Ein Client-Framework für Swing

Tim Wellhausen

Lilienstraße 20
81669 München
Tel. 089 / 44 14 27 45
kontakt@tim-wellhausen.de
http://www.tim-wellhausen.de

Version vom 04.12.2004

Zusammenfassung: Trotz einer leistungsfähigen Bibliothek ist die Entwicklung einer sowohl bedienerfreundlichen als auch gut strukturierten Client-Anwendung mit Swing immer noch eine Herausforderung. In umfangreichen Informationssystemen verliert der Client-Programmcode häufig jegliche Struktur und lässt sich kaum noch verändern. Die Entwickler sind in solchen Fällen mehr damit beschäftigt, den vorhandenen Code zu verbessern, als neue Anforderungen umzusetzen. Hilfe bieten Frameworks, die die einzelnen Bestandteile einer Client-Anwendung dauerhaft entkoppeln und somit robuste Anwendungen ermöglichen. Anhand eines frei verfügbaren Beispiel-Frameworks stellt dieser Artikel bewährte Konzepte dafür vor.

Einführung

Ein Framework bildet die grundlegende Infrastruktur einer Client-Anwendung. Indem es die Kommunikationsstruktur zwischen den Bestandteilen einer Anwendung vorgibt, regelt es die Abhängigkeiten innerhalb der Codebasis. Ohne ein Framework verliert eine Anwendung schnell ihre Struktur. Die einzelnen Bestandteile drohen immer enger aneinander gekoppelt zu werden, bis schließlich ein Monolith entsteht – schwierig zu verstehen und kaum zu verändern.

Auf der Client-Seite gibt es keine etablierte Abstraktionsschicht, die die Entwickler von den Details der Oberflächenentwicklung abschirmt. Auf der Server-Seite hingegen hat sich dazu die Java 2 Enterprise Edition (J2EE) mit ihren Enterprise Java Beans bewährt; eine analoge Technologie für den Client – eine Art Client-Container – existiert nicht.

Ein standardisiertes Framework für Swing wird wahrscheinlich auch in absehbarer Zeit nicht zur Verfügung stehen. Unternehmen, die große Anwendungen bzw. eine Familie von Anwendungen mit Swing entwickeln, erstellen deswegen häufig eigene Client-Frameworks.

Die Anforderungen an solche Client-Framework hängen davon ab, welche Funktionalität die damit erstellten Anwendungen aufweisen sollen. Je flexibler ein Framework sein soll, desto komplexer wird seine Struktur und desto schwieriger wird es, damit eine Anwendung zu entwickeln. Um die Komplexität zu reduzieren, hilft es, zunächst eine Richtlinie für das Aussehen sowie die Bedienung der zukünftigen Client-Anwendungen aufzustellen und mit einem Framework nur diese Richtlinie zu unterstützen.

Über die rein technischen Aspekte hinaus sorgt ein Framework auch dafür, dass die Applikationen einheitlich aussehen und sich gleichartig bedienen lassen. Unternehmen mit einer durchgängigen *Corporate Identity* legen auch bei ihren Anwendungen Wert auf eine gewisse Einheitlichkeit.

Beispiel

Als Beispiel, auf das im Verlauf des Artikels häufig verwiesen wird, dient die Entwicklung einer Kundenverwaltung. Abbildung 1 zeigt einen Bildschirmabdruck dieser Anwendung, die die folgenden Funktionen enthält:

- Mit Hilfe eines Suchdialogs können Anwender im vorhandenen Datenbestand nach Kundeneinträgen suchen. (1)
- Eine Tabelle listet Kundendaten auf. Die dargestellten Daten stammen beispielsweise aus einem Suchergebnis. (2)
- Ein Formular stellt den Kundendatensatz dar, der in der Tabelle selektiert ist. (3)
- Kundendatensätze werden angelegt und gelöscht, zudem der Datenbestand geladen und gespeichert (um das Beispiel einfach zu halten, ist keine Datenbank integriert).

Abb. 1: Eine Kundenverwaltung als Beispiel-Anwendung

Ansatz und Konzepte

Der vorliegende Artikel stellt Techniken und Konzepte für ein Client-Framework vor, die sich in der Praxis bewährt haben. Die drei folgenden Konzepte haben sich dabei als die wichtigsten herausgestellt:

1. *Entkopplung*: Die Funktionalität einer Client-Anwendung wird in voneinander weitestgehend unabhängigen Modulen implementiert. Die Bindung von Modulen geschieht entweder durch das Versenden von Nachrichten oder durch den Aufruf von Diensten.[1]
2. *Trennung von Darstellung und Funktionalität*: Module bestehen aus grafischen Komponenten für die Darstellung von Formularen und Dialogen sowie aus Diensten für die Implementierung fachlicher oder technischer Funktionalität.
3. *Konfiguration*: Module werden zur Laufzeit durch Konfigurationsdaten zu einer Applikation verbunden. In diesen Konfigurationsdaten stehen unter anderem die Namen der Mo-

1 Der Begriff *Modul* wird in diesem Artikel ohne historischen Kontext verwendet. Er dient unter anderem zur Abgrenzung von dem Konzept der Plugins, wie sie beispielsweise im Eclipse-Projekt ([2]) eingesetzt werden.

dul-Klassen, die benötigten Ressourcen (Texte und Verweise auf Symbole) und die Zusammensetzung der Menü- bzw. Werkzeugleiste.

Konfigurationsdaten sind eine wichtige Grundlage für eine modulare Entwicklung. Wenn der Programmcode alle Informationen über eine Anwendung enthält, ist es schwierig, Teile einer Anwendung zu extrahieren und wiederzuverwenden. Deshalb werden alle veränderlichen Eigenschaften einer Client-Anwendung aus dem Programmcode entfernt und in Konfigurationsdateien gesammelt.

Das Framework liest die Konfigurationsdaten aller Module ein und verarbeitet sie. Es kapselt die Logik, wie eine Anwendung gestartet wird, es erzeugt und initialisiert die einzelnen Module, es erstellt die Menü- und Werkzeugleiste und bietet dann den Modulinstanzen seine Dienste an. Ein Modul hat somit weder die Kontrolle über seinen eigenen Lebenszyklus, noch über die Startphase der Anwendung.

Die interne Implementierung eines Moduls unterliegt nur wenigen Vorgaben. Um ein Modul in eine Anwendung zu integrieren, muss es jedoch Schnittstellen implementieren und Konfigurationsdateien bereitstellen, die das Framework vorgibt. Die Implementierung eines Moduls ist somit verborgen; nur die Schnittstellen seiner Dienste sind innerhalb einer Anwendung bekannt.

Module ermöglichen es, Formulare und Dialoge von der Geschäftslogik zu trennen, indem die Geschäftslogik in eigene, lokale Dienste extrahiert wird. Auch der Zugriff auf einen Applikationsserver kann in Dienste gekapselt werden, sodass sich unterschiedliche Implementierungen für den Produktiv- und Testbetrieb leicht austauschen lassen (z. B. BUSINESS DELEGATE-Pattern, Unit-Tests).

Ein Modul wird durch Auskommentierung seiner Konfigurationseinträge deaktiviert. Die Funktionalität dieses Moduls ist dann zwar nicht mehr vorhanden; der Rest der Anwendung arbeitet aber wie zuvor. In der Praxis lässt sich diese Idealvorstellung nicht bis in letzter Konsequenz durchhalten. Es ist jedoch möglich, durch einen sorgfältigen Entwurf der Abhängigkeiten zwischen den Modulen dieser Vorstellung nahe zu kommen.

Der hier vorgestellte Ansatz erfordert es, dass die Anwendungsentwickler von Beginn an planen, in welche Module, Dienste und grafischen Komponenten sie die Bestandteile einer Anwendung zerlegen. Der anfängliche Programmieraufwand steigt dadurch zwar an, da unter anderem zusätzliche Schnittstellen definiert werden müssen. Langfristig führt dieser Ansatz jedoch dazu, dass die Abhängigkeiten zwischen den Bestandteilen einer Anwendung besser zu überblicken und zu kontrollieren sind.

Der Ansatz weist Parallelen zu der J2EE-Technologie auf: Der EJB-Container verwaltet EJB-Komponenten, die auf ihren Lebenszyklus ebenfalls keinen Einfluss haben. Auch EJB-Komponenten werden über Schnittstellen angesprochen, und ihre Einstellungen sind in Konfigurationsdateien ausgelagert. Sie kommunizieren entweder über direkte Aufrufe oder über das Versenden von JMS-Nachrichten.

Komponenten des Frameworks

Nach den allgemeinen Konzepten geht es nun um ein konkretes Framework, das zur Illustration dieses Artikels entwickelt wurde und frei verfügbar ist [1]. Zunächst werden die wichtigsten Bestandteile vorgestellt, die den Kern des Frameworks bilden:

- Ein *Ressourcenverwalter* sammelt zentral Bezeichnungen, Symbole, Tastaturkürzel, Tooltipps und Meldungen.

- Ein *Modulverwalter* (`ModuleManager`) liest beim Start einer Anwendung die Konfigurationsdaten der Anwendung, initialisiert die dort aufgelisteten Module und bietet ihnen zur Laufzeit Zugriff auf die Funktionalität des Frameworks.
- Ein *Verwalter grafischer Komponenten* (`ComponentManager`) sorgt dafür, dass die Komponenten auf der Arbeitsoberfläche angezeigt werden. Dazu bettet er sie in Fenster ein und bietet Funktionen an, um diese Fenster einzublenden und zu verbergen.
- Ein *Dienstevermittler* (`ServiceBroker`) verwaltet zentral alle Dienste, die die Module sich gegenseitig bereitstellen, und vermittelt den Zugriff darauf.
- Ein *Nachrichtenübermittler* (`MessageBus`) erlaubt es den Modulen, untereinander Mitteilungen zu versenden, ohne dass die Module dadurch eng aneinander gekoppelt werden. Er dient daher der asynchronen, entkoppelten Kommunikation zwischen Modulen.

Die Verständigung zwischen dem Framework und den Modulen ist vollständig durch Schnittstellen definiert. Auf der einen Seite greifen die Module nur über Schnittstellen auf die Funktionalität zu, die das Framework zur Laufzeit bereitstellt. Auf der anderen Seite muss ein Modul Schnittstellen implementieren, die das Framework vorgibt, damit es in eine Anwendung integriert werden kann. In Abbildung 2 sind diese Schnittstellen mit ihren wichtigsten Funktionen aufgeführt. Die Komponenten des Frameworks sind gelb eingefärbt, die Schnittstellen, die die Module implementieren, grün.

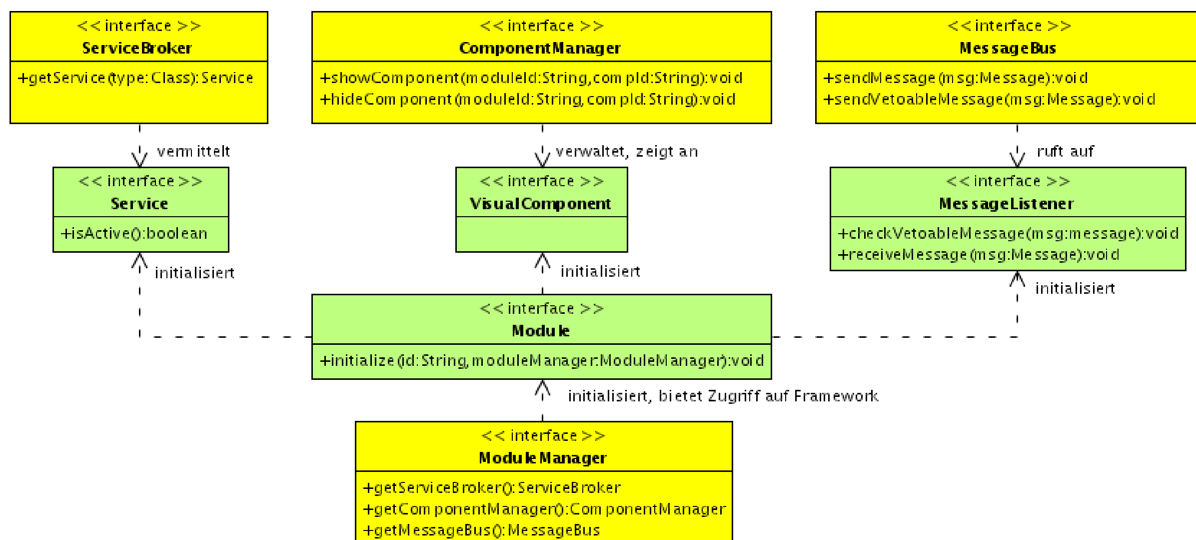

Abb. 2: Die Schnittstellen des Frameworks

Entwicklung eines Moduls

Die Funktionalität des Frameworks und die Entwicklung eines Moduls werden in den folgenden Abschnitten anhand der Kundenverwaltung vorgeführt. Der erste Schritt bei der Entwicklung einer Anwendung besteht darin, ihre Funktionalität in einzelne Module aufzuteilen. In der Kundenverwaltung lassen sich beispielsweise folgende Module finden (die Ziffern verweisen auf Abbildung 1):

- Ein erstes Modul enthält den Suchdialog (1) und stellt einen Dienst zur Verfügung, der Suchen ausführt.
- Ein zweites Modul besteht aus der Tabelle für die kompakte Darstellung von Kundendaten (2).

- Ein drittes Modul enthält das Formular, in dem Kundendaten verändert werden können (3).
- Ein viertes Modul stellt einen Dienst zur Verfügung, der die Kundendaten im Speicher hält und Operationen zum Anlegen, Abfragen, Ändern und Löschen anbietet. Dieses Modul enthält keine für den Anwender sichtbaren Bestandteile.

Die Module sind mit ihren Diensten und grafischen Komponenten in Abbildung 3 dargestellt. Die Klassen der Kundenverwaltung sind, wie in allen folgenden Abbildungen, blau eingefärbt.

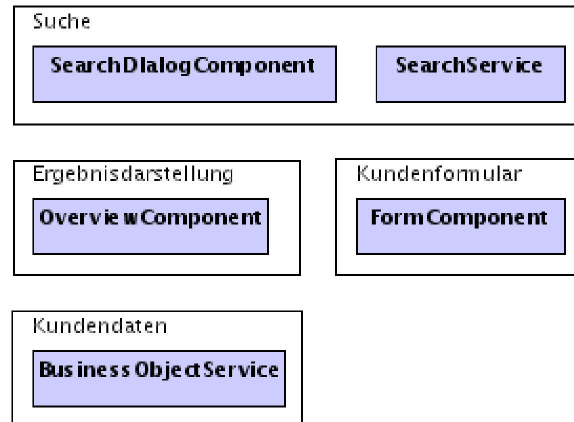

Abb. 3: Die Module der Kundenverwaltung

Das Modul „Suche“ dient als Beispiel, wie ein Modul initialisiert und seine Bestandteile vom Framework verwaltet werden. Es besteht aus drei Klassen und einer Schnittstelle (s. Abb. 4).

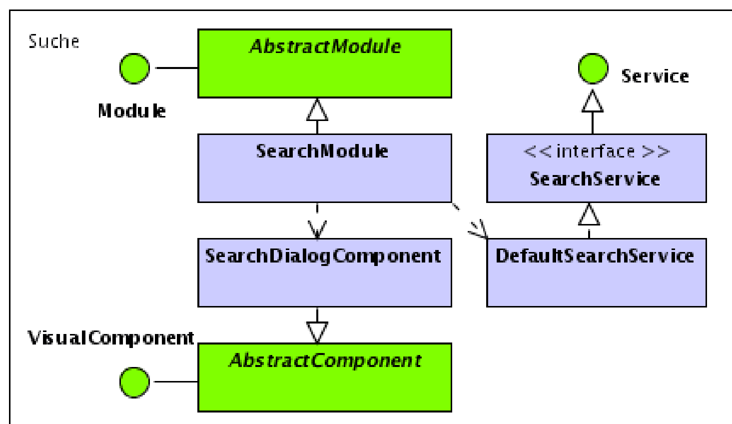

Abb. 4: Die Klassen und Schnittstellen des Suchmoduls

Die Klasse `SearchModule` ist die Hauptklasse des Suchmoduls und implementiert die Schnittstelle `Module`, indem sie die vom Framework bereitgestellte Basisklasse `AbstractModule` erweitert. Beim Start der Anwendung erzeugt das Framework eine Instanz von `SearchModule` und ruft die Initialisierungsmethode `initialize` auf mit der eindeutigen Bezeichnung des Moduls als Parameter (s. Abb. 5). Während der Initialisierung erstellt das Suchmodul die grafische Komponente für den Suchdialog (die Klasse `SearchDialogComponent`) und den Suchdienst (die Klasse `DefaultSearchService`).

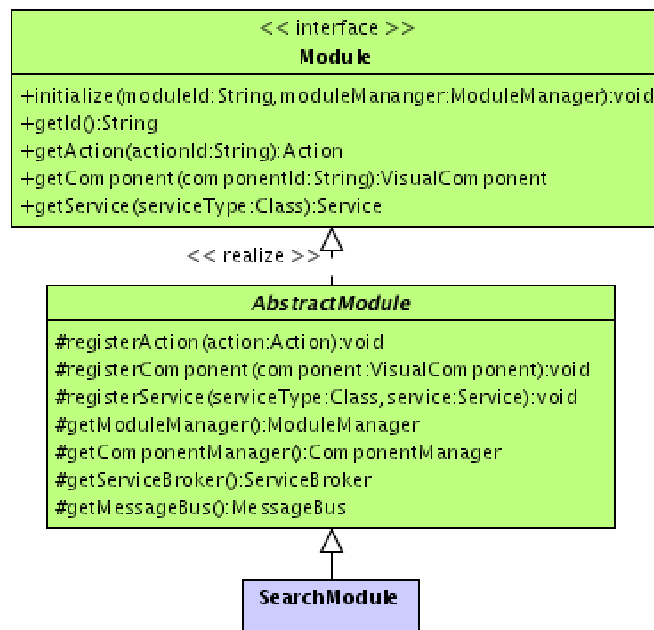

Abb. 5: Die Schnittstelle `Module`

Initialisierung

Auf den folgenden Seiten wird das Zusammenspiel zwischen dem Framework und den Klassen des Moduls näher erläutert. Dazu werden zunächst die Konfigurationsdaten und die Initialisierung des Suchmoduls zusammenhängend abgedruckt. Zunächst die Konfigurationsdaten:

```
<module id="SearchModule" class="SearchModule">
  <menubar>
    <menu id="customer.menu" behindof="edit.menu">
      <menuitem id="customer.search.show"/>
    </menu>
  </menubar>
  <toolbar>
    <button id="customer.search.show"/>
  </toolbar>
  <services>
    <service interface="SearchService"/>
  </services>
  <visualcomponents>
    <visualcomponent id="customer.search.component" modal="false">
      <size width="225" height="125"/>
    </visualcomponent>
  </visualcomponents>
</module>
```

Das Suchmodul erhält keinen Zugriff auf seine Konfigurationsdaten. Stattdessen liest der Modulverwalter die Daten ein und entnimmt ihnen alle wesentlichen Informationen über das Suchmodul, dessen Bestandteile und die gewünschte Integration des Moduls in die Anwendung.

Der folgende Programmcode zeigt die Implementierung der `initialize`-Methode des Suchmoduls:

```
public void initialize(String moduleId,
                       ModuleManager moduleManager)
  throws ConfigurationException
{
```

```
  super.initialize(moduleId, moduleManager);

  // Action-Objekt anlegen und registrieren, um Suchdialog zu öffnen
  ShowComponentAction showComponentAction =
    new ShowComponentAction(
      "customer.search.show",             // Id des Action-Objekts
      moduleManager.getComponentManager(), // Referenz auf Framework
      moduleId,                           // Id des Suchmoduls
      "customer.search.component");       // Id des Suchdialogs
  registerAction(showComponentAction);

  // Suchdienst anlegen und registrieren
  SearchService searchService =
    new DefaultSearchService(getServiceBroker());
  registerService(SearchService.class, searchService);

  // Suchdialog anlegen und registrieren
  SearchDialogComponent searchDialogComponent =
    new SearchDialogComponent(
      moduleId,                           // Id des Suchmoduls
      "customer.search.component",        // Id des Suchdialogs
      moduleManager,                      // Referenz auf Framework
      searchService);                     // Referenz auf Suchdienst
  registerComponent(searchDialog);
}
```

Das Modul initialisiert alle seine Bestandteile selbst und registriert sie intern (geregelt von der Basisklasse `AbstractModule`). Das Framework kennt über die Konfigurationsdaten die Bezeichnungen der Bestandteile und erhält sie auf Anfrage von dem Modul.

Eine Alternative zu dieser Vorgehensweise besteht darin, in den Konfigurationsdaten eines Moduls nicht auf eindeutige Bezeichner der Aktionen, Dienste und Komponenten des Moduls zu verweisen, sondern auf deren Klassennamen. In diesem Fall wäre das Framework dafür verantwortlich, diese Bestandteile zu erzeugen und zu initialisieren.

Diese alternative Vorgehensweise hat jedoch zwei Nachteile: Zum einen wird die Initialisierung der Bestandteile eines Moduls komplizierter, da sich jeder Bestandteil selbst alle Informationen beschaffen muss, die ansonsten das Modul übergeben kann. Zum anderen geht die (gewünschte) enge Bindung innerhalb eines Moduls verloren.

Grafische Komponenten

Grafische Komponenten werden von der Klasse `JPanel` abgeleitet und implementieren zusätzlich die Schnittstelle `VisualComponent` (s. Abb. 6). Eine Standardklasse des Frameworks, `AbstractComponent`, erleichtert die Implementierung einer grafischen Komponente.

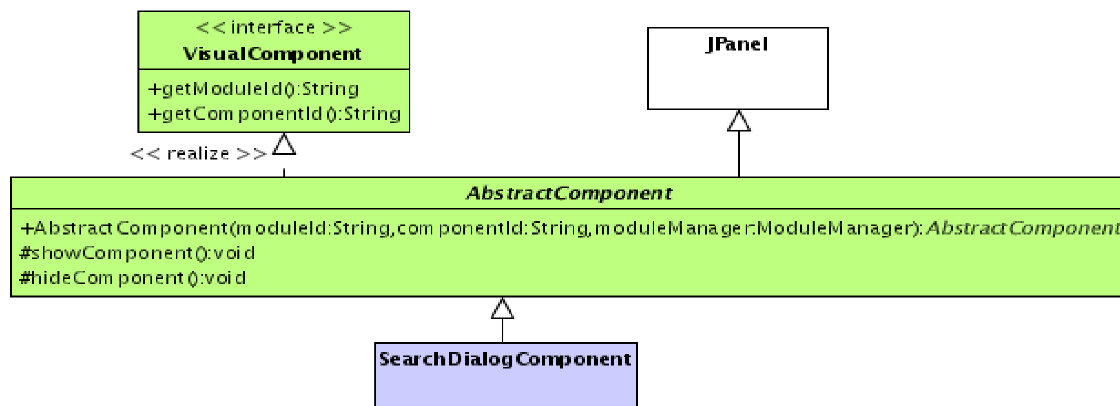

Abb. 6: Die Schnittstelle `VisualComponent`

Für die grafischen Komponenten ist der Komponentenverwalter verantwortlich. Er bettet eine grafische Komponente in ein Fenster ein und zeigt das Fenster mit der Komponente an. Jede Komponente hat eine eindeutige Bezeichnung, die in den Konfigurationsdaten bestimmt wird. Unter Angabe dieser Bezeichnung kann eine grafische Komponente jederzeit mit Hilfe des Komponentenverwalters angezeigt werden (s. auch Abb. 2).

Die Trennung der grafischen Komponenten von ihren Fenstern ist sinnvoll, um grafische Komponenten sowohl innerhalb modaler als auch nicht-modaler Fenster anzeigen zu können. Die Entscheidung trifft der Entwickler durch Einträge in den Konfigurationsdaten. Der Komponentenverwalter bettet die grafischen Komponenten dementsprechend entweder in `JInternalFrame`- oder `JDialog`-Instanzen ein.

Diese Trennung erhält zudem die Flexibilität des Frameworks. Angenommen, grafische Komponenten sollen nicht mehr in frei beweglichen Fenstern dargestellt werden, sondern andockbar, wie es z. B. in Eclipse üblich ist. Dies ließe sich durch den Austausch des Komponentenverwalters realisieren. Änderungen an den Client-Anwendungen wären nicht notwendig.

Der zweite Teil dieses Artikels folgt in der nächsten Ausgabe von JavaSPEKTRUM. Er behandelt die Kommunikation zwischen Modulen und erläutert, wie Module in eine Client-Anwendung integriert werden.

Der erste Teil dieses Artikels hat bewährte Konzepte für die Entwicklung von Client-Frameworks erläutert. Aufbauend auf dem dort eingeführten Beispiel einer Kundenverwaltung bespricht dieser zweite Teil, wie Module, die zentralen Bestandteile einer Client-Anwendung, miteinander kommunizieren und wie sie in eine Client-Anwendung integriert werden.

Interaktion zwischen den Modulen

Zur Interaktion zwischen Modulen bieten sich zwei Konzepte an: Dienste und Nachrichten. Dienste haben definierte, öffentliche Schnittstellen, können von jedem Modul aufgerufen werden und liefern optional Rückgabewerte zurück. Nachrichten hingegen werden an Nachrichtenkanäle versendet. Die Sender besitzen keine direkten Objektreferenzen auf die Empfänger und wissen nicht, wer die Nachrichten empfängt. Beide Konzepte werden in den folgenden Abschnitten erläutert.

Als Beispiel dient die Suche aus der Kundenverwaltung (s. Abb. 1): Ein Anwender startet eine Suche. Dazu erzeugt der Suchdialog eine Suchanfrage anhand der eingegebenen Daten und richtet diese Anfrage an den Suchdienst. Der Suchdienst führt die Suchanfrage aus und liefert Ergebnisdaten zurück. Die Ergebnisdaten werden an die Übersichtstabelle übermittelt und dort dargestellt. Der erste Eintrag wird in der Tabelle automatisch markiert und an das Kundenformular weitergeleitet, in welchem alle Detailinformationen des markierten Kundendatensatzes angezeigt werden.

Vermittlung von Diensten

Jeder Dienst besteht aus einer öffentlichen Schnittstelle und einer privaten Implementierung. Die Trennung von Schnittstelle und Implementierung ist erforderlich, da Dienste für alle Module erreichbar sind und kein Modul von der Implementierung eines anderen Moduls abhängen darf. Der Dienstevermittler besorgt sich nach der Initialisierung eines Moduls über die Methode `getService` der Schnittstelle `Module` Referenzen auf alle Dienste, die in den Konfigurationsdaten definiert wurden. Anschließend kann er allen Modulen den Zugriff auf die Dienste vermitteln.

Die Kundenverwaltung enthält zwei Dienste: Der eine lädt und speichert die Kundendaten, der andere sucht nach ihnen im Datenbestand. Der Suchdienst besteht beispielsweise aus der öffentlichen Schnittstelle `SearchService` und der privaten Implementierung `DefaultSearchService` (s. Abb. 7). Die Schnittstelle `SearchService` erbt von `Service`, damit sie vom Framework als Dienst akzeptiert wird.

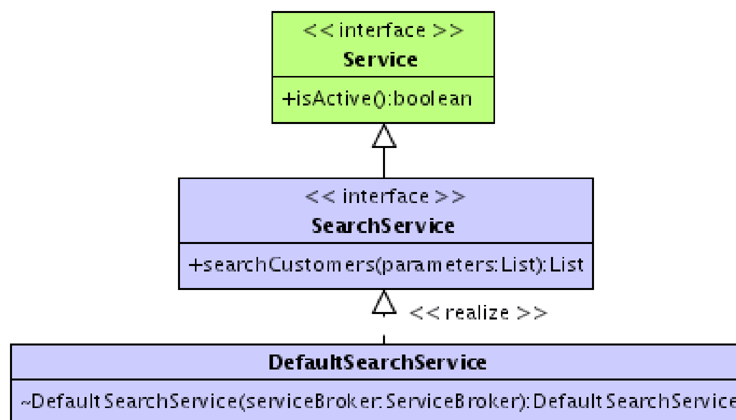

Abb. 7: Die Schnittstelle `Service`

Ein Dienst erhält typischerweise vom Modul alle benötigten Informationen und Referenzen. Das Suchmodul muss beispielsweise auf die bestehenden Kundenobjekte zugreifen können, um die für eine Suchanfrage relevanten Kundenobjekte zu finden. Dazu wird dem Suchdienst ein Verweis auf den Dienstevermittler übergeben. Bei Bedarf holt sich der Suchdienst über diesen Dienstevermittler eine Referenz auf den Geschäftsobjektdienst (`BusinessObjectService`):

```
BusinessObjectService businessObjectService = (BusinessObjectService)
  serviceBroker.getService(BusinessObjectService.class);
```

Dieser Aufruf ähnelt der Vorgehensweise, aus einem `InitialContext`-Objekt einen Verweis auf eine Enterprise Bean zu erhalten. Alternativ könnte der Dienstevermittler alle Dienste in einer JNDI-Struktur registrieren. Dies verspricht jedoch nur wenige Vorteile, da die hier vorgestellten Dienste per Definition nur lokal in einer Client-Anwendung verfügbar sind.

Übermittlung von Nachrichten

Nachrichten werden innerhalb des vorgestellten Frameworks nach dem PUBLISHER SUBSCRIBER-Muster versendet. Alternative Kommunikationsmodelle wären ebenso möglich; für die meisten Fälle reicht dieses Modell jedoch aus.

Die Grundlage des Kommunikationssystems sind Nachrichtenkanäle. Jedes Objekt, das sich an einem Nachrichtenkanal registriert, erhält alle Nachrichten übermittelt, die an den Kanal versendet werden. Um als Sender eine Zustandsänderung publik zu machen, reicht es aus, eine Nachricht an einen Kanal zu schicken. Die Empfänger sind dann dafür verantwortlich, darauf geeignet zu reagieren.

Nachrichtenkanäle werden in den Konfigurationsdaten der Client-Anwendung explizit aufgelistet. Sie sind nicht Teil der Konfiguration der einzelnen Module, da sie Modul übergreifend verwendet werden. Hier als Beispiel die Nachrichtenkanäle der Kundenverwaltung:

```
<messagebus>
  <channel name="broadcast"/>
  <channel name="customersearch"/>
  <channel name="customer"/>
</messagebus>
```

Eine Nachricht wird verschickt, indem der Sender ein Nachrichtenobjekt erstellt und dieses unter Angabe des Nachrichtenkanals dem Nachrichtenübermittler übergibt. Der Suchdialog der Kundenverwaltung versendet beispielsweise eine Nachricht mit dem Ergebnis einer Suche an den Kanal „`customersearch`“:

```
List customers = searchService.searchCustomers(parameters);
Message message = new Message("result", customers);
getMessageBus().getChannel("customersearch").sendMessage(message);
```

Um Nachrichten zu empfangen, die an einen bestimmten Nachrichtenkanal versendet werden, muss die Schnittstelle `MessageListener` implementiert und ein Objekt der implementierenden Klasse beim Nachrichtenübermittler registriert werden.

In der Kundenverwaltung lässt sich das Modul mit der Übersichtstabelle davon informieren, wenn ein neues Suchergebnis vorliegt. Dazu implementiert es die Schnittstelle `MessageListener` und registriert sich selbst. Die Methode `receiveMessage` überprüft die Bezeichnung der Nachricht, übernimmt das Datenobjekt und aktualisiert die Tabelle, in der die Kundendaten angezeigt werden:

```
public class OverviewModule extends AbstractModule implements MessageListener
{
  public void initialize(...) {
    // ...
    Channel channel = getMessageBus().getChannel("customersearch");
    channel.registerMessageListener(this);
  }

  public void receiveMessage(Message message) {
    if (message.getCommand().equals("result")) {
      List customers = (List) message.getData();
      overviewComponent.updateTable(customers);
    }
  }
}
```

Der Sender einer Nachricht erhält in diesem Szenario keine Antwort vom Empfänger (asynchrone Kommunikation). Der Empfänger hat daher keine Möglichkeit, dem Sender zu über-

mitteln, ob er die Nachricht angenommen hat. Daher unterstützt das Framework Nachrichten mit einem Vetorecht für die Empfänger.

In der Kundenverwaltung versendet beispielsweise die Übersichtstabelle eine Nachricht, wenn ein Anwender einen Datensatz markiert, sodass im Kundenformular immer der momentan markierte Datensatz angezeigt wird. Wenn der Anwender jedoch den zuletzt markierten Datensatz im Kundenformular verändert und noch nicht gespeichert hat bzw. nicht speichern kann, muss das Kundenformular die noch nicht gespeicherten Werte beibehalten und den neu markierten Datensatz ablehnen.

Für solche Fälle werden Nachrichten in zwei Phasen versendet. In der ersten Phase erhält jeder potenzielle Empfänger die Nachricht zur Prüfung. Nur dann, wenn alle Empfänger die Nachricht akzeptieren, wird sie wirklich an die Empfänger versendet. Falls hingegen ein einziger Empfänger die Nachricht ablehnt, wird sie verworfen und der Sender darüber informiert.

Realisiert wird dieses Verhalten über einen Veto-Mechanismus, ähnlich wie Swing ihn intern für die Verbreitung von Zustandsänderungen nutzt: Eine Nachricht wird explizit als Veto-Nachricht verschickt. Um sie abzulehnen, reicht es als Empfänger aus, eine `VetoException` zu werfen. In der Übersicht der Kundenverwaltung verläuft dies so:

```
Customer selectedCustomer = customerTableModel.getSelectedCustomer();
Message message = new Message("selectionchanged", selectedCustomer);
try {
  getMessageBus().getChannel("customer").sendVetoableMessage(message);
}
catch (MessageVetoException e) {
  // Selektion wird rückgängig gemacht
}
```

Dienste oder Nachrichten

Dienste und Nachrichten weisen jeweils eigene Vor- und Nachteile auf, die sich auf die Architektur einer Client-Anwendung auswirken. Deswegen lohnt ein Vergleich beider Konzepte.

Dienste bündeln logisch zusammenhängende Funktionalität. Sie helfen, Geschäftslogik vom Programmcode für Dialoge und Formulare zu trennen. Ein Nachteil von Diensten besteht darin, dass der Programmcode Verweise auf die Schnittstellen der Dienste enthält und es somit schwierig ist, die Module, welche die Dienste bereitstellen, zu deaktivieren. Dienste koppeln also die Module einer Anwendung aneinander. Daher sollten möglichst nur Basis-Funktionen, die auf jeden Fall benötigt werden, als Dienste modelliert werden.

Der Austausch von Nachrichten über konfigurierbare Kanäle löst die statischen Bindungen im Programmcode. Ein Modul versendet eine Nachricht, um auf eine Zustandsänderung aufmerksam zu machen, ohne dass das Modul selbst auf diese Nachricht angewiesen ist. Nachrichten haben aber den Nachteil, dass die Kommunikationsstruktur einer Anwendung in kurzer Zeit unübersichtlich werden kann. Insbesondere, wenn mehrere Entwickler ohne Absprache untereinander zwischen den Modulen Nachrichten versenden, passiert es schnell, dass manche Nachrichten so lange zirkulieren, bis das Programm sich durch das Überlaufen des Stacks beendet.

Integration in eine Client-Anwendung

Nachdem zuletzt die dynamischen Eigenschaften von Modulen behandelt wurden, geht der letzte Abschnitt dieses Artikels der Frage nach, auf welche Weise ein Modul in eine Client-Anwendung integriert wird. Es wird erläutert, wie Ressourcen verwaltet und wie `Action`-Klassen verwendet werden, sowie wie die Menüleiste aufgebaut wird.

Verwaltung von Ressourcen

Ressourcen sind Konfigurationsdaten, die nicht unmittelbar für die Konfiguration von Modulen gebraucht werden. Dazu gehören Texte für Beschriftungen in Formularen und Menüs, Dateinamen von Symbolen, Tooltipps und Tastaturkürzel.

Ressourcen sind in Property-Dateien als Name/Wert-Paare hinterlegt. Über eine Punkt-Notation werden Namensräume gebildet, sodass Ressourcen sowohl Anwendung übergreifend als auch für einzelne Anwendungen definiert werden können. Jede Ressource wird über eine eindeutige Bezeichnung identifiziert. Der erste Teil der Bezeichnung ist der Name des Ressourceneintrags; darauf folgt die Bezeichnung des Typs der Ressource. Hier einige Beispiele:

```
# Übergreifend verwendete Ressourcen für "Kopieren"
copy.text=&Kopieren
copy.image=icons/copy.png
copy.key=control C
copy.tooltip=Kopiert Daten in die Zwischenablage
# Ressourcen für die Kundenverwaltung
customer.menu.text=&Kunde
customer.search.show.text=&Suchen
customer.search.show.key=control F
customer.search.show.image=icons/search.png
```

Der Ressourceneintrag „`copy`" enthält zum Beispiel vier Ressourcen: einen Text, einen Verweis auf eine Symbol-Datei, ein Tastaturkürzel und einen Tooltipp. Das Kaufmanns-Und (&) in den Texten stammt aus der Microsoft-Welt und kennzeichnet den Mnemonic, der im Text unterstrichen dargestellt wird. Der Zugriff auf die Ressourcen erfolgt über die Klassen `ResourceManager` und `ResourceItem` (s. Abb. 8).

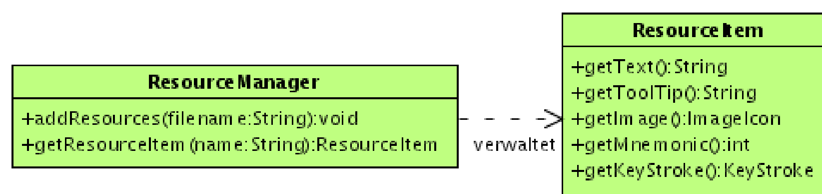

Abb. 8: Der Ressourcenverwalter

Verwendung von Action-Klassen

Die Menüeinträge und Schaltflächen einer Client-Anwendung werden mit `javax.swing.Action`-Objekten verbunden, die die Module erzeugen. Die Initialisierung eines `Action`-Objekts erfolgt über Ressourcen. Zum Framework gehört eine von `AbstractAction` abgeleitete Oberklasse, die die benötigten Informationen selbstständig aus einem Ressourceneintrag ausliest:

```
public abstract class GenuineAction extends javaw.swing.AbstractAction {

  public GenuineAction(String resourceName) {
    ResourceItem resourceItem =
      ResourceManager.getInstance().getResourceItem(resourceName);
    putValue(Action.NAME, resourceItem.getText());
    putValue(Action.SHORT_DESCRIPTION, resourceItem.getTooltip());
    putValue(Action.SMALL_ICON, resourceItem.getImage());
    putValue(Action.MNEMONIC_KEY, new Integer(resourceItem.getMnemonic()));
    putValue(Action.ACCELERATOR_KEY, resourceItem.getKeyStroke());
  }
}
```

Ein Beispiel zur Verwendung dieser `Action`-Klasse: Im Suchmodul der Kundenverwaltung wurde die Klasse `ShowComponentAction` verwendet. Ein Objekt dieser konfigurierbaren Klasse zeigt, wenn es aufgerufen wird, mit Hilfe des Komponentenverwalters eine grafische Komponente an. Es erhält dazu im Konstruktor den Namen der zu verwendenden Ressource sowie die eindeutige Bezeichnung der anzuzeigenden Komponente:

```
public class ShowComponentAction extends GenuineAction {

  private ComponentManager componentManager;
  private String moduleId;
  private String componentId;

  public ShowComponentAction(String resourceName,
                             ComponentManager componentManager,
                             String moduleId, String componentId) {
    super(resourceName);
    this.componentManager = componentManager;
    this.moduleId = moduleId;
    this.componentId = componentId;
  }

  public void actionPerformed(ActionEvent e) {
    componentManager.showComponent(moduleId, componentId);
  }
}
```

Aufbau der Menüleiste

Jedes Modul definiert in seinen Konfigurationsdaten seine Menüeinträge. Der Modulverwalter integriert diese Einträge in die Menüleiste während der Initialisierung der Module. Die Menüleiste wird zwar zu Beginn dynamisch aufgebaut; zur Laufzeit ist sie jedoch statisch.

Die Module stellen für jeden Menüeintrag ein passendes `Action`-Objekt bereit. Der Modulverwalter besorgt sich diese Objekte über die Methode `getAction` der Schnittstelle `Module` (s. Abb. 5), erzeugt damit ein `JMenuItem`-Objekt und hängt es an die gewünschte Stelle in der Menüleiste. Um die Zuordnung zwischen Ressourcen- und Menüeinträgen zu vereinfachen, stimmen die Bezeichnungen der Menüeinträge bzw. `Action`-Objekte mit den Bezeichnungen der Ressourceneinträge überein.

Jeder Menüeintrag kann auf einen anderen, schon zuvor angelegten Menüeintrag verweisen und vor oder hinter diesem Eintrag positioniert werden. Zusätzlich können Trennlinien eingefügt werden. Ein Beispiel: Ein Standardmodul definiert eine Grundstruktur für die Menüleiste und registriert für jeden Menüeintrag ein generisches `Action`-Objekt:

```
<menubar>
  <menu id="file.menu">
    <menuitem id="exit"/>
  </menu>
  <menu id="edit.menu">
    <menuitem id="cut"/>
    <menuitem id="copy"/>
    <menuitem id="paste"/>
  </menu>
  <menu id="help.menu">
    <menuitem id="info"/>
  </menu>
</menubar>
```

Der erste Eintrag, „`file.menu`“, bezeichnet das Datei-Menü. Da Menüs nicht mit `Action`-Objekten verbunden werden, greift das Framework für die Beschriftung direkt auf die Ressource „`file.menu.text`“ zurück. Dem Datei-Menü wird der Menüeintrag „`exit`“ hinzugefügt. Das nächste Menü, „`edit.menu`“, wird mit seinen Menüeinträgen hinter das Datei-Menü angefügt, usw.

Als nächstes wird das Suchmodul mit der folgenden Menüstruktur initialisiert:

```
<menubar>
  <menu id="customer.menu" behindof="edit.menu">
    <menuitem id="customer.search.show"/>
  </menu>
</menubar>
```

Das Kunden-Menü soll direkt hinter das Bearbeiten-Menü positioniert werden. Das Suchmodul integriert einen Menüeintrag, um den Suchdialog aufzurufen. Dafür hat es in seiner Initialisierungsmethode ein `ShowComponentAction`-Objekt unter der Bezeichnung „`customer.search.show`“ registriert (siehe Abschnitt zur Modul-Initialisierung).

Auch das (hier nicht näher vorgestellte) Modul für die Verwaltung der Geschäftsdaten fügt zwei Menüeinträge in die Menüleiste ein, nämlich zum Laden und Speichern des Datenbestands:

```
<menubar>
  <menu id="file.menu">
    <menuitem id="open" infrontof="exit"/>
    <menuitem id="save" infrontof="exit" separator="behind"/>
  </menu>
</menubar>
```

Dem schon bestehenden Datei-Menü werden zwei weitere Menüeinträge hinzugefügt, die vor dem Menü-Eintrag zum Beenden der Anwendung positioniert werden. Hinter dem Speichern-Eintrag soll zusätzlich eine Trennlinie erscheinen.

Abbildung 9 zeigt in einer Montage, wie die Menüleiste aussieht, nachdem alle Module ihre Einträge vorgenommen haben.

Abb. 9: Die Menüleiste der Kundenverwaltung

Fazit

Dieser Artikel hat einen weiten Bogen gespannt: von grundsätzlichen Eigenschaften, die ein Client-Framework aufweisen sollte, bis hin zu konkreten Details eines Beispiel-Frameworks. Im Fokus stand dabei, Konzepte und Ideen zu erläutern, die sich in Industrieprojekten bewährt haben.

Ein Framework prägt maßgeblich die Architektur einer Client-Anwendung. Es muss die Struktur einer Anwendung vorgeben, darf aber die Möglichkeiten der Anwendung nicht zu sehr einschränken. Dieser Artikel hat vorgeführt, wie mit den Mitteln der Entkopplung und Konfiguration modulare Client-Anwendungen erstellt werden können, die robust sind und auch in groß angelegten Projekten wartbar bleiben.

Vergleich mit anderen Frameworks und Produkten

Unter den verfügbaren Frameworks ragen zwei Plattformen hervor: die Eclipse Rich Client Platform [2] und die NetBeans Platform [3]. Die Eclipse Rich Client Platform (RCP) bildet die Grundlage für alle Eclipse Tools wie beispielsweise die Eclipse Java IDE. Jede Anwendung wird als Sammlung von Plugins programmiert, die über XML-Daten gekoppelt werden. Obwohl sich damit große Anwendungen realisieren lassen, weist die RCP zwei Nachteile auf: Zum einen lassen sich Plugins beliebig tief ineinander verschachteln, wodurch der Komplexitätsgrad einer Anwendung steigt. Zum anderen unterstützt die RCP nur das Eclipse eigene Standard Widget Toolkit (SWT).

Die NetBeans Platform basiert auf Swing und bildet die Grundlage der NetBeans IDE für Java. Die Funktionalität einer Client-Anwendung wird, ähnlich wie in diesem Artikel beschrieben, auf Module verteilt, die über Konfigurationsdaten zur Laufzeit initialisiert werden. Im Gegensatz zu Eclipse wird die NetBeans Plattform nur von wenigen Client-Anwendungen verwendet. Ihre API ist zudem an manchen Stellen historisch gewachsen.

Viele Produkten verfolgen den Weg, Client-Anwendungen durch eine Beschreibung in XML zu realisieren. Im Java-Umfeld gehören dazu beispielsweise die JDesktop Network Components von Sun ([4]), Luxor ([5]) und die Swing Markup Language ([6]). Auch Microsoft entwickelt derzeit eine Bibliothek mit ähnlicher Zielsetzung: Avalon ([7]).

Diese Bibliotheken haben zum Ziel, vollständige Client-Anwendungen in XML zu modellieren. Einfache und kleinere Anwendungen lassen sich mit diesen Produkten schnell realisieren. Trotzdem bleibt Java-Programmcode immer notwendig, da kein Produkt alle möglichen Anwendungsfälle abdecken kann. Eine der Folgen ist, dass die Mischung von XML und Java unübersichtlich wird. Zudem bieten die wenigsten der vorhandenen Produkte Möglichkeiten, komplexe Anwendungen sinnvoll zu strukturieren.

Weiterhin existieren Bibliotheken, die die Framework-Entwicklung vereinfachen. Dazu gehören zum Beispiel die JGoodies Swing Suite ([8]) und das JIDE Docking Framework ([9]). Eine umfangreiche Referenz auf Projekte für die Client-Entwicklung mit Swing befindet sich auf der Seite JavaDesktop von Sun [10].

Referenzen

[1] Genuine (Genuine User Interface): http://genuine.sourceforge.net

[2] Eclipse Rich Client Platform: http://www.eclipse.org

[3] NetBeans Platform: http://www.netbeans.org/products/platform/

[4] JDesktop Network Components (JDNC): http://jdnc.dev.java.net/

[5] Luxor - XML UI Language (XUL) Toolkit: http://luxor-xul.sourceforge.net/

[6] Swing Markup Language: http://swingml.sourceforge.net/

[7] Avalon: http://msdn.microsoft.com/msdnmag/issues/04/01/Avalon/default.aspx

[8] JGoodies Swing Suite: http://www.jgoodies.com/products/swingsuite.html

[9] JIDE Docking Framework: http://www.jidesoft.com/products/dock.htm

[10] JavaDesktop: http://community.java.net/javadesktop/