8

Das Erstellen des Displayfile-Objektes

Dieses Kapitel beginnt das Studium des **com.pbd.pub.jdspf**-Paketes. Da es von Notwendigkeit ist, ist dieses Paket wesentlich größer, komplexer und weniger vollständig als die anderen, bisher besprochenen. Die Grenzen dieses Buches gestatten es mir nicht, tiefer in die Architektur einzutauchen, als die einfachste aller Operationen: die EXFMT. Wenn Sie sich diesen Code jedoch anschauen, werden Sie sehen, dass die Grundlage da ist, um jede der anderen Eigenschaften hinzuzufügen, die benötigt werden, um einen voll funktionsfähigen Displayfile-Emulator zu erzeugen.

In diesem Kapitel werden Sie lernen, Ihr erstes wirkliches Revitalization-Objekt zu erzeugen – nämlich das Objekt **ITMMNT1D_DisplayFile**. Dieses Kapitel ist in die folgenden Abschnitte gegliedert:

- „Das Definieren einer **JdspfDisplayFile**: **ITMMNT1D_DisplayFile**“ ist Teil des schnellen Pfades (wie in der Einführung zu Teil 2 dieses Buches definiert). Es dient dazu, Ihnen zu ermöglichen, schnell ein **JdspfDisplayFile**-Objekt zu definieren, um das **ITMMNT1D**-Displayfile zu emulieren. Sie werden sehen, wie einfach es ist, das Displayfile-Objekt zu definieren, ohne ein Java-Experte zu sein.

- „Die JDSPF-Classes“ geht wesentlich detaillierter auf das JDSPF-Paket hinein. Sie werden sehen, wie es beschaffen ist, um eine Brücke zwischen dem alten Prozeduralmodell und der mutigen neuen Welt der Ereignisse und Listener darzustellen und wie dieser Gedanke in dem **JdspfDisplayFile**-Objekt eingesetzt ist. Diese Brücke ist es, was sie ausnutzen werden, wenn Sie JDSPF als ein gemeinsames Verbindungsrohr sowohl zu dem Swing-Client als auch in späteren Abschnitten zu dem Browserclient verwenden.

- „Die Zukunft von JDSPF“ untersucht einige der beträchtlichen Möglichkeiten, die noch existieren. Zur Verfassungszeit unterstützt das JDSPF-Paket nur EXFMT und keine Display-Attribute, obwohl die Infrastruktur für wesentlich mehr existiert. Dieser Abschnitt sieht sich an, was noch hinzugefügt werden muss und wie es getan werden kann.

Das Definieren einer JDSPFDISPLAYFILE: ITMMNT1D_DISPLAYFILE

Um ein **ITMMNT1D_DisplayFile**-Objekt zu definieren müssen Sie etwas über zwei Sätze von Classes lernen: die **JdspfAbstractRecord-**Class und ihre Subclasses sowie die **JdspfDisplayFile**-Class. **JdspfAbstractRecord** ist eine abstrakte Class, die die grundlegende Arbeitsweise eines Displayfile-Datensatzformates definiert. Da die ganze Programm-I/O auf der AS/400 auf der Datensatzformatebene durchgeführt wird, ist **JdspfAbstractRecord** geschaffen, um die verschiedenen I/O-Operationen zu unterstützen.

Gleichzeitig kann die wirkliche Benutzerschnittstelle multiple Datensätze auf einmal anzeigen. Ein perfektes Beispiel ist ein Subfile und sein Subfile-Kontrolldatensatz. Beide werden gleichzeitig angezeigt; tatsächlich haben in vielen Fällen beide eingabefähige Felder. An einem bestimmten Punkt müssen diese Datensätze in ein einziges Objekt kombiniert werden, auf das von dem UI-Server zugegriffen werden kann. Die **JdspfDisplayFile**-Class führt diese Funktion durch. Es ist nun an der Zeit, ein **JdspfDisplayFile** zu erzeugen, besonders das **ITMMNT1_DisplayFile**.

Sehen wir uns zuerst die Auflistungen 8.1 und 8.2 an: **JdspfAbstractRecord** und die einzige derzeitig unterstützte konkrete Subclass, **JdspfRecord.**

```
Class com.pbd.pub.jdspf.JdspfAbstractRecord
   public abstract class JdspfAbstractRecord

   This is the abstract superclass for all JDSPF record classes.

Constructors
   JdspfAbstractRecord
    public JdspfAbstractRecord(String name,
                    Dc400Structure structure)
```

*Auflistung 8.1: Der Konstruktor für die **JdspfAbstractRecord**-Class*

JdspfAbstractRecord ist die Basic-Class für alle anderen Datensatz-Classes. Sie wird später in diesem Kapitel im Detail betrachtet; für den Moment genügt es, zu wissen, dass die Class einen Datensatznamen und eine **Dc400Structure** erfordert.

Die **Dc400Structure** muss die Daten in der extern beschriebenen Datenstruktur, die diesem Datensatz zugeordnet ist, beschreiben. Für das PROMPT-Datensatzformat zum Beispiel benötigen Sie eine Struktur, die die **ITMMNT1DR1** Datenstruktur beschreibt[1] . Alle Datensatztypen stammen von diesem Basistypen ab. Tatsächlich hat der JdspfRecord, der in Auflistung 8.2 gezeigt ist und den AS/400-Displayfile-Datensatzformattypen RECORD emuliert, genau denselben Konstruktor.

[1] Zur Verfassungszeit dieses Buches gibt es eigentlich keine Möglichkeit, ein Datensatzformat ohne Datenfelder zu erzeugen, nur mit Befehlstasten. Dies ist jedoch eine sehr triviale Änderung und verbleibt Ihnen als Übung.

Class com.pbd.pub.jdspf.JdspfRecord

public class JdspfRecord
extends JdspfAbstractRecord

This class corresponds to DDS type RECORD.

Constructor
JdspfRecord
public JdspfRecord(String name,
Dc400Structure structure)

*Auflistung 8.2: Der Konstruktor für die **JdspfRecord**-Class*

Das Erzeugen eines Datensatzes ist sehr einfach, wenn Sie einmal mit dem Erzeugen einer **Dc400Structure** vertraut sind. Auflistung 8.3 zeigt zwei geringfügig unterschiedliche Möglichkeiten zum Erzeugen eines Auslösedatensatzes. Die erste verwendet die Strukturvariable, die in Kapitel 7 als eine Zwischenvariable erzeugt wurde, während die zweite alle benötigten Objekte in einer Erklärung erzeugt. Alle Datensätze werden auf dieselbe grundlegende Weise erzeugt, mit leichten Unterschieden: ein Subfile-Steuerdatensatz muss auf seinen entsprechenden Subfile-Datensatz zeigen. Der allgemeinen Vorstellung, dass ein Datensatz einen Namen und eine Struktur hat, wird in der Revitalization-Architektur konsistent Rechnung getragen.

```
Creating the JdspfRecord, Style 1
    Dc400Structure prompt =
        new Dc400Structure(
            new Dc400Field[] {
                new Dc400CharacterField("X1ITEM", 15),
                new Dc400CharacterField("X1ERR", 70)
```

*Auflistung 8.3: Zwei alternative Stile für das Erzeugen eines **JdspfRecord-**Objektes (Teil 1 von 2)*

```
            }
        );

    JdspfRecord promptRecord = new JdspfRecord("PROMPT  ", prompt);


Creating the JdspfRecord, Style 2
    JdspfRecord promptRecord =
        new JdspfRecord(
            "PROMPT  ",
            new Dc400Structure(
                new Dc400Field[] {
                    new Dc400CharacterField("X1ITEM", 15),
                    new Dc400CharacterField("X1ERR", 70)
                }
            )
        );
```

*Auflistung 8.3: Zwei alternative Stile für das Erzeugen eines **JdspfRecord**-Objektes (Teil 2 von 2)*

Bis hierhin erzeugten Sie eine Variable, immer wenn eine bestimmte Version eines allgemeineren Objektes benötigt wurde. Beim Erzeugen des **JdspfDisplayFile**-Objektes werden Sie zum ersten Mal Vererbung verwenden, um etwas spezieller zu machen. Wenn Sie fertig sind, haben Sie Ihre erste „funktionierende" Classes-Definition, die **ITMMNT1D_DisplayFile**-Class, mit allem, was erforderlich ist zum Emulieren des **ITMMNT1D**-Displayfile auf der AS/400.

Untersuchen Sie zu Anfang die **JdspfDisplayFile**-Class und ihre Konstruktoren in Auflistung 8.4.

Class com.pbd.pub.jdspf.JdspfDisplayFile
public class JdspfDisplayFile

The JdspfDisplayFile is used to emulate an AS/400 display file. A JdspfDisplayFile is identified by a name, and either a JdspfRecord or an array of JdspfAbstractRecords.

Constructors

JdspfDisplayFile
public JdspfDisplayFile(String name,
JdspfAbstractRecord records[])

JdspfDisplayFile
public JdspfDisplayFile(String name,
JdspfRecord record)

*Auflistung 8.4: Die **JdspfDisplayFile**-Class und ihre Konstruktoren*

Eine JdspfDisplayFile ist eigentlich ein relativ simples Konstrukt. Es ist einfach ein Name und eines oder mehrere **JdspfAbstractRecord**-Objekte. Es gibt zwei Konstruktoren: Einer akzeptiert eine Feldgruppe von **JdspfAbstractRecords** und einer akzeptiert einen einzelnen **JdspfRecord.**

Sie werden sich vielleicht fragen, warum der zweite Konstruktor nur ein **JdspfRecord**-Objekt anstatt eines **JdspfAbstractRecord** akzeptiert. Der Grund dafür ist, dass Sie mindestens einen Datensatz haben müssen, der das EXFMT-Verfahren unterstützt und nur **JdspfRecord** und seine Nachkommen unterstützen EXFMT.

Deshalb können Sie zum Erzeugen einer **JdspfDisplayFile** einige Variablen, die die Nachkommen eines **JdspfAbstractRecord** enthalten, vordefinieren oder Sie können Sie heimlich mit einem verschachtelten Konstruktor – ähnlich dem zum Erzeugen des **JdspfRecord**-Auslösers an früherer Stelle in diesem Kapitel verwendeten – erzeugen. Auflistung 8.5 zeigt, wie man ein Displayfile erzeugen kann, die nur das Einzel-Auslöser-Datensatzformat unterstützt, unter Verwendung von Zwischenvariablen.

```
Dc400Structure prompt =
    new Dc400Structure(
        new Dc400Field[] {
            new Dc400CharacterField("X1ITEM", 15),
            new Dc400CharacterField("X1ERR", 70)
        }
    );

JdspfRecord promptRecord = new JdspfRecord("PROMPT  ", prompt);

JdspfDisplayFile promptOnlyDisplayFile =
    new JdspfDisplayFile("PROMPTER", promptRecord);
```

Auflistung 8.5: Wie man Zwischenvariablen verwendet zum Erzeugen eines Einzel-Auslöser-Datensatz-Displayfile

Weil alle Classes, die zum Bauen einer **JdspfDisplayFile** erforderlich sind, unmittelbare Konstruktoren (Konstruktoren, die eine Feldgruppe von Objekten akzeptieren) erlauben, ist es ziemlich einfach, das gesamte Displayfile als einen Einzelaufruf an den Superclass-Konstruktor zu definieren. Und wenn es etwas komplex wird, so werden Sie bei genauerem Hinsehen bemerken, wie gut es das Original-Displayfile DDS widerspiegelt. Mögliche wirkliche Felder (im Gegensatz zu Konstanten) werden als **Dc400Fields** innerhalb eines Datensatzes definiert und jeder Datensatz wird als ein **JdspfRecord** definiert. Die Feldgruppe von Datensätzen wird an den Superclass-Konstruktor zusammen mit dem Displayfile-Namen überbracht, was in einer konkreten Subclass spezifisch für das **ITMMNT1D**-Displayfile resultiert. Auflistung 8.6 zeigt, wie`s gemacht wird.

```
import com.pbd.pub.dc400.*;
import com.pbd.pub.jdspf.*;

/**
 * This is the emulation object for the ITMMNT1D display file.
 * This object supports the PROMPT and MAINT records.
```

*Auflistung 8.6: Wie man Subclasses für die **JdspfDisplayFile**-Class schafft, um ein spezifisches Displayfile zu erstellen, indem man einen Aufruf an den Superclass-Konstruktor mit einem verschachtelten Konstruktor verwendet (Teil 1 von 2)*

```
 */
public class ITMMNT1D_DisplayFile extends JdspfDisplayFile {
/**
 * ITMMNT1D_DisplayFile constructor comment.
 * @param name java.lang.String
 * @param records com.pbd.pub.jdspf.JdspfAbstractRecord[]
 */
public ITMMNT1D_DisplayFile() {

     // Call superclass constructor
    super(

        // Display file name ITMMNT1D
        "ITMMNT1D",

        // Array of record formats
        new JdspfAbstractRecord[] {

            // PROMPT record format
            new JdspfRecord(
                "PROMPT    ",
                new Dc400Structure(
                    new Dc400Field[] {
                        new Dc400CharacterField("X1ITEM", 15),
                        new Dc400CharacterField("X1ERR", 70)
                    }
                )
            ),

            // MAINT record format
            new JdspfRecord(
                "MAINT     ",
                new Dc400Structure(
                    new Dc400Field[] {
                        new Dc400CharacterField("X2MODE", 10),
                        new Dc400CharacterField("X2ITEM", 15),
                        new Dc400CharacterField("X2DESC", 30),
                        new Dc400NumericField("X2QTYO", 9, 2),
                        new Dc400CharacterField("X2UOMI", 2),
                        new Dc400CharacterField("X2UOMS", 2),
                        new Dc400CharacterField("X2ERR", 70)
                    }
                )
            )

        }
    );
}
}
```

Auflistung 8.6: (Teil 2 von 2)

Datensatz-UI-Objekte

Das letzte Stück des Puzzles ist die **JdspfRecordUI**-Class, von der eine Feldgruppe an das Displayfile unter Verwendung des **setRecordUIObjects**-Verfahrens überbracht wird. Dies ist ein anderes Gebiet, bei dem ich ein wenig Schwierigkeiten hatte, ein bestimmtes Stückchen Code zu platzieren. Bis hierhin war die Definition des Displayfile vollkommen unabhängig von dem Typen der Benutzerschnittstelle, die verwendet wird. Im Laufe des Designs jedoch ist die Annahme die gewesen, dass an irgendeiner Stelle irgendein wirkliches UI-Objekt mit jedem Datensatzformat verbunden werden müsste oder vielleicht mit jeder Gruppe von verwandten Datensatzformaten.

Weil ich diesen Teil der Architektur noch nicht vollständig unter Dach und Fach gebracht habe, habe ich die Datensatz-UI-Objekte nicht als Teil des Konstruktors eingeschlossen. Wie auch immer ich mich entscheide, dies anzugehen, wird die sehr einfache **JdspfRecordUI**-Class zum Kommunizieren zwischen dem Displayfile und dem UI-Server verwendet werden. Die **JdspfRecordUI**-Class bestimmt einfach den Datensatzformatnamen und ein Objekt irgendeiner Class. Der Typ des Objektes ist dem UI-Server bekannt, aber nicht dem Displayfile. Das **setRecordUIObjects**-Verfahren verbindet die Objekte mit ihren bestimmten Datensatzformaten. Es speichert das UI-Objekt in dem Datensatzformatobjekt. Der UI-Server erhält das Datensatzformatobjekt entweder von dem **JdspfDisplayFileAction**-Ereignis oder durch Abfragen der **JdspfDisplayFile**. Egal wie, der UI-Server kann dann das Datensatz-UI-Objekt abrufen und die notwendigen Schritte zum Anzeigen der Daten an den Benutzer durchführen.

Datensatz-UI-Objekte werden detaillierter in den Kapiteln 10 und 11 beleuchtet, in denen das eigentliche Erzeugen von UI-Servern besprochen wird. Ich wollte das Konzept hier ansprechen, weil die Datensatz-UI-Objekte vielleicht Teil eines **JdspfDisplayFile-**Konstruktors in späteren Veröffentlichungen der Software sein werden.

Die JDSPF-Classes

Das Hauptkonzept hinter den JDSPF-Classes ist das möglichst direkte Emulieren des AS/400-Displayfile, was ziemlich unkompliziert erscheint. Zuerst dachte ich, ich müsste nur die Datensatzformate definieren und sie in einem Displayfile kombinieren und dann den Zugriff durch Verfahren, die die verschiedenen I/O-Operationen nachahmen, ermöglichen, wie zum Beispiel WRITE und EXFMT. Als ich begann, mir das Design etwas näher anzusehen, tauchten ein paar Diskrepanzen auf. Sie werden sehen, wie ein paar von ihnen hier angesprochen werden und weitere in Kapitel 10.

Der erste Designpunkt beinhaltete die verschiedenen DDS-Datensatztypen. Ich machte mir zwar keine ernsten Sorgen über den Unterschied zwischen einem Popup-Fenster und einem normalen Display-Menü, aber ich sorgte mich sehr wohl, was den Subfile-Datensatz anging. Während ein normaler Datensatz einen einzelnen Puffer oder Daten enthält, enthält der Subfile-Datensatz tatsächlich eine ganze Feldgruppe von Puffern. Ich erkannte, dass dies zwei verschiedene Classes erfordern würde, eine, die einen Einzelpuffer unterstützte und eine, die multiple Puffer unterstützte.

Außerdem wurde es offensichtlich beim Betrachten der UPDAT-Operation für das Subfile, dass es sogar das Konzept eines „laufenden“ Puffers innerhalb des Subfile gab. (Im Falle der UPDAT-Operation ist dies der letzte über CHAIN, READ oder READC abgerufene Datensatz.)

Gleichzeitig musste ich daran denken, wie die UI-Server die Daten aus dem Datensatz bekommen würden. Während ich zwar eine Menge Kontrolle über den Swing-UI-Server hatte und mir leicht eine Adapter-Class vorstellen konnte, die einen Vektor von Datenpuffern in ein Tabellenmodell umwandeln konnte, war es nicht so einfach, festzulegen, wie der Browser-UI-Server dies erledigen würde.[2] Ein wichtiges Ziel war es, die Schnittstelle so klar wie möglich zu halten. Ich erkannte bald, dass ich schon vorher auf das Konzept gestoßen war: der „laufende“ Puffer.

[2] Sie haben an dieser Stelle noch nicht die JSP-Definitionen gesehen, die erforderlich sind zum Erzeugen der Browserschnittstelle. Wenn Sie einen kurzen Blick darauf werfen möchten, dann überfliegen Sie Kapitel 11, um die **ITMMNT1**-Durchführung zu sehen.

Diese Puffer-Class, in Auflistung 8.7 gezeigt, ist die erste in diesem Buch, die eine andere Class erweitert. Als ich sie mir zuerst ansah, dachte ich, dass ich die **Dc400Structure-**Class vielleicht direkt verwenden würde. Aber es wurde offensichtlich, dass die JdspfBuffer etwas mehr benötigte. Wie das Display-Attributefeld in der **Dc400Field-**Class ist das Indikatorfeld nicht in dieser Veröffentlichung verwendet, aber es ist als Aufhänger für die nächste Veröffentlichung vorhanden. Ich ließ das Indikatorfeld vor Ort als eine Erinnerung daran, dass die **JdspfBuffer** schließlich etwas mehr Intelligenz benötigen wird, als die einfache Dc400Structure.

Class com.pbd.pub.jdspf.JdspfBuffer

public class JdspfBuffer
extends Dc400Structure

The JdspfBuffer class extends the Dc400Structure to include indicators and a "changed" flag.

Constructors

JdspfBuffer

public JdspfBuffer()

JdspfBuffer

public JdspfBuffer(Dc400Structure structure)

*Auflistung 8.7: Die Konstruktoren und Verfahren, die für die **JdspfBuffer**-Class spezifisch sind, die die **Dc400Structure-**Class erweitert (Teil 1 von 2)*

JdspfBuffer

public JdspfBuffer(Dc400Structure structure,
byte buffer[],
byte indicators[])

JdspfBuffer

public JdspfBuffer(JdspfAbstractRecord record,
byte buffer[],
byte indicators[])

Methods

getIndicators

public JdspfIndicatorArray getIndicators()

isChanged

public boolean isChanged()

setIndicators

public void setIndicators(byte indicators[])

setIndicators

public void setIndicators(JdspfIndicatorArray newValue)

*Auflistung 8.7: Die Konstruktoren und Verfahren, die für die **JdspfBuffer-**Class spezifisch sind, die die **Dc400Structure**-Class erweitert (Teil 2 von 2)*

JdspfBuffer erweitert die **Dc400Structure**-Class. Dies ist ein weiteres Gebiet, wo eine schwierige Design-Entscheidung getroffen werden musste, dazwischen, ob ich die **JdspfBuffer** zu einem Aggregat machen sollte, das einfach eine **Dc400Structure** als eine Variable enthält oder ob ich es zu einer direkten Subclass der **Dc400Structure** machen sollte. Letztendlich entschied ich mich für die Subclass-Option, weil ich Felder so effizient in eine **JdspfBuffer** hinein und hinaus bekommen musste, wie ich sie

aus einer **Dc400Structure** herausbekommen musste. Da diese Verfahren für die **Dc400Structure** bereits existierten, machte es Sinn, sie zu vererben, anstatt sie delegieren zu müssen, was im Grunde genommen das Kopieren der Signatur jedes Verfahrens erfordert hätte und dann das Verschieben durch die enthaltene Class. DA **Dc400Structure** eine ziemlich massive API hat, entschied ich mich, sie zu vererben.

Die zusätzlichen Eigenschaften der **JdspfBuffer** werden hier nicht verwendet, deshalb können Sie sich diese wie eine **Dc400Structure** vorstellen: im Grunde genommen ein Puffer, der eine Menge verschiedener Felder enthält, auf die Sie über Namen zugreifen können. Mit dem im Hinterkopf fahren wir nun mit den anderen Classes fort, beginnend mit der **JdspfAbstractRecord** und ihren Nachfahren, wie in Auflistung 8.8 gezeigt.

Class com.pbd.pub.jdspf.JdspfAbstractRecord
public abstract class JdspfAbstractRecord

This is the abstract superclass for all JdspfDisplayFile records.

Constructors

JdspfAbstractRecord
public JdspfAbstractRecord(String name,
Dc400Structure structure)

Methods

CHAIN
public boolean CHAIN(JdspfBuffer buffer) throws JdspfInvalidOperationException

EXFMT
public boolean EXFMT(JdspfBuffer buffer) throws JdspfInvalidOperationException

getCurrentBuffer
protected abstract JdspfBuffer getCurrentBuffer()

getString
public String getString(String name)

getUIObject
public Object getUIObject()

READ
public boolean READ(JdspfBuffer buffer) throws JdspfInvalidOperationException

READC
public boolean READC(JdspfBuffer buffer) throws JdspfInvalidOperationException

setObject
public void setObject(String name,
Object newValue)

*Auflistung 8.8: Der Konstruktor und ausgewählte Verfahren der **JdspfAbstractRecord**-Class (Teil 1 von 2)*

```
UPDAT
public boolean UPDAT(JdspfBuffer buffer) throws JdspfInvalidOperationException

WRITE
public boolean WRITE(JdspfBuffer buffer) throws JdspfInvalidOperationException
```

*Auflistung 8.8: Der Konstruktor und ausgewählte Verfahren der **JdspfAbstractRecord**-Class (Teil 2 von 2)*

Die derzeitige Funktion des JDSPF-Paketes ist ziemlich eingeschränkt, weil ich einige der Einzelheiten der Bearbeitung multipler verwandter Datensätze noch nicht entschieden habe. Der Rest dieses Abschnitts zeigt Ihnen, wie das Paket derzeitig funktioniert, was ausreichend ist für das einfache Schnell-Pfad-Beispiel. Die übrigen Abschnitte dieses Kapitels gehen viel tiefer ins Detail über die Art und Weise, wie dieses Gebiet funktionieren wird in wesentlich komplexeren Displayfiles.

Heute hat der **JdspfAbstractRecord** fast alle Eigenschaften, die zum Unterstützen von Datensatz-I/O-Operationen erforderlich sind. Es gibt in der ganzen Class nur ein abstraktes Verfahren, welches das Verfahren **getCurrentBuffer** ist. Das Verfahren **getString** in dem **JdspfAbstractRecord** holt die Zeichenfolgerepräsentation eines Feldes aus dem laufenden Puffer, aber es obliegt der Subclass, **JdspfAbstractRecord** mitzuteilen, welcher Puffer der laufende Puffer ist. Dies wird erreicht, indem das abstrakte **getCurrentBuffer**-Verfahren außer Kraft gesetzt wird. Auf diese Weise kann eine Einzelpuffer-Class wie die **JdspfRecord**, gezeigt in Auflistung 8.9, einfach seinen einen und einzigen Puffer ausgeben, während die Subfile-Emulations-Classes mit dem

„laufenden“Puffer Schritt halten können und ihn stattdessen ausgeben.

Class com.pbd.pub.jdspf.JdspfRecord

public class JdspfRecord
extends JdspfAbstractRecord

This class corresponds to DDS type RECORD.

Constructor

JdspfRecord
public JdspfRecord(String name,
Dc400Structure structure)

Methods

*Auflistung 8.9: Der Konstruktor und die Verfahren für die **JdspfRecord**-Class (Teil 1 von 2)*

READ

public boolean READ(JdspfBuffer ioBuffer)

WRITE

public boolean WRITE(JdspfBuffer buffer)

*Auflistung 8.9: Der Konstruktor und die Verfahren für die **JdspfRecord**-Class (Teil 2 von 2)*

Viel wichtiger für die **ITMMNT1-**Durchführung ist, dass die **JdspfAbstractRecord,** während sie zwar alle I/O-Operationen durchführt, sie jedoch als ein Stummelverfahren durchführt, das nicht anderes macht, als eine **JdspfInvalidOperationException**-

Message auszuwerfen. Es ist den Subclasses überlassen, die entsprechenden Verfahren durchzuführen. Die **JdspfRecord** führt zum Beispiel die READ und WRITE-Verfahren durch; eine Subfile-Emulations-Class würde auch CHAIN, READC und UPDAT durchführen.

Schließlich gibt es auch noch das Problem, eine Schnittstelle für den UI-Server tatsächlich zu schaffen. Zum größten Teil ist das **ScJdspfJbuiDisplay-**Objekt ein Aufbewahrungsgebiet, das sich nicht besonders um die tatsächliche Mechanik des Display-Prozesses kümmert. An irgendeinem Punkt müssen die Daten jedoch zum Benutzer gelangen. An dieser Stelle kommt das **UIObject** ins Spiel. Jeder Datensatz soll ein **UIObject** haben, das mit ihm verbunden ist. Die Art von **UIObject** und was es tut, hängt von dem UI-Server ab. Im Falle des Swing-UI-Servers ist das **UIObject** eine Version von **ScJdspfJbuiDisplay**, welches eine Class ist, die das Anzeigen von Feldern auf ähnliche Weise wie bei einem 5250-Display ermöglicht, nur mit den schönen graphischen Eingabefeldern und den Knöpfen, die die Benutzer mittlerweile erwarten. Beim Browser-UI-Server ist das **UIObject** ein **ScJspfJsp-**Objekt, das eine JavaServer Page (JSP) repräsentiert. Die JSP wird ausgelöst, wenn Benutzerinteraktion erforderlich ist.

Innerhalb des **JdspfDisplayFile**-Objektes gibt es eine weitere Verfeinerung, die mit der Benutzerinteraktion zu tun hat. Das Problem umgeht die Reihenfolge der Ereignisse. Bei einem 5250-Display wird dem Benutzer der Bildschirm präsentiert und das Programm unterbrochen, bis der Benutzer eine Befehlstaste drückt. Dies kann „nett“ mit einer **JbuiDisplay** emuliert werden. Tatsächlich ist es eines der fundamentalen Designkonzepte, das hinter der Class steht. Es ist jedoch nicht so einfach mit einer Browserschnittstelle zu

erledigen. Die Daten werden an den Benutzer geschickt und dann bremst das Servlet im Grunde genommen die Ausführung. Sie beginnt nicht wieder, bis der Benutzer einen Knopf auf einem Formular anklickt, welches dann über einen Aufruf an das **doPost**-Verfahren an das Servlet gesendet wird. (Tatsächlich könnte das Formular an ein komplett anderes Servlet gehen, obwohl dies nicht das ist, wofür das **ITMMNT1_Servlet** geschaffen ist.)

Ich hatte eine Reihe langer Nächte, in der ich versuchte, einen Weg aus diesem schlimmen kleinen Paradox zu finden. Schließlich stieß ich auf eine Lösung, die, wenn zwar auch nicht gerade elegant, so doch funktional ist. Um sie zu verstehen, beginnen Sie, indem Sie eines der internen Verfahren von **JdspfDisplayFile,** nämlich **putGetUI,** betrachten. Es ist in Auflistung 8.10 zu sehen.

```
public boolean putGetUI(JdspfBuffer buffer)
    throws JdspfInvalidOperationException
{
    currentRecord.WRITE(buffer);

    boolean breakRequired = true;

    if (listener != null) {
        JdspfDisplayFileAction action =
            new JdspfDisplayFileAction(
                this,
                JdspfDisplayFileAction.JDSPF_EXFMT,
                currentRecord);
        breakRequired = listener.actionPerformed(action);
    }

    if (!breakRequired)
        currentRecord.READ(buffer);

    return breakRequired;
}
```

*Auflistung 8.10: Das **putGetUI-**Verfahren der **JdspfDisplayFile***

Das putGetUI-Verfahren wird ausgelöst, wenn ein EXFMT empfangen wird. Auf der Datensatzebene wird es eine WRITE- und dann eine READ-Operation durchführen, aber irgendwie muss das eigentliche **UIObject** ausgelöst werden.
Sie können dies machen, indem Sie das Konzept des Listener-Objektes verwenden. Der Listener wird ausgelöst, wenn eine Interaktion mit dem Benutzer erforderlich ist:

1. Das WRITE-Verfahren wird für das Datensatzobjekt aufgerufen. Dies schaltet das Datensatzformat auf den laufenden Puffer von dem UI-Server (welcher wiederum es von dem Anwendungsclient empfangen hat).

2. Der Default des **breakRequired**-Signals wird auf wahr geschaltet.

3. Wenn es keinen Listener gibt, hören Sie mit wahr auf, wodurch Sie praktisch dem UI-Server mitteilen, dass er sicherstellen soll, dass die Benutzerinteraktion stattfindet. So funktioniert der Browser-UI-Server. Er setzt dann ein Signal, zeigt die JSP und wartet, dass sein **doPost**-Verfahren aufgerufen wird.

4. Andernfalls führen Sie das **actionPerformed-**Verfahren des Listener durch und überbringen ihm den laufenden Datensatz. Derzeitig funktioniert so der Swing-UI-Server. Er holt des entsprechende **JbuiDisplay**-Objekt von dem **getUIObject**-Verfahren des Datensatzes, aktualisiert es mit den Pufferdaten und führt dann sein EXFMT-Verfahren aus. Wenn der Benutzer die Daten eingibt und einen Knopf anklickt, überbringt der UI-Server die Daten von der **ScJdspfJbuiDisplay** zurück in den

Puffer und gibt falsch aus, was der **JdspfDisplayFile** mitteilt, dass sie die Daten aus dem Datensatz abrufen soll. Die **JdspfDisplayFile** teilt wiederum dem UI-Server mit, die Daten an den Anwendungsclient zurückzuschicken.

Sie werden diese Architektur etwas detaillierter in den Einsatzabschnitten der Swing- und Browserclient-Classes in den Kapiteln 10, und entsprechend, 11 sehen und in dem Einsatzabschnitt der **ScJdspfUIServer**-Class in Kapitel 12.

Die Zukunft von JDSPF

Wie schon vorher erwähnt unterstützt diese Veröffentlichung der Software keine Display-Attribute oder Subfiles, aber die Aufhänger sind vor Ort, um Ihnen eine Vorstellung davon zu geben, wie sie durchgeführt werden. Dieser Abschnitt liefert ein wenig Anleitungshilfe, wenn Sie sich entschließen sollten, sich selbst beim Hinzufügen dieser Eigenschaften zu versuchen. Auf diese Weise haben Sie eine vernünftige Chance, mit der nächsten Veröffentlichung kompatibel zu sein, wenn Sie erscheint.

Subfiles

Sehen wir uns an, was durch das Hinzufügen der Subfile-Verarbeitung zu den Paketen erreicht werden soll. Auflistung 8.11 zeigt einen kleinen Ausschnitt aus der JSP, die zum Emulieren eines Subfile verwendet werden sollte. Obwohl die **setRecord-** und **getNextRow**-Verfahren nicht in dem Schnell-Pfad-Beispiel an

früherer Stelle in diesem Kapitel verwendet wurden, ist es angemessen, sie hier kurz anzusprechen. Hoffentlich können Sie auch ohne viel Hintergrundwissen von JSP eine Einsicht darin bekommen, was dieses Codebruchstück tut.[3] Zuerst setzt es den Datensatz, der in der **JdspfDisplayFile** (in der Variablen **jdspf)** zu verwenden ist, auf **SFLRCD01**, was das Subfile-Datensatzformat ist. Als nächstes durchläuft es den Zyklus des Subfile unter Verwendung des **getNextRow-**Verfahrens. Wie das **next-**Verfahren in einem SQL-Ergebnissatz gibt **getNextRow** wahr aus, wenn es noch eine Reihe gibt und falsch, wenn es keine mehr gibt. Dann ruft es für jedes Feld in dem Subfile-Datensatz das **getString**-Verfahren auf, um jedes Feld zu bekommen und es in einer HTML-Tabelle zu speichern. Abbildung 8.1 zeigt, wie der sich ergebende Browserbildschirm aussehen könnte.

```
<%
jdspf.setRecord("SFLRCD01");
while (jdspf.getNextRow()) {
%>

  <tr>
    <td width="25%" align="right"><%= jdspf.getString("XXCNUM") %></td>
    <td width="50%">                <%= jdspf.getString("XXCNAM") %></td>
    <td width="25%" align="right"><%= jdspf.getString("XXCBAL") %></td>
  </tr>

<%
}
%>
```

Auflistung 8.11: Ein Auszug aus dem Code von einer JSP, die ein Subfile emuliert

[3] Wenn Sie mit der HTML- oder JSP-Syntax nicht vertraut sind, entschuldige ich mich; beide Themen sind viel zu umfassend für die Reichweite dieses Kapitels. Ich werde Sie etwas näher in Teil 3 ansprechen.

00000123	Lillian Johnston	142,314.25
00000134	Jon Nave	317,321.34
00000192	Joe Pluta	1.23

Abbildung 8.1: Die Browser-Ausgabe, die aus der JSP in Auflistung 8.11 resultiert

Haben Sie bemerkt, dass das **getString**-Verfahren in dieser JSP nur einen Feldnamen akzeptiert? Dies bedeutet, dass **JdspfDisplayFile** nicht nur weiß, welches seiner Datensatzformate es anschauen muss, sondern in dem Fall eines Subfile auch, welchen Puffer es innerhalb dieses Datensatzformates suchen muss. Sie werden diese Logik durchführen müssen, wenn Sie die Subfile-Support-Classes erzeugen.

Um Subfiles zu implementieren, werden Sie ein paar neue Classes erzeugen müssen. Ich empfehle vier neue Nachfahren von **JdspfAbstractRecord**: **JdspfSubfile**, **JdspfSubfileControl, JdspfMessageSubfile und JdspfMessageSubfileControl.**

Abbildung 8.2 zeigt die vorgeschlagene Hierarchie.

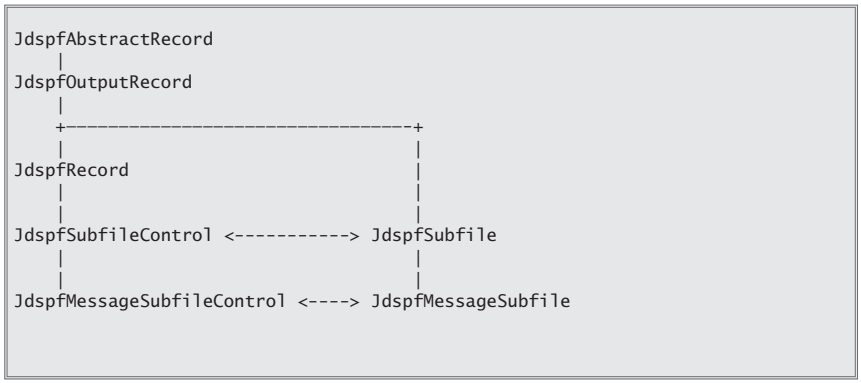

Abbildung 8.2: Die beabsichtigte Hierarchie der JDSPF-Datensatz-Classes

Sehen Sie sich die existierende, nichtfunktionale **JdspfSubfile-**Class an, um eine Vorstellung davon zu bekommen, wie die höher entwickelten Classes aussehen werden. Sie werden sehen, wie multiple Puffer bearbeitet werden: Das Durchführen einer READ, READC oder CHAIN wird einen internen „Cursor" setzen, was den laufenden Puffer anzeigen wird. So wird eine UPDAT den Inhalt des entsprechenden Puffers setzen. Das getNextRow-Verfahren wird dem UI-Server ermöglichen, die Puffer einen nach dem anderen zu wiederholen, wobei er die Cursorposition ändert. Nachfolgende Aufrufe an den **getCurrentBuffer** wird dann den Puffer auf der Cursorposition ausgeben, so dass die **getString**-Operation das Feld von dem entsprechenden Puffer ausgeben wird.

Display-Attribute

Display-Attribute sind eine kompliziertere Angelegenheit. Sie sind Teil eines größeren Konzeptes, der Verwendung von Indikatoren im Allgemeinen. Indikatoren sind eine einzigartige RPG-Eigenschaft, aber eine, die so stark verwendet wird, dass sie in jedem Versuch des Emulierens eines Displayfile behandelt werden muss. Fast jede Facette des Displayfile, von dem Attribut der Felder bis zu den Verhältnissen, die zwischen den Datensätzen bestehen, wird durch Indikatoren gesteuert. Da sie so vorherrschend sind, habe ich ein Indikatorgebiet zu der **JdspfBuffer**-Class hinzugefügt. Tatsächlich ist dieses Indikatorgebiet an seinem Platz den ganzen Weg hinauf in der Green-Screen-Schnittstelle. Obwohl es von der aktuellen Veröffentlichung der Software nicht verwendet wird, kann dieses Indikatorgebiet leicht untergebracht werden und dies wird in der nächsten Ausgabe der Software der Fall sein. Es ist Teil der **DQMMSG-**Datenstruktur, die zum Kommunizieren zwischen dem Anwendungsclient und dem UI-Server verwendet wird.

Der erste Schritt zum Einsetzen von Indikatoren wäre das Hinzufügen eines Parameters zu dem **DQMBAPI2-**Programm, was den Indikatoren ermöglichen würde, von dem Anwendungsclient verschoben zu werden und ausgegeben zu werden. Um das Durchführen des Designs zu vollenden, müssten Sie die folgenden Classes und Verhältnisse hinzufügen:

1. Definieren Sie einen Indikator als numerischen Index des Indikators, von 1 bis 99, und den Status, an oder aus. So würde eine Bedingung von N99 durch **JdspfIndicator(99, false)** repräsentiert werden.

2. Kombinieren Sie multiple Indikatoren mit ANDs und ORs, um komplexe Konditionen wie **((61 and N99) or N73).** Fügen Sie Konstruktoren oder Verfahren hinzu, die geschaffen sind, um die Definition nicht nur der Konditionen, sondern auch ihrer Verhältnisse (die ANDs und ORs) zu erlauben.

3. Ermöglichen Sie eine Kombination einer Kondition und eines Attributes – wie zum Beispiel PROTECT – um zu einem Feld zugeordnet zu werden. Die Attribute würden abgefragt werden, bevor die Daten tatsächlich dem Benutzer präsentiert werden. Im Falle des Browsers würde dies spezielle HTML auslösen, während es in dem Fall der Swing-Schnittstelle Aufrufe auf manche der Feldverfahren erfordern würde.

Sie werden Konditionen zuordnen können müssen zu den anderen Schlüsselwörtern, die in einem Displayfile verwendet werden. Sie werden vielleicht zum Beispiel eine Kondition für das SFLCLR-Schlüsselwort in dem Subfile-Kontrolldatensatz haben. Immer, wenn eine WRITE auf diesem Datensatz durchgeführt wird und die entsprechende Bedingung erfüllt wird, muss das **JdspfDisplayFile**-Objekt das entsprechende Subfile räumen. Wenn das SFLNXTCHG-Schlüsselwort während einer WRITE oder UPDATE auf ein Subfile an ist, wird es bewirken, dass das changed-Signal für diesen Puffer gesetzt wird. Im Laufe der Zeit können mehr Indikatoren innerhalb des **JdspfDisplayFile**-Objektes durchgeführt werden.