JCICS Tutorial Teil 2
Externe Programmaufrufe und Datenaustausch unter JCICS

Dank an Stefan Huster für die Bereitstellung des Tutorials

1. Hintergrund

Die Hauptaufgabe der Softwareintegration ist es, die Kommunikation zwischen verschiedenen Systemen und Anwendungen zu ermöglichen. In diesem Tutorial werden Sie die CICS-internen Methoden kennenlernen, mit denen Sie eine solche Kommunikation aufbauen können. Im ersten Abschnitt werden Ihnen Befehle vorgestellt, mit denen Sie andere unter CICS installierte Programme aufrufen können. Im Anschluss erhalten Sie einen Einblick in Techniken für den Datenaustausch zwischen verschiedenen CICS-Programmen. Am Ende dieses Kapitels wird Ihnen noch der praktische Einsatz der vorgestellten Methoden und Verfahren in einem kurzen JCICS-Beispiel demonstriert.

Programme unter CICS können mit einer Vielzahl verschiedener Sprachen entwickelt werden, dazu zählen neben Java unter anderem auch Cobol, C++, oder PL/1. Die Systemschnittstelle von CICS ermöglicht es auf eine sehr einfache Weise, Kommunikationswege zwischen allen unter CICS installierten Programmen aufzubauen, unabhängig von der Sprache, mit der sie implementiert wurden. Für den Anwendungsentwickler stellt diese eine große Vereinfachung dar, da er ansonsten auf Verfahren wie zum Beispiel dem Remote Procedure Call (RPC), der Java Methode Invocation (JMI) oder der Common Object Request Broker Architecture (CORBA) zurückgreifen müsste. Das Problem dieser Techniken ist, dass sie jeweils nur von einer Teilmenge der verfügbaren Sprachen unterstützt werden.

2. Externe Programmaufrufe

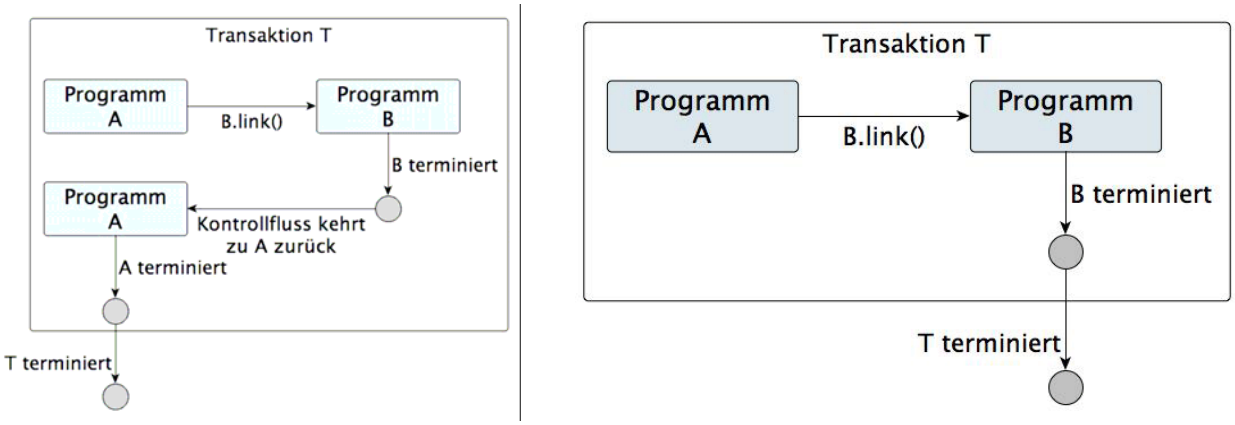
In diesem Abschnitt lernen Sie, wie Sie aus Ihrer CICS-Anwendung andere CICS-Programme aufrufen. Die Systemschnittstelle von CICS stellt Ihnen dafür zwei Befehle zur Verfügung: LINK und XCTL. Unter JCICS sind beide Befehle als Methoden der Klasse com.ibm.cics.server. Program aus der externen Bibliothek dfjcics.jar realisiert. Nach ihrer Ausführung werden die link- und die xctl-Anweisungen direkt unter CICS verarbeitet. Mit ihnen kann daher jedes CICS-Programm aufgerufen werden, unabhängig davon, in welcher Sprache es geschrieben wurde. An dieser Stelle werden Ihnen die konzeptionellen Unterschiede beider Methoden vorgestellt. Die genaue Syntax und die Verwendung unter JCICS werden Sie innerhalb des Abschnitts 3 kennenlernen.

2.1. LINK

Betrachten wir ein Szenario, bestehend aus einer Transaktion T, einem Programm A und einem Programm B. Innerhalb dieser Transaktion muss Programm A ein anderes Programm B aufrufen. Mit der abstrakten Anweisung B.link() kann das Programm A die Laufzeitumgebung auffordern, die momentane Programmausführung in der main-Methode von Programm B fortzusetzen. Terminiert Programm B, springt der Kontrollfluss zurück zu Programm A und die Programmausführung wird hinter dem Aufruf von B.link() in A fortgesetzt. Die gesamte Transaktion T terminiert demnach erst, wenn auch Programm A terminiert. Das Verfahren der externen Programmaufrufe kann natürlich transitiv fortgesetzt werden, indem auch Programm B weitere Programme aufruft.

2.2. XCTL

Die Aufgaben des xctl-Befehls sind identisch zu denen des link-Befehls. Auch die Syntax beider Anweisungen sind orthogonal zu verwenden. Der Unterschied beider Methoden liegt im Programmablauf. Betrachten wir hierfür noch einmal das Szenario aus Abschnitt 1.1. In diesem Beispiel hat Programm A ein anderes Programm B aufgerufen. Gehen wir an dieser Stelle davon aus, dass der Aufruf nicht via B.link(), sondern mit B.xctl() erfolgt ist. In diesem Fall kehrt der Kontrollfluss nach der Terminierung von B nicht mehr zu A zurück. Stattdessen terminiert die gesamte Transaktion T. Dies bedeutet, dass der xctl-Befehl die gesamte Ablaufkontrolle an das aufgerufene Programm überträgt. Den Unterschied zwischen LINK und XCTL finden Sie noch einmal in der Abbildung 1 illustriert.



(a) Der Kontrollfluss einer link-Anweisung

(b) Der Kontrollfluss einer xctl-Anweisung

Abbildung 1: Ablauf des Kontrollflusses bei LINK- und XCTL-Anweisungen

Selbst-Test

Können LINK und XCTL eingesetzt werden

- Nur zwischen 2 CICS Programmmen, die beide in Cobol geschrieben sind ?
- Nur zwischen 2 CICS Programmmen, die beide in Java geschrieben sind ?
- Auch zwischen 2 CICS Programmmen, die beide in untrschiedlichen Sprachen geschrieben sind ?

3 Datenaustausch

In diesem Abschnitt lernen Sie, wie Sie Daten zwischen verschiedenen CICS-Programmen austauschen können. Dieser Prozess spiegelt die eigentliche Kommunikation zwischen verschiedenen Softwarelösungen wieder. Diese herzustellen ist ein Ziel der Softwareintegration. Wie im vorangegangenen Abschnitt können die hier beschriebenen Verfahren dazu verwendet werden, Daten zwischen beliebigen CICS-Programmen auszutauschen, unabhängig davon, in welcher Sprache sie geschrieben wurden.

3.1. COMMAREA

Die COMMAREA ist ein speziell reservierter Speicherbereich. Jedes Programm, das z.B. über eine link-Anweisung aufgerufen wird, erhält eine Referenz auf eine übergebene COMMAREA. Auf diese Weise kann der speziell reservierte Speicher zum Austausch von Daten verwendet werden. Sowohl das aufrufende als auch das aufgerufene Programm verfügen über eine Referenz auf denselben Speicherbereich. Aus diesem Grund kann eine COMMAREA zur Ein- und zur Ausgabe verwendet werden. Die Größe des reservierten Speicherbereichs wird durch das aufrufende Programm festgelegt und kann später nicht mehr geändert werden. Ihre maximale Größe beträgt 32kB [1]. Die Übergabe der Daten erfolgt auf Byte-Ebene. Das heißt, auch unter JCICS werden Byte-Arrays ausgetauscht. Es liegt beim Anwendungsentwickler, diese in das richtige Format zu konvertieren und entsprechende Fehlerquellen zu berücksichtigen. Dieses Vorgehen ist sehr systemnah und auf einem sehr niedrigen Abstraktionslevel. Dennoch bildet diese Methode die Basis für jeden Kommunikationspfad innerhalb von CICS.

3.2. Channels und Container

Eine andere und auch etwas modernere Variante des Datenaustauschs zwischen verschiedenen CICS Programmen ist die Verwendung von „Channels und Container“. Ein Container dient der Kapselung der zu sendenden Daten. Man könnte ihn mit einem Umschlag vergleichen, der den Inhalt eines Briefes übermittelt, vergleichbar mit einem SOAP Envelope. Technisch gesehen ist ein Container nicht viel mehr als eine benannte COMMAREA. Ein Channel ist eine Verbindung zwischen zwei verschiedenen CICS-Programmen, in dem Container transportiert werden können. Der Kanal entspricht daher dem Briefträger, der einen Umschlag vom Sender zum Empfänger bringt. In Abschnitt 3 werden Sie die Verwendung von Container und Channels im praktischen Einsatz sehen.

Ein Channel bietet gegenüber der Verwendung der COMMAREA folgende Vorteile (Auszug aus [2]):

- **Channels sind nicht auf eine maximale Größe von 32kB beschränkt. Es besteht keine Limitierung bzgl. der Anzahl und Größe der übertragenen Container. Die einzige Beschränkung ist durch den physisch vorhandenen Speicher des Mainframes gegeben.**
- **Ein Channel kann Container verschiedener Typen enthalten. Dies ermöglicht, Daten strukturierter zu übertragen, vergleicht man das Vorgehen mit der COMMAREA, die nur einen einzigen Typ unterstützt.**

Channels sind ähnlch wie COMMAREA standardisiert und werden von allen unter CICS-lauffähigen Programmiersprachen unterstützt.

Auf der anderen Seite gilt es auch einige Punkte bei der Verwendung von Channels zu beachten:

- **Verwendet man Channels zur Datenübertragung zwischen CICS-Anwendungen, die auf unterschiedlichen Mainframes betrieben werden, kann der Transport größerer Datenmengen zu erheblichen Performanceverlusten führen.**
- **Für die Übertragung von Daten mit Hilfe von Channels werden mehr Systemressourcen benötigt als bei einer Übertragung via COMMAREA. Dies liegt daran, dass Container-Objekte in unterschiedlichen Speicherbereichen im System hinterlegt sein können. Dies erfordert unter Umständen häufiger langsame Lesezugriffe. Des Weiteren werden Daten innerhalb der COMMAREA nur als Referenz übergeben. Die Daten in einem Container werden kopiert.**

Um einen Container innerhalb eines Channels zu erzeugen, verwenden CICS-Programme den API-Befehl:

EXEC CICS PUT CONTAINER(Name des Containers) CHANNEL(Name des Channels)

Die Umsetzung innerhalb eines JCICS-Programms wird in Abschnitt 3 erläutert. Die Übertragung der Daten erfolgt mit Hilfe der oben beschriebenen link- oder xctl-Kommandos. Das Empfängerprogramm kann die Container über folgenden Befehl lesen:

EXEC CICS GET CONTAINER(Name des Containers)

Der Name des Channels wird automatisch impliziert, da jedes Programm zum Zeitpunkt des Aufrufs genau einen Channel zugewiesen bekommt.

Ein weiterer großer Vorteil von Channels ist die damit verbundene Erweiterungsmöglichkeit von Programmen. In Abbildung 2 sehen Sie ein Programm illustriert, welches über zwei verschiedene Channels aufgerufen werden kann. In der IBM-Literatur wird dieser Zustand als „loose coupling“ [3] bezeichnet. Damit ist gemeint, dass das Interface zum aufrufenden Klienten nicht eindeutig und fest definiert sein muss, sondern erst zur Laufzeit über die Auswahl des Channels bestimmt werden kann. Der Klient wählt einen entsprechenden Kanal des Hauptprogramms aus. Dieses interpretiert die Daten je nach Kanal anders und leitet sie unter Umständen an das entsprechende Unterprogramm weiter. In Abschnitt 3 werden Sie dieses Vorgehen in einem kleinen Beispiel im Einsatz sehen.

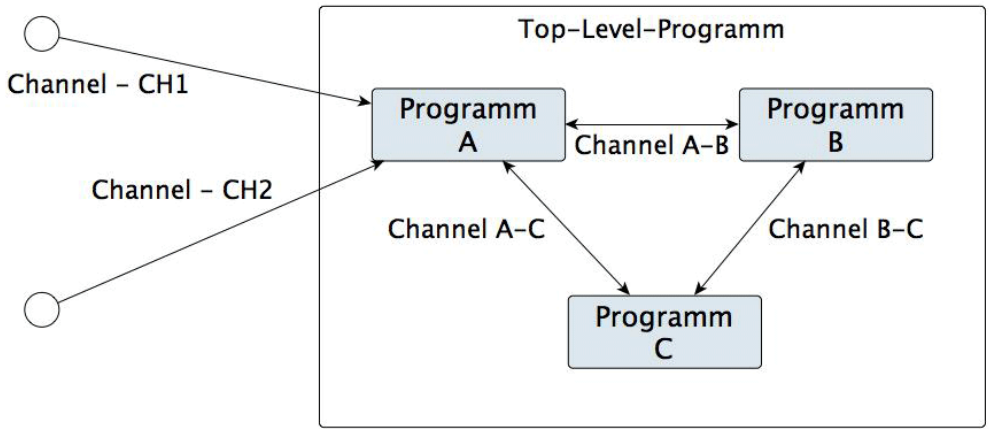


Abbildung 2: Channel Multiplexing

3.3. Channels vs. COMMAREA

Wie Sie gesehen haben, verfügen beide Verfahren über ihre Vor- und Nachteile. Die COMMAREA ist sehr leichtgewichtig und aufgrund des Datenaustauschs auf Basis von Referenzen sehr performant und zudem schnell zu implementieren. Auf der anderen Seite bieten Channels die Möglichkeit, Daten strukturiert zu übergeben und Programme sehr leicht zu erweitern. Dieser Komfort wird dafür mit einem erhöhten Verbrauch von Systemressourcen bezahlt.

Für die Entscheidung der verwendeten Technik stellt IBM in [1] drei einfache Kriterien bereit:

- **Ist es notwendig, größere Datenmengen jenseits der Grenze von 32kB zwischen Programmen zu übertragen, sind Channels und Container die eindeutig bessere Wahl.**
- **Arbeiten Sie mit bestehenden Programmen, die eine COMMAREA zur Kommunikation verwenden und nur geringe Datenmengen austauschen, lohnt sich die Umstellung auf das Channel-Konzept unter Umständen nicht.**
- **Entwickeln Sie neue Projekte, ist die Verwendung des Channel-Konzepts in der Regel die bessere Entscheidung.**

Selbst-Test

Welches der beiden Verfahren Channels vs. COMMAREA

- ist performanter ?
- gestattet die Übertragung beliebig großer Datenmengen ?
- ist einfacher zu programmieren
- hat einen größeren Funktionsumfang
- bewirkt Datenübertragung auf der Basis von Referenzen?

4. Anwendungsbeispiele

In den folgenden zwei Beispielen werden Sie sehen, wie die oben beschriebenen Techniken unter JCICS angewendet werden können.

4.1. LINK, XCTL, COMMAREA

In diesem Beispiel verwenden wir zwei Programme, von denen eins das andere einmal über die link- und einmal über die xctl-Anweisung aufruft. Zusätzlich wird dem aufgerufenen Programm über die COMMAREA eine Frage übermittelt. Das aufgerufene Programm wird dann die richtige Antwort in die COMMAREA schreiben.

Das Beispiel der link-Variante besteht aus zwei Programmen: „DoLink“ und „CallMe“. Deren Quelltexte finden Sie in den Listings 1 und 3. Wir beginnen mit „DoLink“:

Zu Beginn geben wir zur Bestätigung in (1) den Programmnamen (hier „DoLink“) des aufgerufenen Programms aus. Dies zeigt uns, dass der Kontrollfluss zurzeit von der Klasse DoLink gesteuert wird.

Als nächstes erzeugen wir in Schritt (2) eine Nachricht, die wir dem aufrufenden Programm senden möchten. Da die COMMAREA unter JCICS als Byte-Array repräsentiert wird, muss der String vor dem Senden konvertiert werden. Die Größe der COMMAREA ist damit automatisch festgelegt worden. Sie entspricht der Größe der gespeicherten Zeichenkette.

Unter Punkt (3) erzeugen wir eine neue Instanz der Program-Klasse. Dieser Typ wird von der JCICS API bereitgestellt und dient der Abstraktion über CICS-Programme. Dem erzeugten Programm weisen wir im Anschluss einen Namen zu. Dieser darf nicht willkürlich vergeben werden.

Er muss mit dem Namen des aufzurufenden Programms übereinstimmen, der innerhalb der entsprechenden CICS-Ressource definiert wurde. Die Übereinstimmung des Klassennamens mit dem Programmnamen ist hingegen nicht vorgeschrieben. Mit dem Befehl prog.link(COMMAREA) wird schließlich das zweite Programm des Beispiels aufgerufen. Dessen Analyse folgt im Anschluss an diese Besprechung.

Das von uns aufgerufene Programm wird, wie wir später sehen werden, eine Antwort in die übergebene COMMAREA schreiben. Diese wird in Schritt (4) ausgegeben.

Sowohl der link-Befehl als auch die Abfrage des Programmnamens unter Punkt (1) sind potentielle Fehlerquellen. Es könnte zum Beispiel passieren, dass das aufgerufene Programm nicht existiert. In diesem Fall würde die link-Anweisung eine Exception werfen. Diese werden bei Punkt (5) gefangen. Genauere Informationen über die möglichen Exceptions finden Sie in [2] und [4].

Als Nächstes widmen wir uns der Betrachtung des Quelltextes des aufgerufenen Programms. In diesem Beispiel heißt dieses CALLME. Die entsprechende Java-Klasse trägt den Namen CallMe und ist in Listing 3 abgedruckt.

Wie auch zuvor beginnen wir das Programm mit einem kurzen Lebenszeichen (1). Den Sinn und Zweck der println-Anweisungen besprechen wir später, wenn wir die Ausgabe betrachten und diese mit der des xctl-Beispiels vergleichen.

Die übergebene COMMAREA wird durch den COMMAREAHolder als Parameter der main-Methode gekapselt. Dieser hat genau ein Klassenfeld namens value vom Type byte[]. Diesem Feld weisen wir in Schritt (2) die richtige Antwort auf die übertragene Frage zu.

Nachdem der Quelltext beider beteiligten Klassen besprochen wurde, wird es Zeit, einen Blick auf die Ausgabe der entsprechenden Transaktion zu werfen. Diese sehen Sie in Abbildung 3. Bereits bei der oberflächlichen Betrachtung sind zwei Punkte besonders auffällig:

1. **Es fehlt bei der Ausgabe der Transaktion in Zeile 2, 3 und 4 jeweils der Anfangsbuchstabe.**
2. **Neben der richtigen Antwort 42 [5] ist in Zeile 5 noch der Rest der Anfrage sichtbar.**

Die Erklärung der ersten Beobachtung führt uns direkt zurück zu der Frage, wieso in der Klasse CallMe so viele println-Anweisungen verwendet wurden. Die println-Anweisungen beginnen ihre Ausgabe in der ersten Zeile des Terminalfensters. In dieser Zeile steht i.d.R. jedoch der Name der Transaktion, hier XDOL. Aus diesem Grund begannen wir in der Klasse DoLink auch die Ausgabe mit einer leeren Zeile. Übergibt man den Kontrollfluss via LINK oder XCTL an ein anderes Programm, welches ebenfalls Ausgaben auf das Terminalfenster druckt, beginnt die Ausgabe wieder in der ersten Zeile. Dies liegt daran, dass dem aufgerufenen Programm eine neue Task-Umgebung zugewiesen wird. Aus diesem Grund gibt die Klasse CallMe zuerst drei leere Zeilen aus, da ansonsten die Ausgabe des DOLINK-Programms überschrieben werden würde. Die Ausgabe von println() ist jedoch nicht leer, sondern wird im Terminal als „-„ dargestellt, um jeweils das Ende einer Zeile zu markieren. Das “-„ überschreibt auch den ersten Buchstaben der Zeilen 2-4.

Der Grund für die zweite Beobachtung wurde bereits im Abschnitt 2.1 besprochen. Dort wurde festgestellt, dass die COMMAREA immer eine feste Länge hat, die vom aufrufenden Programm festgelegt wird. In diesem Beispiel entspricht die Größe der COMMAREA genau dem Platz, der durch die Anfrage “the answer to life the universe and everything” benötigt wird. Die Antwort „42" hingegen ist kürzer und füllt daher nicht die gesamte COMMAREA aus. Der nicht überschriebene Speicher bleibt unverändert. Dies führt dazu, dass ein Teil der Anfrage auch noch in der Antwort zu lesen ist. Es ist die Aufgabe des Anwendungsentwicklers, dafür zu sorgen, dass gegebene Bedingungen an den Speicherbereich der COMMAREA erfüllt werden. Eine mögliche Bedingung für dieses Beispiel wäre gewesen, dass die Antwort keinen Teil der Anfrage mehr enthalten darf. In diesem Fall hätte der nicht überschriebene Teil der COMMAREA manuell gelöscht werden müssen.

Bevor wir unsere Aufmerksamkeit auf die Ausgabe der xctl-Variante lenken, betrachten wir noch kurz den Quellcode dieser Alternative. Die entsprechende Klasse heißt DoXctl (Listing 1) und das CICS-Programm DOXCTL. Der Quelltext der Java-Klasse ist an dieser Stelle mehr der Vollständigkeit zuliebe aufgeführt, da die Änderungen verglichen zu der link-Varainte nur minimal ausfallen. Der einzige Unterschied ist unter Punkt (3) zu finden. Dort wird einfach anstelle von prog.link(COMMAREA) prog.xctl(COMMAREA) verwendet.

Die Auswirkung der xctl-Anweisung kann direkt in der Ausgabe in Abbildung 3b verfolgt werden. Vergleicht man diese mit der in Abbildung 3a gezeigter Ausgabe der DOLINK Transaktion, fällt auf, dass DOXCTL keine vierte Zeile ausgibt. Dies liegt daran, dass die entsprechende Transaktion DOXCTL nach der Ausführung von CALLME terminiert und nicht wie zuvor zum aufrufenden Programm zurückkehrt. Das heißt, die Zeilen unter Punkt (4) in der DoXctl-Klasse werden aufgrund der xctl-Anweisung nie erreicht und ausgeführt.

Ein weiterer kleiner Unterschied zu der Ausgabe von DOLINK ist das Vorhandensein der ersten Zeichen in Zeile 2 und 3. Dies hat jedoch nichts mit der xctl-Methode zu tun, sondern ist das Resultat eines vorangestellten Leerzeichens jeder println-Anweisung in DoXctl.

4.2. Channels und Container

Im zweiten Beispiel betrachten wir die Kommunikation zwischen verschiedenen CICS-Programmen unter Verwendung von Channels und Container. Des Weiteren wird Ihnen gezeigt, wie Sie über einen Channel-Multiplexer mit einem Programm verschiedene Klienten bedienen können.

```
XLNK
    his is program:TUTLINK
    y question:the answer to life the universe and
everything
    his is program CALLME
And the answer is: 42e answer to life the universe
and everything
```

(a) Ausgabe der Transaktion XDOL

```
XCTL
    This is program:TUTXCTL
    My question:the answer to life the universe and
everything
This is program CALLME
```

(b) Ausgabe der Transaktion XDOL

Abbildung 3: Ausgabe der externen Programmaufrufe

Diese kleine Anwendung besteht aus drei verschiedenen Klassen: NeedEuro, NeedDollar und MoneyConverter. Die ersten beiden Klassen erfüllen die Rolle der Klienten, während die dritte Klasse die Rolle eines Serviceproviders einnimmt. Die Idee ist, dass die Klienten jeweils entweder einen Euro- oder einen Dollar-Betrag an die Serviceklasse schicken und als Antwort den entsprechenden Betrag in der jeweils anderen Währung erhalten.

Damit der Serviceprovider beide Anfragen unterscheiden kann, verwenden die Klienten Channels mit unterschiedlichen Namen. Natürlich wären an dieser Stelle auch andere Möglichkeiten zur Unterscheidung denkbar. Streng genommen wären diese sogar allein durch unterschiedlich benannte Container möglich, jedoch würde eine solche Optimierung dieses Beispiel zunichtemachen.

Da beide Klassen bis auf ihren Namen und den Namen des verwendeten Channels identisch aufgebaut sind, begnügen wir uns hier mit der Betrachtung nur eines Quellcodes, dem der NeedEuro-Klasse in Listing 4. Den Quelltext der anderen Klasse finden Sie im Anhang.

In Schritt (1) erzeugen wir einen neuen Kanal. Dies erfolgt über eine Factory-Methode des Task- Objekts (vgl. Factory-Pattern [6]) mit dem Namen createChannel. Wir weisen dem Kanal den Namen „DollarToEuro“ zu.

Als nächstes benötigen wir einen Container, den wir über den erzeugten Kanal versenden können.

Auch dieser wird in Schritt (2) über eine Factory-Methode erstellt. Der Container erhält den Namen „DOLLAR". Container sind vom Prinzip her nichts anderes als benannte COMMAREAs. Aus diesem Grund können sie ebenfalls nur Daten als Byte-Array speichern. Dafür erstellen wir in Punkt (3) einen String mit dem zu konvertierenden Betrag als Wert. Im Anschluss wird der String über die put-Methode im Container gespeichert. Die Konvertierung in eine Byte-Array erfolgt durch die Methode automatisch.

Schritt (4) sollte Ihnen aus Abschnitt 3.1 vertraut vorkommen. Wir erzeugen eine Instanz des rogram-Objekts und rufen über die link-Methode das externe CICS-Programm MONCONV auf.

Dieses Mal übergeben wir der link-Anweisung jedoch keine COMMAREA, sondern den erzeugten Kanal (5).

Der Wert des Containers wird, ähnlich wie im ersten Beispiel, durch das aufgerufene Programm geändert. Der geänderte Wert wird in Schritt (6) mit der get-Methode neu ausgelesen und im Anschluss ausgegeben.

Betrachten wir als Nächstes, wie der Dollarwert durch den Serviceprovider in Euro umgerechnet wird. Den Quelltext der Klasse MoneyConverter finden Sie in Listing 6.

Zu allererst wird der Channel benötigt, mit dem dieses Programm aufgerufen wurde. Diesen erhalten wir von der Laufzeitumgebung (1), über die unter CICS durch das Task-Objekt abstrahiert wird.

Der Channel, den die Task zurückgegeben hat, kann unter Umständen auch null sein, zum Beispiel dann, wenn das Programm mit einer übergebenen COMMAREA aufgerufen wurde. Diese Möglichkeit muss überprüft werden (2), bevor mit jeglicher Verarbeitung des Channels begonnen werden kann. Im Block (3), bestehend aus Block (3-a) und (3-b), findet das Multiplexen des Channels statt. Dafür wird zuerst der Name des übergebenen Channels erfragt und anschließend mit allen unterstützten Namen verglichen. Diese Serviceklasse unterstützt zwei verschiedene Channels: „EuroToDollar“ und „DollarToEuro“. Die Verarbeitung des Channels erfolgt im Anschluss innerhalb von Block (3-a) bzw. (3-b). Dort wird der erwartete Container mit ch.getContainer() aus dem Channel extrahiert und dessen Wert in das gewünschte Format konvertiert. In diesem Fall erwarten wir einen Betrag, den es in eine andere Währung umzurechnen gilt. Der Wert ist daher numerisch und kann mit Double.valueOf() konvertiert werden.

Für die Umrechnung in die entsprechende Währung sind die Methoden unter (4-a) und (4-b) verantwortlich. Deren Ergebnis wird mit der put-Methode des Containers zurückgeschrieben und kann im Anschluss vom aufrufenden Programm gelesen werden.

Die Ausgabe beider Programme sehen Sie in Abbildung 4.

XEUR Give100 Dollars Get76.92307692307692 Euros	XUSD Give100 Euros Get130.0 Dollars
(a) Ausgabe Konvertierung Euro nach Dollar	**(b) Ausgabe Konvertierung Dollar nach Euro**

Abbildung 4: Ausgabe der beiden Klienten des Währungsrechners

Hinweis

Die Methoden LINK und XCTL können auch mit anderen Parametern aufgerufen werden, als die, die in diesem Beispiel verwendet wurden. Eine detaillierte Liste finden Sie in der IBM JCICS API [7].

Channels können auch mehr als einen Container enthalten. Wie im Abschnitt 2.1 erwähnt wurde, ist deren Anzahl nur durch die Größe des physischen Speichers beschränkt. In Fällen, in denen mehrere Container übergeben werden, kann es sehr sinnvoll sein, über diese als Liste zu iterieren. Für diesen Zweck stellt die JCICS API einen ContainerIterator bereit. Dieser implementiert das bekannte java.util.Iterator Interface. Listing 7 zeigt ein Beispiel, entnommen aus [2], für die Verwendung dieses Iterators.

5. Zusammenfassung

Die CICS API stellt zwei Methoden für den Aufruf externer CICS-Programme bereit: LINK und XCTL. Beide Methoden verfügen über eine ähnliche Schnittstelle. Die link-Methode ruft ein anderes Programm auf und kehrt nach dessen Terminierung wieder zum aufrufenden Programm zurück. Die xctl-Anweisung hingegen beendet die gesamte Transaktion, nachdem das aufgerufene Programm terminiert ist.

Für den Austausch von Daten zwischen CICS-Programmen existieren ebenfalls zwei verschiedene Verfahren: Datenaustausch über die COMMAREA und über Channels.
Die Commera ist ein spezieller, geteilter Speicherbereich zweier CICS-Programme. Die Daten werden als Referenz übergeben und müssen vom Type byte[] sein. Die Größe einer COMMAREA wird einmal festgelegt und kann maximal 32kB betragen.

Ein Channel überträgt Daten, die zuvor mit einem Container gekapselt wurden. Channel und Container können benannt und somit auch unterschieden werden. Auch hier werden Daten als Bytes transportiert. CICS gibt jedoch keine Obergrenze für die Anzahl der Container und deren Größe vor. Sowohl die Methoden zum Aufruf externer Programme als auch die Verfahren zur Datenübertragung funktionieren zwischen allen unter CICS installierten Programmen.

Selbst-Test

- Erklären Sie, warum im Beispiel 4 die link-Methode verwendet wurde und nicht die xctl Anweisung ?

<u>Aufgabe:</u> Entwickeln Sie einen JCICS-Taschenrechner. Dieser soll die vier Grundrechenarten unterstützen. Ein zweites CICS-Programm soll die verschiedenen Methoden des Taschenrechners aufrufen und das berechnete Ergebnis auf der Konsole ausgeben. Entscheiden Sie selbst, welches Verfahren Sie für den Programmaufruf und für die Datenübertragung verwenden

Quellcodes

```
1 public class DoLink {
2  public static void main (CommAreaHolder cah) {
3       Task task = Task.getTask ();
4       PrintWriter out = task.out;
6       try {
7        // ## (1)
8        out.println();
9        out.println("This is program :" + task.getProgramName());
11            // ## (2)
12            String myMsg = "the answer to life the universe and
everything";
13            byte[] commArea = myMsg.getBytes();
14            out.println("My question :" + myMsg);
16            // ## (3)
17            Program prog = new Program();
18            prog.setName("CALLME");
19            prog.link(commArea);
21            // ## (4)
22            out.println();
23            String rply = new String(commArea);
24            out.println("And the answer is: " + rply);
26            // ## (5)
27           } catch (Exception e) {
28             e. printStackTrace(out);
29           }
30      }
31 }
```

Listing 1: Aufrufendes Programm mit LINK

```
1 public clas s DoXctl {
2  public static void main(CommAreaHolder cah) {
3       Task task = Task.getTask();
4       PrintWriter out = task.out;

6       try {
7        // ## (1)
8        out.println();
9        out.println(" This is program :" + task.getProgramName ());

11            // ## (2)
12            String myMsg = "the answer to life the universe and
everything";
13            byte[] commArea = myMsg.getBytes();
14            out.println("My question :" + myMsg);

16            // ## (3)
17            Program prog = new Program();
18            prog.setName("CALLME");
19            prog.xctl(commArea);

21            // ## (4)
22            out.println();
23            String rply = new String(commArea);
24            out.println("And the answer is: " + rply);

26            // ## (5)
27           } catch (Exception e) {
28            e.printStackTrace(out);
29           }
30      }
31 }
```

Listing 2: Aufrufendes Programm mit XCTL

```
1 public class CallMe {
2  public static void main (CommAreaHolder cah) {
3       PrintWriter out = Task.getTask().out;
4       out.println();
5       out.println();
6       out.println();
7       // ## (1)
8       out.println("This is program CALLME");
9       // ## (2)
10           String theAnswer = "42";
11           cah.value = theAnswer.getBytes();
12      }
13 }
```

Listing 3: Aufrufendes Programm mit XCTL

```
1 public class NeedEuro {
2  public static void main (CommAreaHolder cah) {
3       try {
4        // ## (1)
5        Channel ch = Task.getTask().createChannel("DollarToEuro");
6        // ## (2)
7        Container dollarCon = ch.createContainer("DOLLAR");
8        // ## (3)
9        String dollarStr = "100";
10            dollarCon.put(dollarStr);
11            // ## (4)
12            Program moneyConverter = new Program ();
13            moneyConverter.setName("MONCONV");
14            // ## (5)
15            moneyConverter.link(ch);
17            PrintWriter out = Task.getTask().out;
18            out.println();
19            out.println("Give " + dollarStr + " Dollars");
20            // ## (6)
21            String euroStr = new String(dollarCon.get());
22            out.println("Get " + euroStr + " Euros");
24           } catch ( Exception e) {
25            e. printStackTrace(out);
26           }
27      }
28 }
```

Listing 4: Anfrage mit Channel

```
1 public class NeedDollar {
2  public static void main ( CommAreaHolder cah ) {
3       try {
4        // ## (1)
5        Channel ch = Task.getTask().createChannel("EuroToDollar");
6        // ## (2)
7        Container euroCon = ch.createContainer("EURO");
8        // ## (3)
9        String euroStr = "100";
10            euroCon.put(euroStr);
11            // ## (4)
12            Program moneyConverter = new Program ();
13            moneyConverter.setName("MONCONV");
14            // ## (5)
15            moneyConverter.link(ch);
17            PrintWriter out = Task.getTask().out;
18            out.println();
19            out.println("Give " + euroStr + " Euros");
20            // ## (6)
21            String dollarStr = new String(euroCon.get());
22            out.println("Get " + dollarStr + " Dollars");
24           } catch (Exception e) {
25            e. printStackTrace(out);
26           }
27      }
28 }
```

Listing 5: Anfrage mit Channel 2

```
1 public class MoneyConverter {
2  public static void main(CommAreaHolder cah) {
4       PrintWriter out = Task.getTask().out;
6       // ## (1)
7       Channel ch = Task.getTask().getCurrentChannel();
9       // ## (2)
10           if ( ch != null ) {
11            String chName = ch.getName().trim();
13            try {
15             // ## (3)
16             double resValue ;
17             if ( chName.equals("EuroToDollar")) {
18              // ## (3-a)
19              Container euroRec ;
20              euroRec = ch. getContainer("EURO");
21              double reqVal = Double.valueOf(new
String(euroRec.get()));
22              resValue = euroToDollar(reqVal);
24              euroRec.put(Double.toString(resValue));
26             } else if (chName.equals("DollarToEuro")) {
27              // ## (3-b)
28              Container dollarRec = ch. getContainer("DOLLAR");
29              double reqVal = Double.valueOf(new
String(dollarRec.get()));
30              resValue = dollarToEuro(reqVal);
31              dollarRec.put(Double.toString(resValue));
32             }
33            } catch (Exception e) {
34             e.printStackTrace(out);
35            }
36           }
37      }
38      // ## (4-a)
39      private static double euroToDollar(double euro) {
40       return (euro * 1.30);
41      }
42      // ## (4-b)
43      private static double dollarToEuro(double euro) {
44       return (euro / 1.30);
45      }
47 }
```

Listing 6: Java Channel Multiplexer

```
1 Task t = Task.getTask();
2 ContainerIterator ci = t.containerIterator();
3 while (ci.hasNext()) {
4   Container custData = ci.next();
5   // Verarbeitung Container Inhalt
6 }
```

Listing 7: Beispiel für einen ContainerIterator

6. Literatur

[1] Rayns, C., Clee, G.B., Grieve, T., Taylor, J., Ge, Y.P., Li, G.Q., Zhang, Q., Wen, D.: Java Application Development for CICS. 4 edn. IBM (2009)

[2] IBM: CICS Transaction Server for z/OS Release Guide. (3.1 edn.)

[3] IBM: Java Applications in CICS. IBM. 3.2 edn. (2008)

[4] IBM: (http://publib.boulder.ibm.com/infocenter/cicsts/v3r1/index.jsp?topic=/com.ibm.cics.ts31.doc/dfhpk/com/ibm/cics/server/package-summary.html)

[5] Adams, D.: The Hitchhiker's guide to the galaxy. delrey (1979)

[6] Gamma, E., Helm, R., Johnson, R.E., Vlissides, J.: Design Patterns. Elements of Reusable Object-Oriented Software. 1 edn. Addison-Wesley (1994)

[7] Oracle: http://download.oracle.com/javaee/1.4/tutorial/doc/index.html (2010)