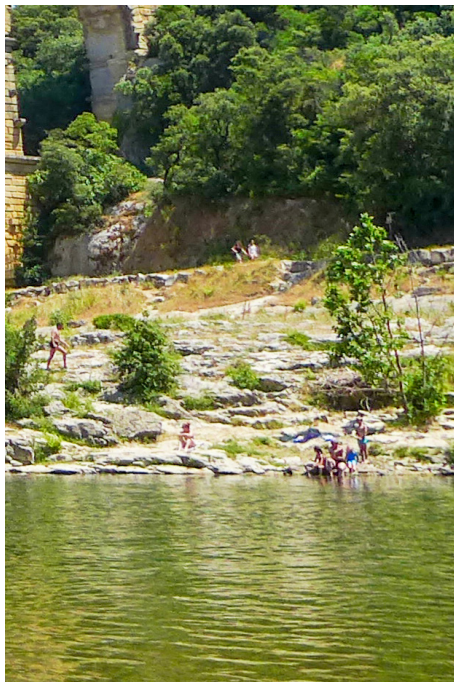




06/2015 Pont du Gard (Département Gard, Okzitanien)

File Services

Da die Vorgänge im Inneren eines Computers so gar nichts Anschauliches an sich haben, arbeitet die IT gern mit Analogien aus der realen, der „analogen“ Welt. Eine dieser Analogien ist die Vorstellung vom „Datenfluss“ als der eines kontinuierlich dahinströmenden Transportmediums. Damit verbinden sich weitere Begriffe, nämlich die von Quelle und Senke, von der Mächtigkeit solcher Flüsse als Volumen, und von der Transfargeschwindigkeit, die beispielsweise bei langsamen Internetverbindungen als „Tröpfeln“ verspottet wird. Datenflüsse sind natürlich nicht möglich, wenn die dafür erforderliche Infrastruktur fehlt, sei es die Hardware oder die zur Organisation der Transfers erforderliche Software. Hier liegt es auf der Hand, dass dafür rein technische Mittel zur Anwendung kommen müssen, während die Menschheit in der längsten Zeit ihrer Existenz auf natürliche Wasservorkommen und deren Strömen von der Quelle zur Senke angewiesen war.

Erst in der Antike wurden Bauten dafür errichtet, Wasser von einem Ort zum anderen zu bringen, wobei die Römer sich als wahre Meister zeigten. Wohl das beste – heute noch existierende – Beispiel dafür ist der Pont du Gard¹ in Südfrankreich. Er war der wichtigste Teil eines 50 km langen Aquädukts, das einst eine Quelle beim Dorf Uzès mit dem 25 km entfernten Nîmes verband. Die zusätzlichen 25 km resultieren aus der geländeangepassten Führung des Aquädukts, der zudem den Fluss Gard (oder Gardon) überqueren musste.

Die im Folgenden beschriebenen File Services schlagen auf ihre Weise eine Brücke von der Programmierung, bei der man sich nicht mit den Details der Transfers von und zu sequentiellen Dateien belasten möchte, zur Realisierung in Form handlicher Schnittstellen, die aus OO-Sicht auf statischen Klassen aufsetzen.

Es ist aus theoretischer Sicht vielleicht keine gute Idee, in einer Buchreihe über die strukturierte Programmierung immer wieder tief in den „Maschinenraum“ abzutauchen, also dorthin, wo die Operationen realisiert werden. Andererseits zeichnen sich manche Bücher zur Software-Methodik dadurch aus, dass sie ihre Leser(innen) ratlos zurücklassen, wie das alles im Innersten funktioniert. Die verbreitete Häppchenfütterung mit Code Snippets befreit zwar die Autoren von der lästigen Aufgabe, wirklich funktionierenden Code herzustellen und zu testen, aber sie bringt auch oft genug Beispiele hervor, denen rasch anzusehen ist, dass sie nicht korrekt sein können. In der Folge keimt bei den Leser(inne)n ein gesundes Mißtrauen gegenüber solchen Werken...

Daher geht diese Buchreihe einen anderen Weg, indem sie anhand zahlreicher funktionierender, getesteter Code-Beispiele zeigt, wie der jeweilige JSP-Entwurf erfolgreich umgesetzt werden kann – nicht: umgesetzt werden muss. In den allermeisten Fällen ist ein Teil davon, die Umsetzung der Operationen in die Zielsprache, eine relativ einfache Sache, die zwar möglicherweise einige Varianten zulässt, aber letztlich recht nüchtern abläuft.

Das gilt jedoch nicht für die sequentiellen Dateizugriffe. Diese sind schon im jeweiligen Betriebssystem von erheblicher Komplexität. Es können hier diverse Fehler auftreten, auf die angemessen reagiert werden muss. Je nach Zielsprache werden zudem höchst verschiedene Konstruktionen für die Datei-Zugriffsschicht zwischen Programm und Betriebssystem angeboten, auf welche die Programmierung in jeder Hinsicht sorgfältig Rücksicht zu nehmen hat. In diesem Kapitel geht es darum, eine Zugriffsschicht für die Java-Programme dieser Buchreihe zu definieren und implementieren (siehe Beschreibung im Band 2-Kapitel „8 File Services in Java“ auf Seite 205 ff).

Ein Motiv für dieses Vorhaben wurde am Anfang des vorhergehenden Kapitels genannt: Generalisierung. Egal wieviele sequentielle Dateien ein Programm verwendet, es sollte immer derselbe Code für die Zugriffe einschliesslich Open / Close zuständig sein. Weitere Motive ergeben sich teils aus der Java-Implementierung

von SonnetAnalysis, teils aus negativen Erfahrungen mit Lauf-Abbrüchen, teils aus Empfehlungen anderer Autoren:

- Programme, die mit sequentiellen Dateien arbeiten, haben nicht immer sowohl Eingabe- als auch Ausgabedateien. Es ist durchaus üblich, nur eine Sorte davon anzutreffen. Die Anzahlen gleichzeitig geöffneter Dateien sind zwar meistens klein, können aber auch mindestens zweistellig werden (selten sind es mehr als circa 10). Daher benötigt die Zugriffsschicht vor dem ersten Öffnen eine Information, wieviele Dateien eingabeseitig oder ausgabeseitig parallel geöffnet sein sollen. Diese Information sollte für beide Seiten getrennt erfolgen, um diese zu entkoppeln. Eine Abfragemöglichkeit ist vorzusehen. Als Default ist Null (Dateien) vorzugeben.
- Es ist nicht erwünscht, Objekt-Referenzen zwischen den Anwendungsklassen durchzureichen, um damit Zugriffe ausführen zu können. Stattdessen sollte jede Datei eine numerische Identifikation erhalten, die über eine Enumeration mit einem symbolischen Namen verbunden ist. Zugriffswünsche in der Applikation beziehen sich allein auf diese Namen. Es ist eine Aufgabe der Zugriffsschicht, die numerischen Identifikationen mit den dadurch repräsentierten Objekten zu verknüpfen. Sie braucht die symbolischen Namen nicht zu kennen. Die numerische Identifikation dient zugleich als Array-Index, weshalb für Ein- und Ausgabe unabhängig voneinander ab Null aufwärts gezählt werden sollte.
- Wenn während des Laufs ein Fehler auftritt, der einen sofortigen Abbruch erfordert, so sollten – auch auf Wunsch der Applikation – alle offenen Dateien durch die Zugriffsschicht geschlossen und ihre Namen protokolliert werden. Andernfalls überlässt man das Schließen der Java-VM oder gar dem Betriebssystem. Dabei geht gewöhnlich der bis dahin aufgebaute Inhalt der Ausgabedateien verloren, weil dort keine normale Close-Sequenz abläuft. Es sind auch Blockaden möglich, weil Betriebsmittel nicht ordnungsgemäß freigegeben werden.
- In vielen Programmen werden zwar Zugriffszähler geführt, aber die Ergebnisse stimmen nicht, oder die Zähler werden zum falschen Zeitpunkt abgelesen. Eine weitere Aufgabe der Zugriffsschicht ist es daher, alle Zugriffe zuverlässig zu zählen und die Zählerstände abfragbar zu machen.
- Wenn ein Fehler auftritt, sollte die fehlerbeschreibende Information nicht nur – auf Wunsch – protokolliert werden, sondern auch weiterhin abrufbar sein – bis sie durch einen anderen Fehler oder eine korrekte Zugriffs-Ausführung überschrieben wird. Die Fehlerprotokollierung sollte an- und abschaltbar beziehungsweise umschaltbar sein, und ihr aktueller Status sollte abgefragt werden können.
- Schleifen, in denen tausende Zugriffsfehler stattfinden, und die womöglich nie enden, sind nicht immer vermeidbar. Jede(r) verprogrammiert sich mal. Daher sollte die Zugriffsschicht ein Fehler-Maximum kennen und nach dessen Überschreitung von sich aus einen kontrollierten Laufabbruch einleiten. Dieses Maximum sollte nicht starr sein, um beispielsweise nach erfolgreichem Anlauf des Programms auf Nulltoleranz umschalten zu können. Auch hier ist eine Abfragemöglichkeit vorzusehen.
- Die Dateinamen, die der Zugriffsschicht für das Öffnen übergeben werden, sollten darin bis zum Schließen registriert bleiben. So kann ein Programm jederzeit die mit den symbolischen Namen verknüpften realen Dateinamen erfragen und diese beispielsweise für Fehlermeldungen oder Statistiken verwenden.
- Es sollte möglich sein, dieselbe Datei mehrfach – zum Beispiel zuerst zum Schreiben und später zum Lesen – oder mehrere Dateien in derselben Rolle nacheinander zu öffnen. Daher ist zwischen den Dateien und ihren Rollen zu unterscheiden, die sie zu einer bestimmten Zeit für das Programm einnehmen. Die oben geforderte numerische Identifikation pro Datei ist daher eigentlich eine Rollen-Identifikation.
- Schwerwiegende Zugriffsfehler, also Exceptions, können von Anwendungsprogrammen meist nicht sinnvoll behandelt werden. Exceptions treten auf, wenn Dateinamen ungültig sind oder die damit bezeichneten Eingabedateien nicht existieren, wenn ein Medium nicht beschrieben werden kann (CD-ROM / Flash-Medium mit Schreibsperre etcetera), wenn es nicht genug Platz für eine große Ausgabedatei bietet, oder wenn davon nicht gelesen werden kann (zum Beispiel nach dem Abstecken eines externen

Mediums), sowie aus vielen anderen Gründen. In dieser Buchreihe geht es nicht darum, Programme auch gegen solche irreparablen Fehler zu härten. Es genügt völlig, dass dann ein kontrollierter Abbruch mit Schließen aller offenen Dateien stattfindet.

Es sind also überraschend viele Anforderungen, welche die hier vorgeschlagene Zugriffsschicht erfüllen soll. Ein zentraler Aspekt ist dabei die Forderung nach numerischen Identifikatoren, also Dateinummern, welche die Eingabedateien und die Ausgabedateien unabhängig voneinander bekommen sollen. Wie steht es damit?

14.1 Numerische Datei-Identifikationen

Im Gegensatz zu COBOL und beispielsweise REXX werden Dateibeziehungen in Java-Programmen – und in vielen anderen, auch algorithmischen Sprachen wie T/TAL – nicht durch starre Namen repräsentiert. In COBOL sind dagegen jeder Datei eine eigene SELECT-Klausel für die Zuordnung des jeweiligen internen Dateinamens zu seinem externen Kurznamen (IBM-Hosts: DD-Namen), ein eigener FD-Eintrag (FD = File Definition) für die – grobe – Satzdefinition und je eine File-Status-Variable fest zugeordnet. Dateibeziehungen können nicht zu COBOL-Arrays zusammengefasst werden, wie das beispielsweise bei einem Ausgabefächer für zahlreiche Abnehmer sinnvoll wäre.

In Java ist das kein Problem. Wir müssen „nur“ die bisherigen einzelnen File-Repräsentanzen, nämlich die `BufferedReader`- und `BufferedWriter`-Objekte, vervielfältigen und dafür sorgen, dass bei der Nutzung dieser Objekte kein Chaos ausbricht. Als Organisationsmittel dient hier jeweils ein Array. Darin sind mindestens so viele Einträge erforderlich, wie es parallel geöffnete Eingabedateien oder Ausgabedateien geben soll. Genauer gesagt: Für jede Datei-Rolle ist ein Eintrag vorzusehen. In Programmen, die mehrere Phasen durchlaufen, können anfangs belegte Rollen-Plätze später frei werden und umgekehrt.

Ein Beispiel für die Belegung der numerischen Identifikatoren zeigt übrigens das Kapitel „9.2.2 Klasse `StreamCode`“ auf Seite 252 im Band 2.

14.2 Erster Vorschlag

14

Vielleicht ist ja alles ganz einfach? Wir könnten uns zunächst an der bisherigen Klasse `FileService` orientieren und deren Verhalten flexibler gestalten, also insbesondere variable Anzahlen von Datei-Rollen anstelle der bisherigen 1:1-Konfiguration ein- und ausgabeseitig zulassen. Damit die Nutzer-Methoden im übrigen Programm jeweils die richtige Datei ansprechen könnten, müssten zwei statische Hilfsklassen die `InFileService`-/`OutFileService`-Exemplare registrieren. Das würde es ermöglichen, aus jeder Methode ausserhalb der `File Services` beliebige Dateien zu bearbeiten.

Die „Abbildung 14.1: Erster Vorschlag für die Klassenstruktur der Zugriffsschicht“ auf Seite 432 zeigt am Beispiel von drei Eingabe- und zwei Ausgabedateien, wie die nutzenden Klassen gemäß diesem Vorschlag die `Services` ansprechen, wie letztere sich bei den beiden Hilfsklassen registrieren, und wie die Aufrufe im Crash-Fall erfolgen:

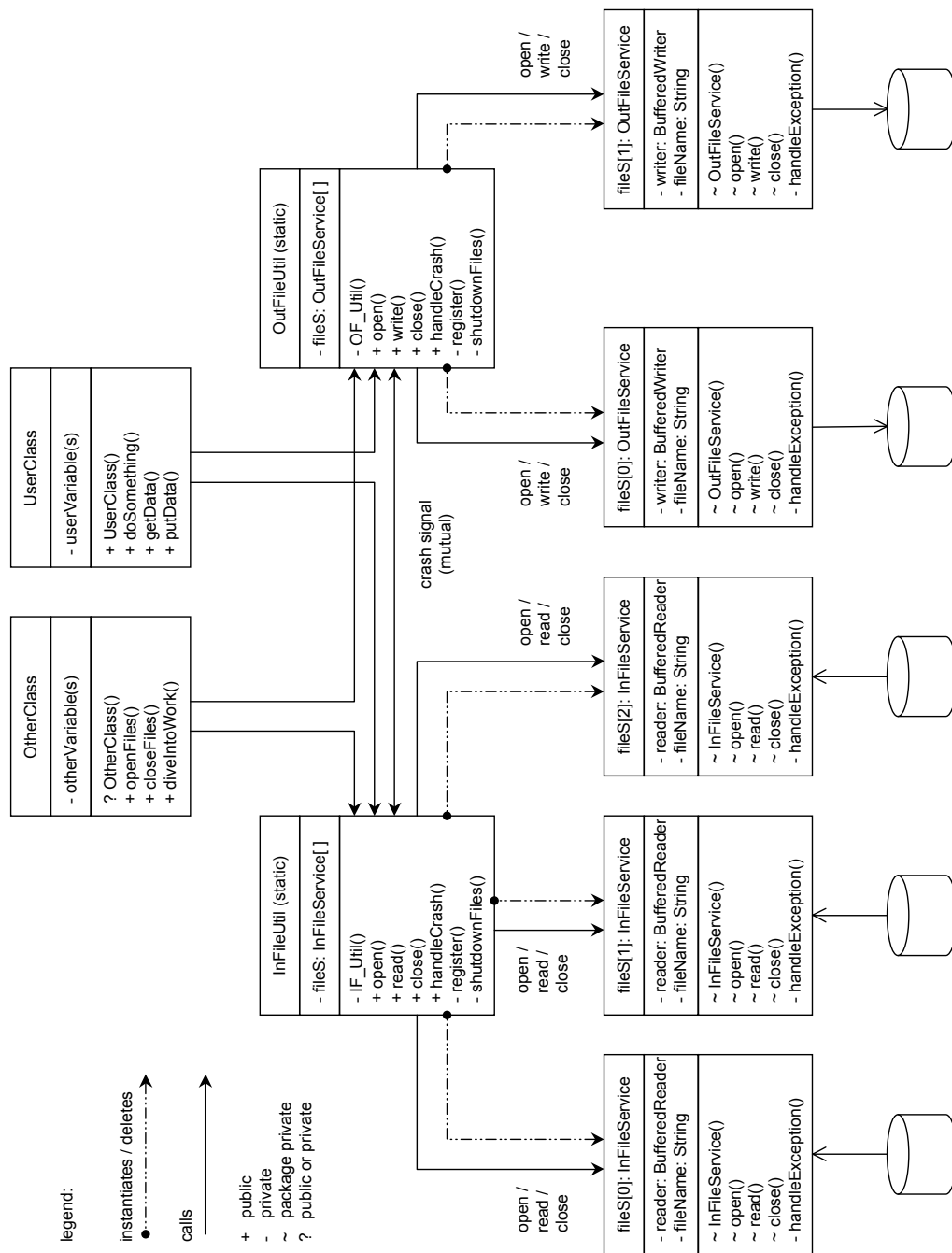


Abbildung 14.1: Erster Vorschlag für die Klassenstruktur der Zugriffsschicht

Die Arbeitsweise dieser Konstruktion ist wie folgt:

- Die `OtherClass` ist unter anderem dafür zuständig, die Dateien öffnen und schließen zu lassen. Für jede zu öffnende Datei ruft sie die dazu passende statische Utility-Klasse auf, zum Beispiel `InFileUtil`. Diese

erzeugt ein `InFileService`-Objekt, das seine Referenz rückmeldet, registriert es im Array `files`, und ruft die `open`-Methode des neuen Objekts auf, die den Dateinamen als Exemplarvariable speichert. Der `InFileService`-Konstruktor stellt also noch kein voll funktionstüchtiges Objekt zur Verfügung. Der Grund dafür ist, dass ein `open()` im Konstruktor eine Exception auslösen könnte, was grundsätzlich unerwünscht ist (siehe das Buch von Michael Inden „Der Weg zum Java-Profi – Konzepte und Techniken für die professionelle Java-Entwicklung“ [9], Kap. 10.3.11). Da die Objekterzeugung und der `open`-Aufruf zusammen gekapselt sind, kann jedoch keine Nutzer-Klasse auf eine unvollständig initialisierte File-Verbindung treffen.

- Analog gilt dasselbe für die Ausgabeseite. `OutFileUtil` erzeugt ein `OutFileService`-Objekt, registriert dessen Referenz, und ruft die `open`-Methode des neuen Objekts auf.
- Lese- und Schreib-Anforderungen aus den Nutzer-Klassen – `UserClass` – gehen nicht direkt an die File Service-Objekte, sondern werden von den Utility-Klassen in objektbezogene Aufrufe der File Service-Methoden umgesetzt. Diese sind – bis auf die private Methode `handleException` – als `package private` vor externen Aufrufen geschützt, wenn sie zusammen mit den Utility-Klassen in einem eigenen Package definiert sind.
- Für jede zu schließende Datei wird die jeweils zuständige Utility-Klasse aufgerufen, die den `close` beim hierfür zuständigen Objekt anfordert, dessen Referenz – für den Garbage Collector – auf `null` setzt und diese aus dem `files`-Array entfernt. Eine geschlossene Datei kann daher erneut geöffnet werden, wozu `TL_Util / FileService` in `SonnetAnalysis` nicht imstande sind.
- Die Arbeitsteilung zwischen `OtherClass` und `UserClass` ist hier zwar naheliegend, aber dennoch willkürlich gewählt. Dateibezogene Anforderungen können beliebig über die Klassen oberhalb der Utility-Klassen verteilt sein. Es ist auch möglich, dass Dateien mehrfach geöffnet, genutzt und geschlossen werden, oder dass bestimmte Datei-Rollen nacheinander durch mehrere Dateien – beispielsweise solchen mit Bilddaten, die seriell konvertiert werden sollen – eingenommen werden.
- Wenn eine Exception auftritt, muss das davon betroffene File Service-Objekt zunächst eine geeignete Protokollierung veranlassen und dann seine Utility-Klasse durch Rückmeldung der Exception-Informationen über den Vorfall informieren. Dieser löst `close`-Aufrufe in allen seinen File Service-Objekten aus, wobei die möglicherweise dabei auftretenden Exceptions ignoriert werden müssen. Ausnahmen: Ein fehlgeschlagener `close`-Aufruf sollte nicht wiederholt werden, und nach einer `open`-Exception ist ein `close`-Aufruf für die betroffene Datei auch nicht sinnvoll. Ausserdem muss jede Utility-Klasse die jeweils andere per `handleCrash`-Aufruf über das Ereignis informieren, damit auch diese die erforderlichen `close`-Aufrufe veranlassen kann. Abschließend wird der Lauf durch einen `System.exit()`-Aufruf beendet.

Diese Konstruktion hat Vorteile, die bislang nicht beachtet wurden:

- Die Utility-Klassen können unsinnige Aufrufe (zum Beispiel Lesen aus einer Ausgabedatei), falsche Aufruffolgen (`open` – `open` / `close` – `close`) oder `read` / `write` für geschlossene Dateien verhindern. Dadurch werden Fehler erkannt, bevor daraus Exceptions werden.
- Die File Service-Objekte übertragen die Kontrolle über die `close`-Aktionen beim Laufabbruch an ihre Utility-Klassen. Somit wird vermieden, dass sich diese Objekte gegenseitig aufrufen und die hierfür erforderlichen Referenzen von den Utility-Klassen abholen müssen.

Als nachteilig erscheint dagegen, dass

- die Utility-Klassen einander kennen. Dadurch wird eine Compile-Abhängigkeit geschaffen, die nicht zu der Forderung passt, dass es beispielsweise nur eine Eingabedatei, aber keine Ausgabedatei geben können soll.

- Nutzer die öffentliche `handleCrash`-Methode einer Utility-Klasse verwenden und das Programm dadurch zum Absturz bringen könnten, denn deren Aufruf führt zu einer halbseitigen Schließung. Es würden also nur die Eingabedateien oder nur die Ausgabedateien geschlossen.
- weitere eingangs genannte Anforderungen ebenfalls eine Zusammenarbeit der Utility-Klassen erzwingen, etwa das Zwischenspeichern von Fehler-Informationen, das Einhalten eines globalen Fehler-Limits oder das Zählen von Zugriffen.

Der potentielle Mißbrauch von `handleCrash()` stellt eher eine theoretische Befürchtung dar. Erheblich ungünstiger ist, dass nach diesem Vorschlag die Utility-Klassen nicht unabhängig voneinander verwendet werden könnten. Daher ist er ungeeignet.

14.3 Zweiter Vorschlag

Nein, es ist nicht alles ganz einfach, im Gegenteil. Daher sollten wir nach diesem „Schnellschuss“ noch einmal nachdenken. Was soll eigentlich erreicht werden?

- Keine, eine oder mehrere Eingabedateien und keine, eine oder mehrere Ausgabedateien sollen flexibel parallel und auch mehrfach nacheinander genutzt werden können, ohne Compile-Abhängigkeiten zu schaffen.
- Die hierfür erforderliche – weiter oben näher beschriebene – Funktionalität soll so versteckt werden, dass sie nicht manipuliert oder mißbräuchlich verwendet werden kann.
- Beim Auftreten einer Exception sollen nach Ausgabe geeigneter Fehlerinformationen *alle* offenen Dateien geschlossen und dann der Lauf beendet werden.

Zusätzlich ist es wünschenswert, im Abbruchfall die Namen der offenen Dateien (mit Pfadangaben), die jeweiligen Betriebsart – Eingabe / Ausgabe – und die Anzahlen der gelesenen / geschriebenen Sätze auszugeben. Das hilft bei der Fehlersuche.

Um das Erzeugen von `InFileService` / `OutFileService`-Exemplaren zu vermeiden, empfiehlt es sich, zu einer statischen Klassenkonstruktion zurückzukehren, wie sie für Utility-Klassen typisch ist. Damit dennoch multiple Dateien verwaltet werden können, müssen die Exemplarvariablen aus dem 1. Vorschlag um Zugriffszähler ergänzt und in Array-Elemente umgewandelt werden. Das Dilemma der gegenseitigen Aufrufe lässt sich mit einer gemeinsamen – dritten – statischen Klasse lösen, die einerseits als Repository für die datei-bezogenen Informationen dient, und andererseits im Crash-Fall subsidiär alle Abschlussoperationen – ausser der Exception-Text- und Stack-Trace-Ausgabe – übernimmt. Daher ist diese Klasse auch der richtige Ort für die Arrays.

Nach diesen Vorgaben wurde programmiert, ohne die Verarbeitungsschritte als Struktogramm zu modellieren, da dies eine nutzlose formale Übung gewesen wäre (wie sich durch Betrachtung des Codes leicht erkennen lässt). Aus der Implementierung ergibt sich die „Abbildung 14.2: Zweiter Vorschlag für die Klassenstruktur der Zugriffsschicht“ auf Seite 435.

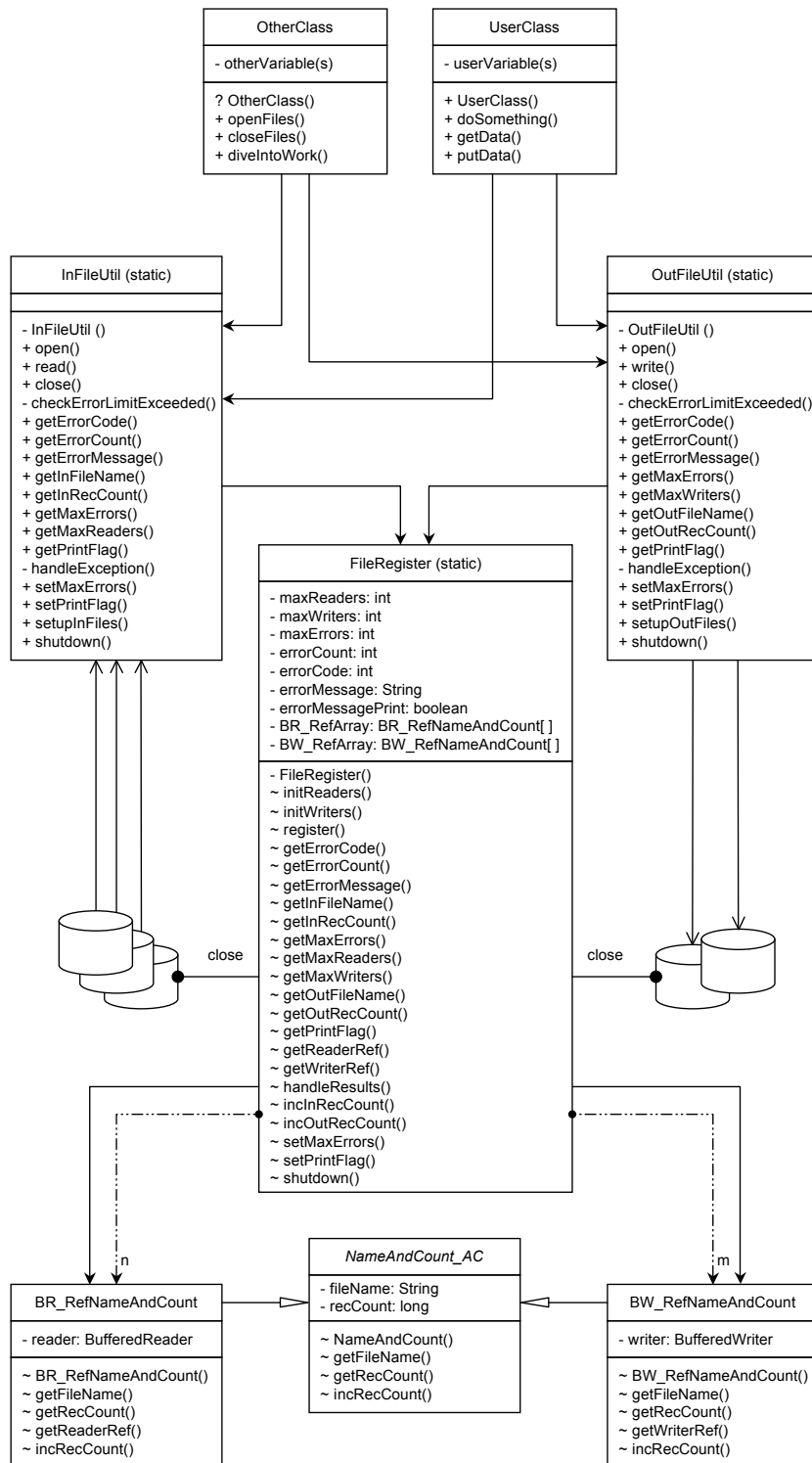


Abbildung 14.2: Zweiter Vorschlag für die Klassenstruktur der Zugriffsschicht

`OtherClass` und `UserClass` können nun über eine erheblich erweiterte Funktionalität der File Services verfügen. Die beiden statischen Utility-Klassen haben keine Klassenvariablen mehr, sind also nur noch Methodensammlungen, die mit lokalen Variablen arbeiten. Sie rufen sich auch nicht mehr gegenseitig auf. Die Klasse `FileRegister` verwaltet Konfigurationsdaten, Fehlerinformationen, sowie die zum Lesen, Schreiben oder Schließen benötigten Objektreferenzen (in zwei Arrays). Ihre Methoden sind `private` (Konstruktor) oder `package private` (alle anderen). Somit können sie nur package-intern aufgerufen werden, sind also für Anwendungsklassen unsichtbar. Die Array-Elemente sind durch die Klassen `BR_RefNameAndCount` und `BW_RefNameAndCount` beschrieben, die von der gemeinsamen abstrakten Klasse `NameAndCount_AC` erben. Die Methoden dieser drei Klassen sind sämtlich `package private`.

Wie funktioniert das?

- Die Klasse `FileRegister` soll ihre Arrays mit festen Längen anlegen. Dafür muss sie „wissen“, wieviele verschiedene Eingabe- und Ausgabe-Datei-Rollen es gibt. Hierfür gibt es die Methoden `initReaders()` und `initWriters()`, die von `setupInFiles()` oder `setupOutFiles()` aufgerufen werden. Da Ein- und Ausgabe voneinander unabhängig konfiguriert werden, besteht kein Zwang für die Anwendungsklassen, die erwarteten Anzahlen der Datei-Rollen zur selben Zeit zu übergeben. Der Wert von `maxReaders` – Kardinalität n in der Grafik – kann mit `getMaxReaders()` und der Wert von `maxWriters` – Kardinalität m – mit `getMaxWriters()` abgerufen werden. Es ist ausdrücklich zulässig, dass ein Programm keine Eingabedateien oder keine Ausgabedateien hat.
- In vielen Fällen sind Anwendungen nicht darauf vorbereitet, dass Fehler oder Exceptions auftreten. Daher kann die maximale Anzahl von Fehlern, welche die File Services selbst feststellen, per `setMaxErrors()` vorgegeben werden. Ist dieses Limit überschritten, wird die Shutdown-Sequenz automatisch eingeleitet. Auch die Vorgabe `Null` ist möglich. Negative Werte bedeuten: Kein Limit. Die Standardeinstellung lässt beliebig viele Fehler zu. Exceptions führen dagegen immer zum Shutdown. Anwendungsklassen können auch selbst einen Shutdown auslösen und dabei einen Exit-Code, sowie – optional – einen einzeiligen Text mitgeben. Das aktuell gültige Limit kann mit `getMaxErrors()` abgerufen werden, und die aktuelle Fehlerzahl mit `getErrorCount()`.
- Wenn die Anwendung die Fehlerausgabe nicht selbst übernehmen, sondern diese den File Services überlassen will, kann sie die Standardeinstellung bestehenlassen. Sie kann diese Ausgabe aber auch unterdrücken und die Fehlerinformationen abrufen. Hierfür stehen in beiden Utility-Klassen `getErrorXXX()`-Methoden zur Verfügung, die auf korrespondierenden Methoden in `FileRegister` aufbauen. Diese Einstellung kann mit `setPrintFlag()` jederzeit geändert und mit `getPrintFlag()` abgerufen werden.
- Jede Open-Anforderung seitens der `OtherClass` übergibt den Pfad- und Dateinamen, sowie eine File-ID, die der Einfachheit halber für Eingabe wie für Ausgabe jeweils bei `Null` beginnt und mit einer – nicht gezeigten – Enumeration `StreamCode` verknüpft ist. Diese Nummer darf den Wert von `maxReaders` respektive `maxWriters` nicht erreichen und muss für jede Datei-Rolle eindeutig vergeben werden. Die Betriebsart geht daraus hervor, welche Utility-Klasse – `InFileUtil` für Eingabe oder `OutFileUtil` für Ausgabe – angesprochen wird.

Diese ruft die zugehörige `getXXXRef()`-Methode in `FileRegister` auf, um festzustellen, ob zu der File-ID bereits eine Datei registriert ist, also eine falsche Aufruffolge (Open nach Open) vorliegt. Trifft das zu, wird es – falls gewünscht – protokolliert und ein Fehlercode rückgemeldet. Andernfalls öffnet die Utility-Klasse die Datei analog zu `FileService` und liefert im Erfolgsfall eine `BufferedReader`- oder `BufferedWriter`-Referenz, sowie die Dateimerkmale – Pfad & Name, File-ID – per `register()`-Aufruf an `FileRegister`. Dort werden diese Angaben an den Konstruktor der jeweils zuständigen Array-Element-Klasse übergeben und der damit assoziierte Zugriffszähler – in der Oberklasse `NameAndCount_AC` – initialisiert. Die Datei ist jetzt betriebsbereit.

- Bei jedem `read`-Aufruf seitens einer `UserClass` muss sich `InFileUtil` die `BufferedReader`-Referenz durch Übergabe der File-ID an `getReaderRef()` aus `FileRegister` holen, wo diese Anforderung an das zuständige Array-Element-Objekt der Klasse `BR_RefNameAndCount` weitergeleitet wird. Mit dieser Referenz greift `InFileUtil` zu und erhöht den Lese-Zugriffszähler per `incInRecCount()` und darauf folgend in `FileRegister` per `incRecCount()`, wenn ein Satz gelesen wurde. Bei EOF wird also nicht inkrementiert.

Die Verwendung einer falschen – negativen, zu hohen oder aktuell nicht verwendeten – File-ID führt dagegen zur Ausgabe einer Fehlermeldung und – wie bei EOF – zur Rückmeldung `null`. Daher muss die lesend zugreifende Anwendungsklasse bei der Rückmeldung `null` per `getErrorCode()` prüfen, um welchen dieser Fälle es sich handelt.

- Analog gilt dasselbe für jeden Schreib-Aufruf an `OutFileUtil` und das Abholen der jeweiligen `BufferedWriter`-Referenz per `getWriterRef()` beim zuständigen Array-Element-Objekt der Klasse `BW_RefNameAndCount`. Der Schreib-Zugriffszähler wird per `incOutRecCount()` und dann `incRecCount()` immer inkrementiert. Bei Verwendung einer ungültigen File-ID meldet `write()` einen `int`-Fehlercode zurück.
- Auch das Schließen erfolgt zweiphasig. Zunächst muss die jeweilige Utility-Klasse die benötigte Referenz beschaffen. Die Fehlerreaktion ist wie beim Lesen / Schreiben. Wenn die Utility-Klasse eine gültige Referenz vom zuständigen Array-Element-Objekt zurückbekommt, schließt sie die Datei und registriert dies in `FileRegister`. Dort wird dieses Array-Element-Objekt durch `null`-Referenz-Zuweisung dem Garbage Collector gemeldet.
- Falls Zugriffszähler aus den Nutzer-Klassen heraus – beispielsweise für statistische Zwecke – unmittelbar vor dem Schließen ausgelesen werden sollen, kann `getInRecCount()` oder `getOutRecCount()` je nach Utility-Klasse aufgerufen werden. Im Fehlerfall wird -1 rückgemeldet, weil sonst vom Aufrufer eine Zahl ab 0 aufwärts erwartet wird.
- Auch die Namen der geöffneten Dateien können jederzeit gezielt abgefragt werden. Hierfür existieren die Utility-Methoden `getInFileName()` und `getOutFileName()`.
- Nun verbleibt noch der Exception-Fall. Die davon betroffene Utility-Klasse erzeugt zunächst mittels `handleException`-Aufruf die fehlerbeschreibenden Protokollausgaben (nach dem Vorbild von `printErrorInfoCloseAndExit()` in `FileService`). Ausser bei einer OPEN-Exception beschafft sie sich zudem den zugehörigen Dateinamen per `getFileName()`-Aufruf von `FileRegister` und fügt diesen den Exception-Ausgaben hinzu. Tritt die Exception bei der `streamReader`- oder `streamWriter`-Erzeugung auf, so steht der Dateiname als Parameter zur Verfügung und wird ebenfalls ausgegeben. Grund: Die Vorgabe der – optionalen – Codierung ist Sache der Nutzer-Klasse und deshalb potentiell fehleranfällig.

Nach der Stack-Trace-Ausgabe folgt ein Aufruf von `shutdown()` in `FileRegister` unter Mitgabe der File-ID, der Betriebsart, der Angabe der fehlgeschlagenen Aktion, sowie eines Exit-Codes. `shutdown()` arbeitet alle registrierten Dateien ab:

- Ausgabe des Dateinamens, der Betriebsart und des Zugriffszähler-Werts
- Close-Versuch ausser nach misslungenem Open oder Close
- Tritt dabei eine Exception auf, wird sie nur kurz protokolliert, aber nicht weiter behandelt
- Final: Aufruf von `System.exit()` mit dem an `shutdown` übergebenen Exit-Code

Der 2. Vorschlag löst das Compile-Problem des 1. Vorschlags und sorgt dafür, dass alle offenen Dateien bei einem Laufabbruch geschlossen werden. Die Methoden von `FileRegister` und seiner Hilfsklassen sind alle privat oder nur paketsichtbar, können also nicht dazu mißbraucht werden, die File Service-Funktionalität durcheinanderzubringen. Die File Services müssen lediglich ein eigenes Paket bekommen, um den gewünschten Schutz zu realisieren.

Positiv zu werten ist zudem, dass auch in Abbruchfällen, die nichts mit der Ein- oder Ausgabe zu tun haben, durch Aufruf von `shutdown()` eine saubere Beendigung des Laufs mit Schließen aller offenen Dateien möglich ist. Hierfür wurde jede Utility-Klasse um eine öffentliche `shutdown()`-Methode erweitert.

In der Grafik und im vorangehenden Text ist nicht erwähnt, dass eine weitere neue Klasse `FS_Code` definiert wurde, um die Fehler- und Exception-Codes zu verwalten. Hier handelt es sich ebenfalls um eine Enumeration. Die Methoden der File Services machen intensiven Gebrauch von den in `FS_Code` hinterlegten Angaben, wie aus dem Implementierungskapitel im Band 2 „8 File Services in Java“ auf Seite 205 ff hervorgeht.

Diese Implementierung basiert auf dem „alten“ Paket `java.io`, das zunehmend – aber derzeit (2022) noch nicht vollständig – durch `java.nio.file` abgelöst wurde. Da es in dieser Buchreihe nicht darum geht, Java zu lehren und sich deshalb an dessen rasche Fortentwicklung anzupassen, sondern vielmehr darum, funktionierende Implementierungen auf der Basis der strukturierten Programmierung zu entwickeln, erscheint diese technologische „Rückständigkeit“ der File Services als durchaus tolerabel. In späteren Jahren, wenn `java.nio.file` ausgereift sein wird, kann ein Umstieg erwogen werden.

Das bedeutet nicht, dass auf `java.nio.file` verzichtet wird. Vielmehr kommt dieses Paket hauptsächlich im Band 4 zur Anwendung, wo die darin enthaltenen „neuen“ Klassen extrem nützlich sind.



Endnoten

1 Der **Pont du Gard** (okzitanisch Pònt de Gard) ist ein römischer Aquädukt im Süden Frankreichs auf dem Gebiet der Gemeinde Vers-Pont-du-Gard im Département Gard. Die Brücke ist von beeindruckender Höhe und stellt einen der am besten erhaltenen Wasserkanäle aus der Römerzeit in Frankreich dar. Der Pont du Gard zählt zu den wichtigsten erhalten gebliebenen Brückenbauwerken der antiken römischen Welt und ist eine der bedeutendsten Sehenswürdigkeiten Südfrankreichs.

Pont du Gard bedeutet übersetzt Gard-Brücke. Der Fluss Gard wird heutzutage meist Gardon genannt, von ihm leitet sich auch der Name des Départements ab.

[...]

Der Pont du Gard war Teil einer etwa 50 km langen Wasserleitung, mit der Wasser von den Quellen nahe Ucetia (Uzès) zur römischen Stadt Nemausus (Nîmes) transportiert wurde. Die Brücke ist 49 m hoch und umfasst drei Etagen:

Untere Ebene: 6 Bögen, 142 m lang, 6 m breit, 22 m hoch

Mittlere Ebene: 11 Bögen, 242 m lang, 4 m breit, 20 m hoch

Obere Ebene: 35 Bögen, 275 m lang, 3 m breit, 7 m hoch

Auf der oberen Ebene verläuft das rechteckige Gerinne der Wasserleitung, das 1,80 m hoch und 1,20 m breit ist und ein Gefälle von 0,34 ‰ aufweist.

Auf der unteren und mittleren Etage der Brücke befinden sich Arkaden aus 61 bis zu 6 t schweren Keilsteinen. Die Pfeiler der mittleren Ebene sind genau auf den Pfeilern der unteren Etage aufgelagert, um die Belastung der unteren Gewölbebögen zu minimieren. Von der Mitte ausgehend wird die Bogenspannweite zum Ufer hin immer kleiner.

[Wikipedia-Seitentitel: Pont du Gard]

Datum der letzten Bearbeitung: 16. Mai 2022, 12:04 UTC

Versions-ID der Seite: 222923377

Permanentlink: https://de.wikipedia.org/w/index.php?title=Pont_du_Gard&oldid=222923377

Datum des Abrufs: 15. Juni 2022, 10:09 UTC]