13. Beispiele verteilter CORBA-basierter Applikationen

Die in Kapitel 11 und 12 vorgestellten Erweiterungsmodule wurden in [Sch02] prototypisch in einem gemeinsamen Erweiterungsmodul implementiert. Um die verschiedenen Aspekte des Systems zu testen und um das allgemeine Konzept zu prüfen, wurden drei Applikationen entwickelt: Ein einfaches System für eine Datenbankabfrage, eine Applikation zur Vermittlung zwischen einem Puffer und einem Drucker und eine komplexe Raumverwaltungsanwendung, die fortgeschrittene Eigenschaften von CORBA einsetzt.

13.1. Datenbankabfrage

Eine Standardarchitektur für Verteilte Informations- und Kontrollsysteme ist eine 3-Schichtenarchitektur. Dabei verwaltet die unterste Schicht die persistenten Daten in einer Datenbank. Die mittlere Schicht enthält die Applikationslogik, während die oberste Schicht die Nutzerschnittstellen enthält. Der Zugriff auf die Funktionalität der einzelnen Schichten wird über CORBA-Objekte realisiert.

Abbildung 13.1 zeigt eine Applikation, die nach diesem Ansatz implementiert wurde. Sie ermöglicht beliebig vielen Nutzen nebenläufig, Anfragen an eine Datenbank zu stellen. Jeder Nutzer erzeugt sich hierfür lokal ein Exemplar der `GUI` Komponente und erhält eine Dialogbox. Damit beliebig viele Exemplare möglich sind, ist in `GUI` das Attribut `Multiplicity` auf `*` gesetzt.

Der Code für die grafische Nutzerschnittstelle enthält keine Applikationslogik und damit auch keine Verwaltung eines Sitzungszustands. Dieses ist Aufgabe der `Logic` Komponente. Für jedes Laufzeitexemplar von `GUI` muss daher genau ein Laufzeitexemplar von `Logic` existieren. Dies wird ausgedrückt, indem das Attribut `cardinality` der Rollen des Assoziationskonnektors, der `GUI` und `Logic` verbindet, beide auf `1` gesetzt sind.

Die `Logic` Komponente stellt die notwendigen Anfragen an die Datenbank `DB`. Diese ist ein externes Objekt, das in die Applikation integriert wird. Die Attribute sind dabei so gesetzt, dass höchstens ein Laufzeitexemplar für `DB` vorhanden sein kann. Zusätzlich gibt es noch eine Komponente `Printer`, die Druckdienste anbietet. Sie wird zur Protokollierung der Datenbankzugriffe benutzt.

Diese Applikation demonstriert, dass es mit diesem Ansatz möglich ist, eine verteilte Applikation mit einer einfachen Struktur mit wenig Aufwand zu realisieren, ohne dabei die Struktur aus den Augen zu verlieren. Auch wenn

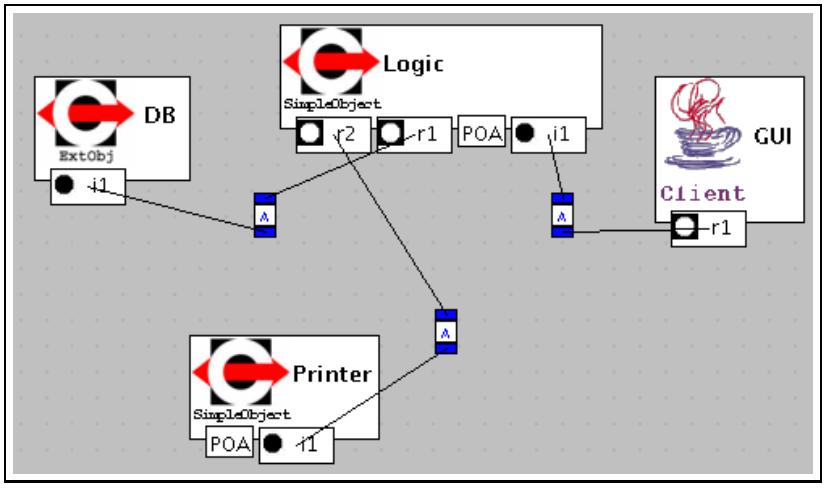

Abbildung 13.1.: Eine einfache Datenbank-basierte Applikation

das Setzen der Attribute auf den ersten Blick komplex erscheint, so ist es doch deutlich weniger umfangreich als das manuelle Aufsetzen einer CORBA-basierten Applikation. Hinzukommend orientieren sich die Attribute an statischen UML-Klassendiagrammen und sind damit i.a. direkt aus einem detaillierten Entwurf zu entnehmen.

13.2. Drucksystem

Während die vorherige Applikation den Einsatz des Entwicklungssystems für verteilte Informations- und Kontrollsysteme demonstriert, soll diese Applikation den Einsatz für arbeitsablaufbasierte Applikationen zeigen. Abbildung 13.2 zeigt eine Anwendung, die zwischen einem Druckerpuffer und einem Drucker vermittelt und so ein Drucksystem realisiert. Sie stellt damit einen Adapter zwischen dem Puffer- und dem Druckerobjekt dar.

Nach dem Starten der Anwendung wird zuerst die Komponente `GetJob` ausgeführt. Diese ruft auf dem von der Komponente `PrinterQueue` repräsentierten CORBA-Objekt die Methode `getNextJob()` auf, die in dem Attribut `Method` gesetzt ist. Das Ergebnis wird der `PrintJob` Komponente übergeben, die diese an die `Printer` Komponente über dessen `print()` weitergibt.

Der Rückgabewert der Methode `print()` wird an den Druckerpuffer zurückgegeben. Ist dieser `true` und gab es damit keine Probleme, wird der vorderste Druckauftrag gelöscht. Ansonsten wird er behalten und im nächsten Durchgang nochmals ausgedruckt. Diese Logik wäre mit einem entsprechenden `Switch`-Konnektor einfacher zu implementieren. Da in Kapitel 9.4.1 spezialisierte Verzweigungskonnektoren untersucht wurden, wurde hier auf eine Implementierung verzichtet.

Nach dem eventuellen Löschen des Druckauftrags wird zurückgesprungen zu der ersten Operation. Kommt es bei einer Operation zu einer Ausnahme, bricht die Applikation ab.

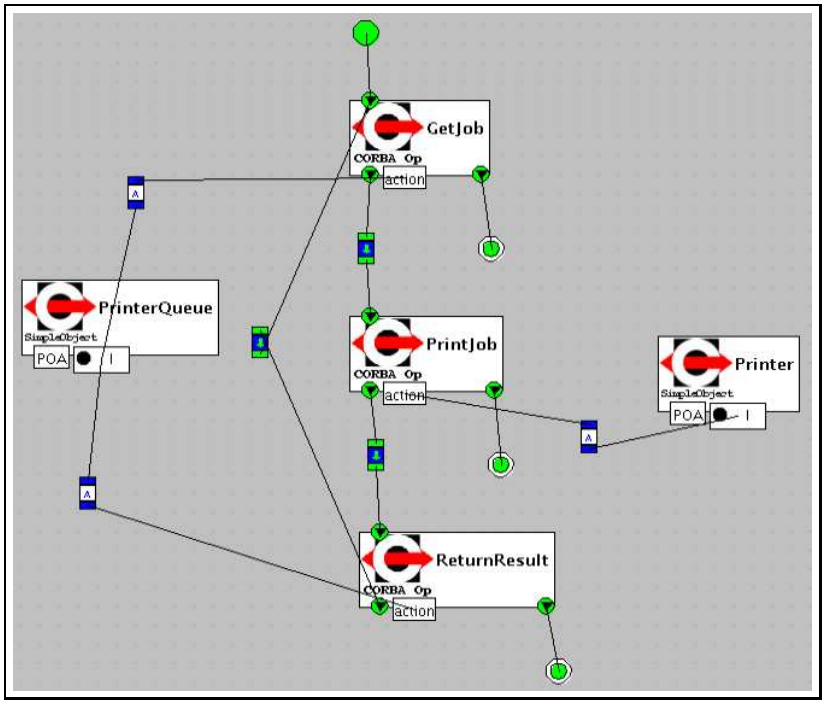

Abbildung 13.2.: Vermittlung zwischen Puffer und Drucker über CORBA

13.3. Verteilte Raumverwaltung

Nach den beiden kleinen Testanwendungen, welche die prinzipielle Benutzung dieses Entwicklungssystems darstellen sollten, wird hier eine komplexere Applikation demonstriert. Diese basiert auf einem Programm aus [BVD01], welches die unterschiedlichen Konfigurationsansätze für POA an einem konkreten System verdeutlicht. Es stellt damit ein gutes Beispiel für fortgeschrittene CORBA-Programmierung dar.

Die Applikation dient zur Verwaltung von Besprechungen. Für diese stehen verschiedene Gebäude mit Besprechungsräumen zur Verfügung. Über eine grafische Schnittstelle kann ein Benutzer für Besprechungen entsprechende Räume reservieren und auch neue Räume in das System integrieren. Alle Daten sind persistent und werden in lokalen Dateien gespeichert. Bis auf eine kleine Änderung wurde der vorgegebene Code eingesetzt, so dass sich diese Applikation auch gut dafür eignet zu untersuchen, wie leicht sich Altsoftware in das Entwicklungssystem integrieren läßt. Die angesprochene Änderung bestand daraus, dass die enge Verzahnung zwischen der grafischen Schnittstelle und der Applikationslogik so aufgebrochen wurde, dass beide auf unterschiedliche Rechner verteilt werden konnten.

Abbildung 13.3 zeigt diese Applikation. Sie lässt sich in drei Teile gliedern: die Gebäudeverwaltung, die Verwaltung der Besprechungen und die Verwaltung der Anwender.

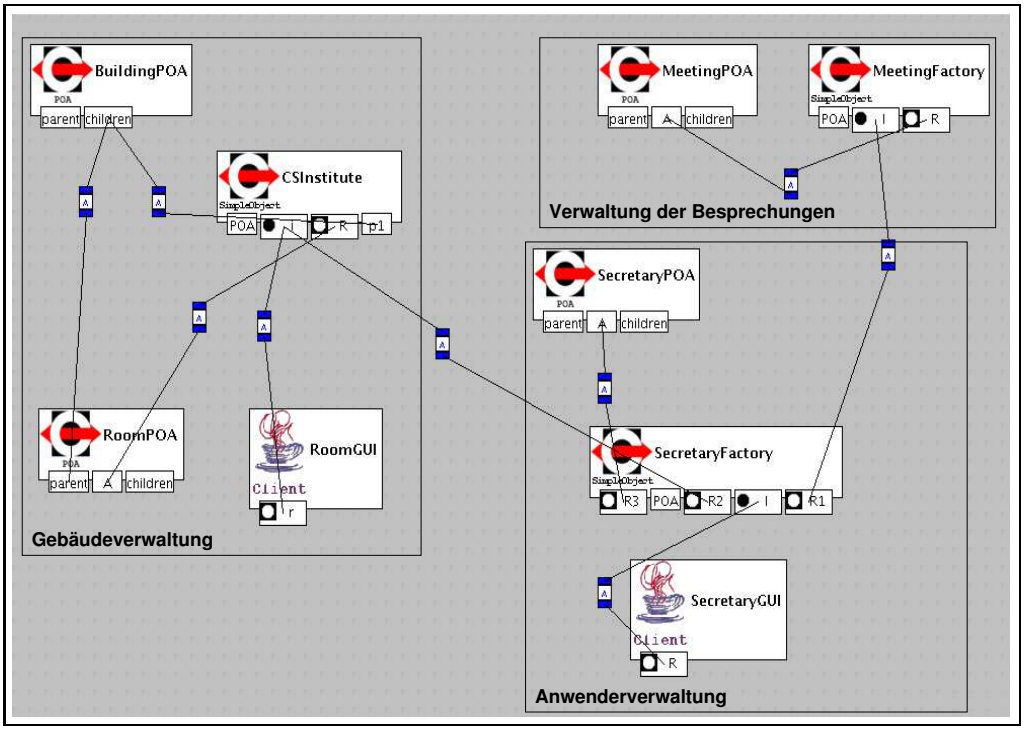

Abbildung 13.3.: Verteilte Raumverwaltung

Gebäudeverwaltung

In der Gebäudeverwaltung werden die einzelnen Gebäude durch Komponenten statisch in der Applikationsarchitektur spezifiziert. In der Abbildung ist nur ein Gebäude vorhanden, das `CSInstitute`. Jedem Gebäude ist eine Menge von Räumen zugeordnet. Zur Laufzeit können für ein Gebäude neue Räume erzeugt werden. Diese Menge kennzeichnet den persistenten Zustand eines Gebäude-Objekts, der in einer Datei gespeichert wird. Immer wenn das Laufzeitexemplar des Gebäudeobjekts erzeugt wird, wird der Zustand aus der Datei geladen. Jede Zustandsänderung wird sofort in die Datei geschrieben.

Da ein Gebäudeobjekt über den Namensdienst auch von anderen externen Klienten benutzt werden kann, die nicht von der Applikationsarchitektur erfasst werden, muss dafür gesorgt werden, dass dessen CORBA-Referenz ebenfalls persistent ist. Hierfür müssen die Attribute der POA-Komponente entsprechend gesetzt werden. Auf die genauen technischen Details wird hier verzichtet[1].

Die einzelnen Räume eines Gebäudes stehen ebenfalls als CORBA-Objekte zur Verfügung. Diese sind allerdings nicht statisch in der Applikationsarchitektur aufgeführt, sondern werden von einem Servant-Verwalter dynamisch

[1] Als kurze Anmerkung ohne weitergehende Erklärung sei hier nur der Ansatz angerissen: Mit dem Setzen der `IDAssignmentPolicy` auf `USER_ID` wird erklärt, dass die Applikation den Objektbezeichner bestimmt und nicht der POA. Das Setzen von `LifespanPolicy` auf `PERSISTENT` erzeugt persistente Referenzen. Schließlich sorgt das Setzen von `RequestProcessingPolicy` auf `USE_ACTIVE_OBJECT_MAP_ONLY` dafür, dass angemeldete Servants automatisch gefunden werden.

erzeugt. Hierfür sind die Attribute von `RoomPOA` entsprechend gesetzt.

Ein Raum enthält die für ihn gebuchten Besprechungen. Diese werden in einer Datei gespeichert. Wird ein Raumobjekt über CORBA angesprochen, für das noch kein Servant erzeugt wurde, so lässt sich der POA von dem eingetragenen Servant-Verwalter ein Servant erzeugen, welcher mit den entsprechenden Werten aus der Datei initialisiert wurde.

Wie schon erwähnt dienen die Gebäudeobjekte auch dazu neue Raumobjekte zu erzeugen. Damit dieses möglich ist, müssen sie aber Zugriff auf den `RoomPOA` haben, an dem die Raumobjekte angeschlossen sind. Daher gibt es eine Verbindung zwischen `RoomPOA` und `CSInstitute`. In dessen Anschluss `R` ist das Attribut `configMethod` auf `setPOA(REQUIRED_OBJ)` gesetzt. So wird dem Laufzeitexemplar zu `CSInstitute` in der Konfigurationsphase das entsprechende Laufzeitexemplar von `RoomPOA` mitgeteilt.

Da eine Referenz auf ein POA-Objekt immer nur eine lokale Referenz und keine entfernte CORBA-Referenz sein kann, muss gewährleistet sein, dass die Laufzeitexemplare von `RoomPOA` und `CSInstitute` sich in dem gleichen Adressraum befinden. Hierfür ist das boolsche Attribut `coloc` in dem verbindenden Konnektor auf `true` gesetzt.

Besprechungsverwaltung

Um die Persistenz von Besprechungen zu gewährleisten, wurde ein anderer Ansatz als bei den Räumen mit dem Servant-Verwalter benutzt. Alle Methodenaufrufe von Besprechungsobjekten werden von einem Standard-Servant bearbeitet. Der verhält sich in Abhängigkeit des Kontextes, in dem er aufgerufen wird, wie das entsprechende Besprechungsobjekt. Die Attribute von `MeetingPOA` sind entsprechend konfiguriert.

Neue Besprechungen werden von der `MeetingFactory` erzeugt. Diese muss hierfür den `MeetingPOA` kennen. Über diesen kann sie auf den Standard-Servant zugreifen, dessen Typ geeignet umwandeln und auf diesem eine neue Besprechung erzeugen. Der letzte Schritt ist möglich, da beide Komponenten, wie oben bei `RoomPOA` und `CSInstitute`, kolokal sind und sich damit im gleichen Adressraum befinden.

Anwenderverwaltung

Um die Persistenz von Anwendern zu gewährleisten, wurde der gleicher Ansatz eines Standard-Servants wie bei den Besprechungen gewählt. Die Attribute der Komponente `SecretaryPOA` sind entsprechend gesetzt. Auch bei der Erzeugung neuer Anwender über die `SecretaryFactory` Komponente wurde der gleiche Ansatz wie bei den Besprechungen gewählt.

Die Komponente `SecretaryGUI` stellt eine grafische Schnittstelle zur Verfügung, um auf das Gesamtsystem zuzugreifen. Jeder Anwender kann sich sein Laufzeitexemplar erzeugen. Die Komponente steht mit der `SecretaryFactory` in einer $* - 1$ Beziehung und benutzt diese, um neue Anwender zu erzeugen.

Da der Code für die Schnittstelle nicht verändert wurde, werden alle anderen Objekte, die von der Schnittstelle benutzt werden, über den Namensdienst lokalisiert. Trotzdem ist die grafische Schnittstelle ohne die ande-

ren Objekte nicht funktionsfähig. Daher wurde in die Applikationsarchitektur Abhängigkeiten von der `SecretaryFactory` zum `CSInstitute` und `MeetingFactory` aufgenommen. Dies sichert, dass immer, wenn eine neue Schnittstelle erzeugt wird, die benötigten Laufzeitexemplare vorhanden sind. Zusätzlich wird auch die Schnittstelle beendet, wenn eines der benötigten Elemente terminiert.

Fazit
Diese Applikation zeigt, dass sich auch komplizierte Anwendungen in dem Entwicklungssystem realisieren lassen. Der Entwickler muss allerdings die speziellen Fähigkeiten von CORBA, die er einsetzen möchte, gut verstanden haben. Dennoch nimmt ihm das System Arbeit ab, da er den konkreten Code für das Aufsetzen der einzelnen Subsysteme und deren Konfiguration nicht schreiben muss.

Bei der Integration von Altsoftware zeigte sich, dass durch den Einsatz des Namensdienstes viele Abhängigkeiten zwischen den einzelnen Komponenten im Code versteckt sind. In der Anwenderverwaltung wurden diese Abhängigkeiten explizit formuliert, wodurch das Laufzeitsystem nun gewährleistet, dass die benötigten Laufzeitexemplare immer rechtzeitig vorhanden sind. Die Stabilität der Applikation hat sich dadurch verbessert.

13.4. Diskussion und offene Punkte

Die in den vorangehenden Kapiteln präsentierten Fallstudien haben gezeigt, dass sich mit ECL-System und den beiden Erweiterungsmodulen verteilte Informations- und Kontrollsysteme unter Einsatz von CORBA als Middleware komfortabel und adäquat erstellen lassen. Unter adäquat wird hier verstanden, dass Applikationen mit einfacher Struktur durch eine einfache Applikationsarchitektur beschrieben werden können, während für Applikationen mit komplexeren Strukturen auch komplexere Applikationsarchitekturen notwendig sind, deren Architekturelemente zusätzlich umfassend zu parametrisieren sind.

Insbesondere bei der Nutzung besonderer Fähigkeiten von CORBA, wie persistente oder virtuelle Objekte, muss der Applikationsentwickler deren Konzepte zuerst gut verstanden haben, bevor er sie in seiner Applikationsarchitektur einsetzen kann. Der Einsatz von ECL beseitigt damit nicht die Notwendigkeit, sich zuerst in das Zielsystem einarbeiten zu müssen. Der Applikationsentwickler muss sich aber nicht in die konkrete Programmierschnittstelle einarbeiten und er braucht ebenfalls keinen Code zu implementieren, der für den Einsatz dieser besonderen Fähigkeiten notwendig ist. Statt dessen kann er sich auf die Implementierung der Applikationslogik konzentrieren. Das ECL-System sorgt dann zur Laufzeit dafür, dass diese korrekt in das CORBA-System eingefügt wird.

Durch den Einsatz des ECL-Systems fällt nur in Konfigurationsphasen, also während des Startens und Terminierens von Teilen der Applikation, ein zusätzlicher Rechen- und Kommunikationsaufwand an. Da die Laufzeitun-

terstützung in der restlichen Zeit passiv ist, verbraucht sie nur wenig Arbeitsspeicher und keine Rechenressourcen. Der zusätzliche Aufwand ist damit vernachlässigbar.

Der deklarative Ansatz, eine verteilte Applikation durch Spezifikation von Abhängigkeiten zwischen den einzelnen Komponenten zu implementieren, ermöglicht es zu gewährleisten, dass sich die Applikation immer in einem konsistenten Zustand befindet. Der Anwender kann jederzeit neue Laufzeitexemplare in der Applikation starten, bzw. Laufzeitexemplare terminieren. Die Laufzeitunterstützung sorgt dafür, dass alle Abhängigkeiten stets erfüllt sind.

Die Eigenschaft der Konsistenz gilt aber nur unter der Annahme, dass alle Abhängigkeiten zwischen den Komponenten in der Applikationsarchitektur korrekt ausgedrückt sind. Gilt dies nicht, sind inkonsistente Zustände möglich. Die Fallstudie aus Kapitel 13.3 hat demonstriert, dass besonders bei dem Einsatz spezieller Fähigkeiten des Zielsystems Abhängigkeiten auftreten können, die in dem von dem Applikationsentwickler implementierten Code der Applikationslogik versteckt sind. Die Ursache hierfür ist, dass CORBA nicht für das ECL-System sondern für die Kombination mit Programmiersprachen konzipiert wurde, die im allgemeinen imperativ sind.

Wenn sich derartige versteckte Abhängigkeiten nicht vermeiden lassen, dann muss der Applikationsentwickler diese durch Einsatz von zusätzlichen Konnektoren in der Applikationsarchitektur explizit ausdrücken, sofern dies möglich ist. Es ist dann nicht möglich, wenn ein Objekt der Applikation nicht durch eine Komponente in der Applikationsarchitektur dargestellt ist, weil es z.B. in der Applikationslogik zur Laufzeit erzeugt wurde. In dem Fall muss eine Komponente, die in der Applikationsarchitektur enthalten ist, dieses versteckte Objekt verwalten.

Ähnlich wie bei dem Amica-System gibt es zwei wichtige offene Punkte in dem hier vorgestellten System für die Implementierung CORBA-basierter Applikationen: *Fehlertoleranz* und *Sicherheit*. Um Fehlertoleranz in der Applikationsklasse verteilter Informations- und Kontrollsysteme zu erzielen, bietet es sich an, redundante Dienstanbieter einzusetzen. Fällt einer aus, können die zugehörigen Replikate die Aufgabe übernehmen, ohne dass der Anwender es merkt. Die OMG hat für dieses Problem eine Spezifikation veröffentlicht [Gro02]. Für CORBA existieren einige Systeme, welche redundante Dienstanbieter unterstützen [GNSY00, LM97].

Um dieses auf ECL-Seite zu unterstützen, müsste der Architekturelementtyp `viks:ServiceProvider` um ein Attribut für die Anzahl der gewünschten Replikate erweitert werden. Zusätzlich muss die Laufzeitunterstützung des CORBA-Erweiterungsmoduls an das spezielle CORBA-System angepasst werden. Mit diesem Ansatz würde mit einem moderaten Implementierungsaufwand eine Fehlertoleranz erreicht werden.

Sicherheit in verteilten Informations- und Kontrollsystemen besteht üblicherweise im Wesentlichen aus einem Zugriffschutz für die einzelnen Methoden der verteilten Objekte. Mit dem in dem Projekt Raccoon entwickelten System ist es möglich, dynamische Sicherheitspolitiken für CORBA-basierte