ kann entweder von einer vorgefertigten Programmbibliothek übernommen oder von einem Server-Entwickler vorgenommen werden, um etwa die Methoden der „Set“-Schnittstelle zu optimieren.

Die Generierung von IDL-Schnittstellen aus Object-Z-Klassenbeschreibungen erfolgt auf der Grundlage von Regeln, die in „ozidl“ festgelegt sind. Die wichtigsten sind im folgenden aufgeführt:

- Eine Object-Z Klasse C wird in die IDL-Schnittstelle C übersetzt.
- Die Vererbungsstruktur der Object-Z-Klasse wird für die IDL-Schnittstelle über-

nommen. Zusätzlich erbt jede aus einer Object-Z-Klasse generierte IDL-Schnittstelle von der Klasse `Object` vom IDL-Modul `Z`.

- Die Typdefinitionen der Object-Z-Klasse werden in IDL-Typdefinitionen umgewandelt, z.B. einfache Datentypen in IDL-Aufzählungstypen und Schematypen in „structs“.
- Jede Deklaration im Zustandsschema einer Object-Z-Klasse wird in eine Attributdefinition in IDL umgewandelt.
- Für die Übersetzung von Operationsschemata in IDL-Methodensignaturen gelten die folgenden Regeln:
 - Es werden nur solche Operationen der Klasse in die IDL-Schnittstelle übernommen, die nicht in anderen Operationen der Klasse verwendet werden.
 - Existiert eine Operation `op` in einer Klasse `C` sowie eine Operation `op_EXCEPTION` in derselben Klasse, dann wird die Exception „`op_EXCEPTION`“ in die IDL-Schnittstelle übernommen und der „raises“-Liste der Methode `op` hinzugefügt.
 - Die Ausnahmen `preViolation` und `postViolation` werden in die „raises“-Liste jeder Operation aufgenommen.
 - Jede Eingabevariable des Operationsschemas wird als Eingabevariable der IDL-Methode gekennzeichnet (Schlüsselwort „in“); entsprechendes gilt für die Ausgabevariablen (Schlüsselwort „out“).

Der Qualitätssicherungscode wird in Form von Zusicherungen generiert (`ozqa`). Für jede Operation `op`, die als Methode in die IDL-Schnittstelle übernommen worden ist, werden zwei Überprüfungsroutrinen generiert:

`opCheckPre` überprüft die Gültigkeit der Vorbedingung der Methode unter Verwendung des „precondition“-Operators von `Z`. Dabei werden die aktuellen Parameter der Methode sowie die momentanen Werte der Zustandsvariablen der Klasse dahingehend überprüft, ob sie mit den im Prädikatenteil der Operation angegebenen Aussagen verträglich sind.

`opCheckPost` überprüft die Gültigkeit der Nachbedingung der Methode. Dabei werden die Werte der Ausgabevariablen sowie die am Ende der Methodenausführung vorliegenden Werte der Zustandsvariablen mit den im Prädikatenteil der Operation angegebenen Regeln verglichen. Ferner wird das Einhalten der Klasseninvarianten überprüft, die im Prädikatenteil des Zustandsschemas der Object-Z-Klasse spezifiziert sind.

Die Integration des Qualitätssicherungscodes in die Implementierung des Servers erfolgt in `ozqa` durch die Bereitstellung einer Pseudo-Objektimplementierung. Diese kapselt den Aufruf der eigentlichen, vom Server-Entwickler erstellten Objektimplementierung und erlaubt somit die Integration der o.g. Zusicherungen. Im Falle einer Verletzung der Vor- bzw. Nachbedingung wird eine entsprechende Ausnahme ausgelöst.

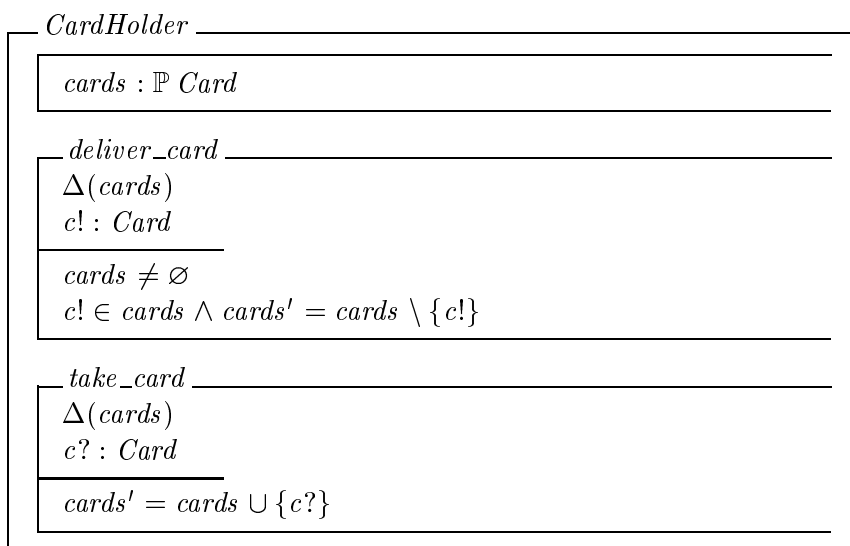
5 Fallstudie: Ein Parkhaus

Die Fallstudie behandelt eine einfache Steuerung der Ein- und Ausfahrten eines Parkhauses. Bei Einfahrt in das Parkhaus wird ein Parkschein ausgegeben, der bei Ausfahrt wieder zurückgenommen wird. Die Steuerung betrachtet die Ein- und Ausfahrten, sowie das Parkhaus selbst als Einheiten, die eine Menge von Parkscheine enthalten:

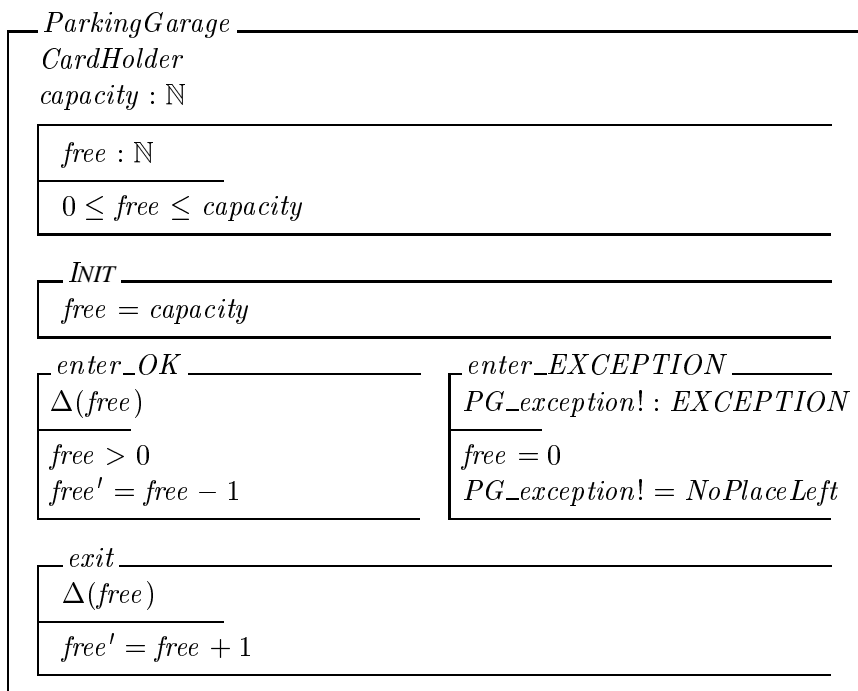
- die Parkscheine einer *Einfahrt* sind diejenigen, die bei Befahren des Parkhauses zur Verfügung stehen
- die Parkscheine einer *Ausfahrt* sind diejenigen, die bei Verlassen des Parkhauses eingezogen wurden und
- die Parkscheine des *Parkhauses* sind diejenigen, die sich im Besitz der Kunden befinden, deren Fahrzeug zur Zeit im Parkhaus abgestellt ist.

5.1 Spezifikation des Beispiels in Object-Z

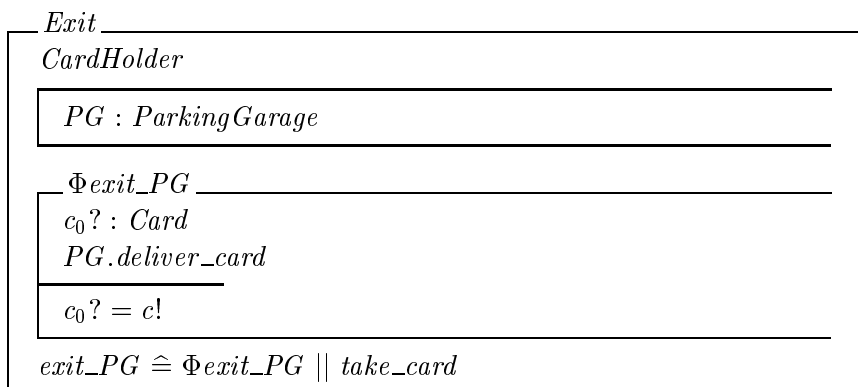
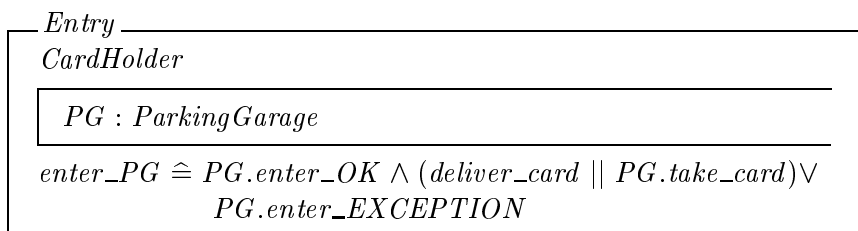
Die folgende Klasse *CardHolder* stellt die entsprechende Abstraktionsklasse dar. Die Operationen *deliver_card* bzw. *take_card* modellieren das Herausgeben eines (beliebigen) Parkscheins bzw. die Rücknahme eines (bestimmten) Parkscheins.



Die Klasse *ParkingGarage* ergänzt die obige Klasse durch die Betrachtung der Aufnahmekapazität des Parkhauses. Die Operationen *entry_OK* und *entry_EXCEPTION* werden von der Klasse *Entry* benutzt, um eine Einfahrt in das Parkhaus zu modellieren. Auf eine Ausnahmebehandlung im Falle der *exit*-Operation wurde hier verzichtet.



Die Klassen *Entry* und *Exit* enthalten als Zustandsvariable das Parkhaus, zu dem sie gehören. Durch Kombination der oben definierten Operationen erhält man die eigentlichen Ein- und Ausfahrtsoperationen (*enter_PG* bzw. *exit_PG*).



```

module ParkingGarage {
  interface Card : Z::Object {
  };
  interface CardHolder : Z::Object {
    attribute Z::Set cards;
    void deliver_card (out Card c) raises (preViolation,postViolation);
    void take_card (in Card c) raises (preViolation,postViolation);
  };
  interface ParkingGarage : CardHolder, Z::Object {
    attribute long capacity;
    attribute long free;
    exception enter_EXCEPTION{};
    void init();
    void enter() raises (enter_EXCEPTION,preViolation,postViolation);
  };
  interface Entry : CardHolder, Z::Object {
    attribute ParkingGarage PG;
    void enter_PG() raises (preViolation,postViolation);
  };
  interface Exit : CardHolder, Z::Object {
    attribute ParkingGarage PG;
    void exit_PG(in Card c) raises (preViolation,postViolation);
  };
};

```

Abbildung 2: IDL-Schnittstellenbeschreibung des Parkhauses

5.2 IDL-Schnittstellenbeschreibung

Aus dem in Object-Z spezifizierten Parkhaus-System werden gemäß den im vorigen Abschnitt angegebenen Regeln die in Abbildung 2 aufgeführten Schnittstellendefinitionen generiert.

5.3 Integration von Zusicherungscode in die generierten Server-Skeletons

Ausgehend von dieser Schnittstellenbeschreibung erzeugt der IDL-Compiler Server-Skeletons, auf deren Grundlage die Objektimplementierungen vorgenommen werden können. Wie bereits erwähnt, wird mit Hilfe von Pseudo-Objektimplementierungen die Integration des Qualitätssicherungscode erreicht. Die eigentliche Implementierung einer Methode `op`, die vom Entwickler vorgenommen wird, erfolgt in der Methode `opImpl`.

Als Beispiel betrachten wir die Methode `deliver_card` aus der Klasse `CardHolder`. Der verwendete IDL-Compiler hat als Zielsprache Java. Die Pseudo-Objektimplementierung von `deliver_card` ergibt sich dann wie folgt:

```

void deliver_card(ParkingGarage.CardHolder c)
    throws Z._Object.preViolation, Z._Object.postViolation {
    if (!deliver_cardCheckPre()) {
        throw new Z._Object.preViolation();
    }
    deliver_cardImpl(c);
    if (!deliver_cardCheckPost(c)) {
        throw new Z._Object.postViolation();
    }
}

```

Zur Überprüfung der Vorbedingung wird die Java-Methode `deliver_cardCheckPre` generiert:

```

boolean deliver_cardCheckPre() {
    return !(cards.isEmpty());
}

```

Die Methode `deliver_cardCheckPre` hat keine Parameter, da die Operation `deliver_card` keine Eingabeparameter besitzt. Entsprechend erhält die Methode `deliver_cardCheckPost` den Parameter `c`, der dem Ausgabeparameter `c!` von `deliver_card` entspricht.

6 Zusammenfassung

Der hier vorgestellte Ansatz ist ein Beispiel, wie man Qualitätssicherung bei Client/Server-Applikationen betreiben kann. Dabei wird nicht, wie bei den meisten anderen Qualitätssicherungsmaßnahmen, der fertige Programmcode zugrunde gelegt, sondern bei der Entwicklung des Programms Qualitätssicherungsaspekte berücksichtigt.

Die Grundlage einer objektorientierten Client/Server-Anwendung, die Schnittstellenbeschreibung, enthält im wesentlichen die Beschreibung der statischen Strukturen der angebotenen Dienste. Die hierfür verwendeten Sprachen (z.B. IDL von CORBA) sehen allerdings meist keine Möglichkeit vor, das Verhalten der implementierten Methoden präzise anzugeben. Das Fehlen dieser Verhaltensbeschreibung verhindert effektive Qualitätssicherungsmaßnahmen auf der Seite der Client-Applikation, weil dadurch die hierfür notwendigen Informationen fehlen.

Ausgehend von dieser Überlegung wurde in diesem Papier ein Ansatz vorgestellt, in dem man die Dienstschnittstelle in Object-Z formal spezifiziert und damit eine sehr hohe Genauigkeit der Verhaltensbeschreibung erreichen kann. Aus der formalen Beschreibung kann dadurch nicht nur eine IDL-Schnittstellenbeschreibung generiert, sondern die spezifizierten Prädikaten der Object-Z-Operationen können auch zum Erzeugen von Zusicherungscode verwendet werden. Dieser stellt bei Ausführung der Methode das Einhalten der Vor- und Nachbedingungen sicher.

Die formale Spezifikationsprache wird bei diesem Ansatz in einer Art und Weise verwendet, die nicht mit der einer rein formalen Softwareentwicklung zu vergleichen ist.

Die Spezifikation erfolgt sehr „implementationsnah“, d.h. die Strukturen der formalen Beschreibung werden 1:1 in Strukturen der Schnittstellenbeschreibung übernommen. Die formale Spezifikationsprache wird als Hilfsmittel zur präzisen Beschreibung der Datenstrukturen und des Verhaltens des implementierten Dienstes verwendet. Dadurch wird eine effektive Durchführung von Qualitätssicherungsmaßnahmen auf der Seite der Dienstanutzer ermöglicht.

Der in diesem Papier beschriebene Ansatz soll die üblichen QS-Verfahren, wie z.B. Testen, Simulation, Testfallgenerierung usw., nicht ersetzen, sondern diese durch das Miteinbeziehen der Entwicklungsphase effizient ergänzen.

Literatur

- [1] D. Carrington, D. J. Duke, R. Duke, P. King, G. A. Rose, and G. Smith. Object-Z: An object-oriented extension to Z. In S. Vuong, editor, *Formal Description Techniques, II (FORTE'89)*, pages 281–296. Elsevier Science Publishers (North-Holland), 1990.
- [2] Telecommunications Information Networking Architecture Consortium. TINA Object Definition Language (TINA-ODL) Manual, Version 1.3. Technical report, 1995.
- [3] R. Duke, P. King, G. A. Rose, and G. Smith. The Object-Z specification language. In T. Korson, V. Vaishnavi, and B. Meyer, editors, *Technology of Object-Oriented Languages and Systems: TOOLS 5*, pages 465–483. Prentice Hall, 1991.
- [4] R. Duke, P. King, G. A. Rose, and G. Smith. The Object-Z specification language: Version 1. Technical Report 91-1, Department of Computer Science, University of Queensland, St. Lucia 4072, Australia, April 1991.
- [5] V. Friesen. Formale objektorientierte Spezifikation hybrider Systeme. Dissertation, in Vorbereitung, 1996.
- [6] Object Management Group Inc. The Object Request Broker Architecture and Specification; Revision 2.0. Technical report, Framingham, MA., jul 1995.
- [7] B. Merkle. CORBA-Anbindung für Java. *iX*, 9:158–160, 1996.
- [8] G. Rose and R. Duke. An object-z specification of a mobile phone system. In K. C. Lano and H. P. Haughton, editors, *Object Oriented Specification Case Studies*, Object Oriented Series, pages 110–129. Prentice Hall International, 1993.
- [9] J. Siegel. *CORBA fundamentals and programming*. Wiley, 1996.
- [10] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, second edition, 1992.
- [11] M. Stal. CORBA und Java. *Java Spektrum*, 4:23–27, 1996.