Das Ergebnis zählt

Aktives Fehlermanagement mit der Rslt-Klasse

Abstract: Die Rslt Klasse ist ein einfaches, aber wirksames Instrument, um die Behandlungen von Fehlern im Programmablauf nicht nur zu behandeln, sondern zu wertvollen Quellen für die Fehlersuche und -Analyse zu machen. So gesehen stellt sie eine Weiterentwicklung der Return-Code-Verarbeitung dar. Exceptions können einfach eingebunden und deren Informationen gewinnbringend verwertet werden.

Die Rslt Klasse bietet die Möglichkeit einer einheitlichen Fehlerbehandlung, die wegen des geringen Aufwands hohe Akzeptanz bei den Entwicklern genießt. Durch die Verringerung des Aufwands der Fehlerbehebung und der verkürzten Turn-Around-Zeiten ermöglicht die Rslt-Klasse im Verlauf des Software-Lebens-Zyklus einen Produktivitätsgewinn, und zwar bereits vor der Auslieferung des ersten Releases.

Fehler in der Software

Fehler gehören in der Software-Entwicklung zum Alltag. Sie verursachen Ausfallzeiten, Irritationen und schmälern die Produktivität. Eine ordentliche Fehlerbehandlung dagegen ist noch immer nicht der Standard.

Die typische Vorgehensweise in der Software-Entwicklung handelt nach dem Motto „wir greifen ein, wenn etwas passiert ist“. Jedoch ist es dann oftmals zu spät und es können nur noch die Symptome des Endzustands übermittelt werden. Fehlermeldungen wie z.B. „Ein unbekannter Fehler ist aufgetreten“ dokumentieren, dass nur noch auf oberster Ebene der Funktionen-Hierarchie eine Exception gefangen wurde, bevor das Betriebssystem die Anwendung beendet hätte. In solchen Fällen kann nicht mehr eingegriffen werden und dem Anwender bleibt nichts weiter übrig, als den Fehler zu quittieren. Die Information, wie es zu dem Fehler kam und wie er in Zukunft vermieden werden kann, ist verloren. Der Wert einer solchen Fehlermeldung lässt den Benutzer Rätsel raten - eine Dokumentation der Hilflosigkeit?

Aktives Fehlermanagement zielt darauf ab die Robustheit der Programme und die Qualität der Information im Fehlerfall zu erhöhen.

Aktives und Reaktives Fehlermanagement

Reaktives Fehlermanagement ist die Art von Fehlermanagement, wie es heute in der Softwareentwicklung weit verbreitet ist: nachdem der Fehler aufgetreten ist wird mit möglichst geringem Aufwand versucht den Schaden eines Fehlers einzugrenzen. Je nach Motivation des Entwicklers wird versucht mehr oder weniger der vorhandenen Information in einem Log auszugeben, schlimmstenfalls wird einfach alles, was verfügbar ist protokolliert. Sollte der Fehler im Zusammenhang mit einer außen stehenden Komponente stehen, so ist es dieser überlassen, die für den Fehler relevanten Informationen zu protokollieren. Die Fehlerprotokollierung macht jeder Hersteller, und unter Umständen jeder Entwickler, anders und die Fehleranalyse ist entsprechen mühselig.

Aktives Fehlermanagement dagegen bedeutet die Verarbeitung nur dann fortzusetzen, wenn ein

klares, positives Signal vorhanden ist. Ist dieses nicht vorhanden, wird sichergestellt, das alle zur Analyse relevanten Informationen weitergereicht werden. Die Verarbeitung wird fortgesetzt, nun aber um die Fehlerinformation zu transportieren. Die sorgt zunächst einmal für Robustheit der Programme. In jedem Fall ist eine kontrollierte Forstsetzung des Programms möglich. Die im Arbeitsfluss folgenden Schritte werden nicht mehr ausgeführt und damit Folgefehler vermieden.

Aktives Fehlermanagement hat aber zunächst einmal einen höheren Kodierungsaufwand zur Folge. Diesen zu minimieren, die Qualität der Fehlerinformation zu steigern und das aktive Fehlermanagement optimal zu unterstützen ist Ziel der Rslt-Klasse.

Der Anfang: Return Codes

Begonnen hatte alles mit der Frage "Wenn ich bei der Abfrage der Datenbank einen Leer-String zurückbekomme, was sagt mir das? Gab es einen Fehler? Oder wurden einfach nur nicht die passenden Daten gefunden?" und weiter „Wie und wo entscheide ich, ob das ein akzeptables Ergebnis ist oder als Fehler gewertet werden muss?". Das Problem stellte sich praktisch bei jeder Operation eines Programms. Zusätzlich sollte im Fehlerfall so viel Information wie möglich gesichert werden, um das Problem schnell eingrenzen zu können.

Der Standard offerierte keine Lösung. Objektorientierung hatte sich noch nicht durchgesetzt. C war die Programmiersprache der Wahl und daher bestand die Lösung darin, den Code so zu

```
//Dies ist eine gegenüber dem tatsächlichen Code stark vereinfachte Version.
int getLicenseString( string p_UserName, string * p_LicenseString ) {

        int rc;
        string clientID;

        rc = readFromDB( „UserName", „ClientID", p_UserName, &clientID );

        if( rc != 0 ) {

                logError( „getLicenseString(): Failed to retrieve 'ClientID' from
                                database. RC: „, rc );
        }
        else {

                rc = readFromDB( „ClientID", „LicenseString", clientID,
                                        p_LicenseString );

                if( rc != 0 ) {
                        logError( „getLicenseString(): Failed to retrieve
                                        'LicenseString' from database. RC: „, rc );
                }
        }

        return rc;
}
```

Text 1: Instrumentierter Code, alt

instrumentieren, dass einerseits eine fehlertolerante Verarbeitung garantiert wurde und gleichzeitig im Fehlerfall hilfreiche Informationen geliefert wird. Auf diese Art kam aber auf eine Zeile Nutz-Code netto drei Zeilen Fehlerbehandlungs-Code. Die Qualität der Fehlerinformation war von der Disziplin der einzelnen Entwickler abhängig. Wurde die Kette unterbrochen, d.h. lieferten die Unterfunktionen oder die nachfolgenden Funktionen nicht ähnliche Fehlerinformationen in ähnlicher Qualität, wurde die Information nutzlos.

Bei nur etwas längeren Code-Sequenzen ergab sich schnell eine wachsende If-Then-Else-Schachtelung, die den Nutz-Code immer weiter nach rechts rückte. Für die Lesbarkeit war dies nicht gerade förderlich.

Exception Handling

Mit der Objektorientierung ging auch die Einführung des Exception-Handlings einher. Dies schien ein recht brauchbares Instrument für die Fehlerbehandlung zu sein. Die Umsetzung sah folgendermaßen aus:

```
//Dies ist eine gegenüber dem tatsächlichen Code stark vereinfachte Version.
string getNumberLicenses( string p_UserName ) {

        int rc;
        string clientID;
        string licenseString;

        try {

                clientID = readFromDB( „UserName“, „ClientID“, UserName );
        }
        catch( Exception excp ) {

                Exception newExcp = new Exception( „getLicenseString(): Failed to
                                                    retrieve 'ClientID' from
                                                    database.“, excp );
                throw newExcp;
        }

        try {

                licenseString = readIntFromDB( „ClientID“, „LicenseString“,
                                                clientID );
        }
        catch( Exception excp ) {

                Exception newExcp;

                StackTraceElement[] ste = excp.getStackTrace();

                nExcp = new Exception( "Failed to retrieve 'LicenseString' from
                                       database." );
                nExcp.initCause( excp );
                nExcp.setStackTrace( ste );

                throw newExcp;
        }


        return licenseString;
}
```

Text 2: Fehlerbehandlung mit Exceptions

Der Versuch, Exceptions zu nutzen, enttäuschte. Zwar konnten jetzt die Funktionen die Ergebnisse direkt zurückgeben und die If-Then-Else-Schachtelung fiel weg, aber die Relation Fehler-Code – Nutz-Code verbesserte sich nicht (sie verschlechterte sich sogar) und der Return-Code als Steuerungsmechanismus ging verloren. Allerdings standen mit dem Inhalt der Exceptions mehr Informationen zur Verfügung.

Die Verwendung von Exceptions durch Datenbanktreiber offenbarte ein weiteres Problem: Exceptions können nicht ignoriert werden. So wurde z.B. der Versuch, einen nicht vorhandenen Datensatz zu lesen, mit einer Exception quittiert. Diese musste dann in jedem Fall behandelt

werden, sollte sie nicht zu einem Abbruch des Programms führen. Gleichzeitig konnte die Service-Routine nicht entscheiden, ob das tolerabel war oder nicht. Daher musste wieder auf die Technik der Return-Codes zurückgegriffen werden.

Die Lösung: Eine Resultat-Klasse.

Mit dem Paradigmenwechsel zur Objektorientierung war schnell klar, dass man leicht eine geeignete Klasse selbst entwerfen konnte, doch es dauerte eine Weile, bis das geeignete Design gefunden war. Sie sollte eine Hilfe sein, kein weiterer Ballast. Sie sollte so attraktiv sein, dass sie von den Entwicklern auch angewendet wurde.

Da man Meta-Informationen zu einem Resultat transportieren wollte, lag es nahe, die Klasse „Result" zu nennen. Leider wurde in der Standard-Java-Library „Result" bereits für das Interface von Klassen, die ein Resultat einer XML-Transformation beinhalteten, vergeben. Um eventuelle Konflikte und Konfusion zu vermeiden wurde daher beschlossen, in „Result" die Vokale wegzulassen und damit mit „Rslt" einen eigenständigen Name zu kreieren.

Die Rslt Klasse

Die erste Version der Rslt-Klasse wurde 2003 kreiert und reifte dann von Jahr zu Jahr. Ihr recht einfacher Aufbau hat sich allerdings nicht geändert. Sie beinhaltet einen Integer und ein Array von Strings, das zu Beginn nicht initiiert ist. Der Integer dient dem Status, das Array dem Transport von Text.

```
// Die Rslt-Status
isOK = 0;       //Alles in Ordnung.
InError = 1;    // Ein Fehler wurde festgestellt.
IsInit = 2;     // Konnte erfolgreich initialisiert werden.
NotInit = 3;    // Das Ergebnis ist nicht initialisiert.
notFound = 4;   // Ein oder mehrere Werte konnten nicht gefunden werden.
IsEmpty = 5;    // Das Ergebnis der Verarbeitung ist leer.
IsDupli = 6;    // Die Operation würde ein Duplikat erzeugen.
IsAmbig = 7;    // Es wurde ein eindeutiges Ergebnis erwartet, aber
                // es liegen mehrere Möglichkeiten vor.
```

Text 3: Die Status der Rslt-Klasse

Die Rslt-Status sind vordefiniert, ihre Werte sind in der Klasse RsltStatVal hinterlegt, zusammen mit dem Label, das den Status in Klartext vorhält. Die letzten Jahre haben diese acht Status vollauf genügt, um den Zustand eines Objekts zu signalisieren.

Eine weitere Aufteilung in detaillierte Status wie bei Return-Codes wurde als nicht erstrebenswert erachtet, da dies einer universellen Verwendung entgegenstehen würde. (Vgl. auch das SQLRslt). In der Regel muss sowieso der Text der Fehlermeldung gelesen und interpretiert werden, um das Problem zu verstehen.

Im Prinzip sind es lediglich drei Funktionen, die die Rslt-Klasse vorhält: den Status des Rslt setzen, einen String anhängen und den Status abfragen.

Dem Komfort zuliebe und um den Mehrwert zu erhöhen hat die Rslt-Klasse allerdings tatsächlich bis jetzt 29 Funktionen.

Zwei zentrale Funktionen der Rslt-Klassen sind die Statusabfragen „isOK()“ und „notOK()“, wobei „notOK()“ einfach „!(isOK()) liefert“.

Der kleine, aber feine Unterschied zur Return-Code-Verarbeitung ist, dass das Objekt bzw. die Funktion entscheidet, in welchem Status das Resultat ist. Bei der Datenverarbeitung mit Return-Codes ist es die nutzende Funktion, die entscheiden muss, was der Return-Code bedeutet. Wenn die Entscheidung immer die gleiche ist, ist dieser Code in den nutzenden Funktionen redundant.

Um den Status abzufragen, verwendet man „getRsltStatVal()“, mit „isInStat(p_Stat)“ kann man den Status des Resultats gegen einen erwarteten Wert vergleichen.

Einer der Vorzüge der Rslt-Klasse (oder einer davon abgeleiteten Klasse) ist, dass das Objekt direkt nach der Konstruktion einen Status hat. Die weitere Verarbeitung kann damit ohne weitere Analyse direkt vom Zustand des Objekts abhängig gemacht werden.

```
RsltDB rDB = new RsltDB( <Access Information> );

if( rDB.isOK() ) { ...
```

Text 4: Status eines Objekts nach Instanziierung

Neben dem Default-Konstruktor kann man das Rslt im Status OK oder inError erzeugen oder die Informationen eines anderen Rslt übernehmen. Der Default-Status des Objekts ist „notInit“.

```
Rslt rOK = new Rslt( true );
Rslt rNotOK = new Rslt( false );
Rslt rNotInit = new Rslt();
Rslt rClone = new Rslt( aRslt );
```

Text 5: Konstruktoren der Rslt-Klasse

„checkRslt()“ ist ein weiterer zentraler Service, von dem es verschiedene Varianten gibt. „checkRslt()“ überprüft zunächst den eigenen Status und dann den Status des übergebenen Rslts. Sollte einer von beiden „nichtOK()“ sein, dann wird mit einem Boolean „False“ als Rückgabewert signalisiert, dass der Gesamtstatus nicht OK ist. Sollte das übergebene Rslt nicht OK gewesen sein, so wird dessen Fehlerstatus und Fehlerinformation übernommen. Macht man die weitere Verarbeitung vom zurückgegebenen Boolean abhängig, so wird im Fehlerfall die weitere Verarbeitung unterbunden.

```
RsltString getSomeData( ... ) {

        // Das wird das Ergebnis der Funktion, momentan leer aber "OK".
        RsltString rSomeDataString = new RsltString( true );

        RsltString someMoreData;
        RsltString evenMoreData;

        someMoreData = getSomeMoreData( ... );

        // Prüfen des Status, wenn OK, dann weiter. Wenn nicht,
        // Fehlerinformation übernehmen.
        if( rSomeDataString.checkRslt( someMoreData ) ) {

                evenMoreData = magicDBRead( someMoreData );

                if( rSomeDataString.checkRslt( evenMoreData ) ) {

                        rSomeDataString = magicDBRead( evenMoreData );
                }
        }

        return rSomeDataString
}
```

Text 6: Anwendung der Rslt-Klasse

Es kann außerdem die Möglichkeit bestimmt werden, „isEmpty“ als „isOK“ zu werten, und zwar in den verschiedensten Funktionen.

Besondere Erwähnung verdienen die Services zum Setzen des Rslt an Hand einer Exception. Sie extrahieren alle verfügbaren Informationen einer Exception und machen sie als Fehlerinformation im Rslt verfügbar.

Die Rslt Klasse geht davon aus, dass der Fehlerfall die Ausnahme ist, und verhält sich daher ökonomisch, solange kein Fehler aufgetreten ist. Im Fehlerfall wird diese Ökonomie zu Gunsten des Informationstransports aufgegeben.

Einsatz der Rslt Klasse

So sieht die Code-Instrumentierung mit Hilfe der Rslt-Klasse aus:

```
//Dies ist eine gegenüber dem tatsächlichen Code stark vereinfachte Version.
RsltString getLicenseString( string p_UserName ) {

        RsltString rLicenseString;
        RsltString rClientID;

        rLicenseString = new RsltString( true );

        rClientID = readFromDB( „UserName“, „ClientID“, p_UserName );

        if( rLicenseString.checkRslt( rClientID ) ) {

                rLicenseString = readFromDB( „ClientID“, „LicenseString“,
                                                        ClientID );
        }

        if( rLicenseString.notOK() ) {

                rLicenseString.addRsltAddInfo( „getLicenseString(): Failed to
                                retrieve 'LicenseString' from database for'.“
                                + p_UserName + “'“);
        }

        return rLicenseString;
}
```

Text 7: Instrumentierter Code, neu

Gegenüber dem ursprünglichen Code ist dieser Code nun deutlich aufgeräumter. Jetzt wird das Ergebnis der Verarbeitung und die Meta-Information über den Ausgang der Verarbeitung in einem Schritt übergeben.

Auch die Behandlung von Exceptions wird mit Hilfe der Rslt-Klasse deutliche vereinfacht. Die Rslt Klasse verarbeitet die Information aus der Exception und setzt den Status auf „notOK“.

```
//Dies ist eine gegenüber dem tatsächlichen Code stark vereinfachte Version.
RsltString readFromDB( string p_WhereCol, string p_ValCol, string p_Search ) {

        RsltString rResult = new RsltString();

        string val;

        try {

                val = SQLread( p_WhereCol, p_Search, p_ValCol );

                //Sets the Rslt to “OK”
                rResult.setString( val );
        }
        catch( Exception excp ) {

                //Sets the Rslt to “NotOK”
                rResult.setRsltByExcp( excp );

                //Zusätzliche fachliche Information zum Fehler
                rResult.addRsltAddInfo( „readFromDB(): Failed to read '“
                                        + p_ValCol + “' where '“
                                        + p_ WhereCol + “' is '“ + p_Search + “'“);
        }

        return rResult;
}
```

Text 8: Exception-Handling mit der Rslt-Klasse

Neben der technischen Information aus der Exception werden nun gleichzeitig die fachlichen

Informationen mitgeliefert. Der Empfänger hat nun die Möglichkeit, den Fehler zu verwerten oder ihn zu ignorieren und alternative Maßnahmen zu ergreifen.

Da jede Service Funktion in gleicher Weise ihren Code instrumentiert, entsteht im Fehlerfall eine Stack-Trace-ähnliche Liste der Funktionen, die bis zum Auftreten des Fehlers durchlaufen wurde.

Um den Fehlerstatus zu setzen, stehen die Funktionen „setInError(p_ErrorMessage)“, „setRsltStat(p_RsltStatus)“ und „setRsltStat(p_RsltStatus, p_ErrorMessage) zur Verfügung. „setInError(p_ErrorMessage)“ setzt den Status auf „inError“ und speichert den mitgegebenen Text.

Um den Status auf „isOK“ zu setzen bedient man sich „setOK()“. Diese lässt aber den Fehlertext unangetastet. Möchte man den Status des Rslts zurücksetzen und den Fehlertext löschen, so bedient man sich der Funktion „resetRslt()“.

Die Fehlerauswertung ist ebenfalls denkbar einfach: man gibt den Status und das Array mit den Meldungen in ein gewünschtes Log aus. Zur Ausgabe von kurzen Fehlermeldungen an die Benutzerschnittstelle dient die Funktion „getRsltLastMsg()“. Sie gibt die letzte Fehlermeldung aus der Liste der Fehlermeldungen aus.

Da Mehrfachvererbung in Java nicht erlaubt ist, steht auch ein Interface IRslt zur Verfügung, das es ermöglicht, abgeleitete Klassen zu Rslt-Klassen zu machen.

Setzt man die Rslt-Klasse konsequent ein, d.h. macht man eigene Klassen zu Ableitungen von Rslt und alle Rückgabewerte zu Ableitungen eines Results, so erreicht man mit minimalem Aufwand eine durchgängig einheitliche Fehlerbehandlung, die das Programm deutlich aufwertet. Vor allem die Nutzer eines so geschriebenen Programms wissen dies zu schätzen. Die qualifizierte Fehlermeldung, die das Programm präsentiert, vermittelt dem Benutzer ein besseres Gefühl gegenüber der Variante „Ein unbekannter Fehler ist aufgetreten...“. Er kann sich ein Bild von der Schwere des Fehlers machen und es eröffnet sich die Möglichkeit, den Fehler zu umgehen. Vor allem aber ist der Lieferant der Software in der Lage, durch Analyse des Fehlerlogs eine Ferndiagnose durchzuführen und damit die Lösung eines Problems zielgenau und in kürzester Zeit zu liefern.

Erfahrungen mit der Rslt Klasse

Heute sind alle unsere Klassen Ableitungen der Rslt Klasse. Fremde Klassen werden mit einem Rslt-Umschlag versehen. Durch Einschließen der verschiedenen Fehler-Informationsquelle in Rslt-Objekte ist eine einfache und einheitliche Fehlerbehandlung durchgängig möglich. 90% aller Fehler können alleine an Hand des erzeugten Logs analysiert und sicher behoben werden, was sich schon beim Unit- und Modul-Test auszahlt.

Die Akzeptanz bei den Entwicklern ist hoch, da der geforderte Aufwand niedrig und der Zugewinn an Information für die Fehleranalyse hoch ist.

Meldungen wie „An unexpected error has occurred...“ wird der Benutzer bei unseren Programmen nicht zu sehen bekommen, statt dessen die bestmögliche Analyse der Ursache und damit die bestmögliche Grundlage für die Fehleranalyse und -behebung.

Auch wenn die RsIt-Klasse eine vergleichsweise einfache Lösung ist, der Einfluss auf die Produktivität ist enorm. Der entscheidende Vorteil stellt sich im Falle eines Fehlers ein: die gute Fehleranalyse, die das Programm von sich aus liefert, verringert den Aufwand der Fehlerbehebung. Schon die im Unit-, Modul- und Systemtest dadurch gewonnene Zeit macht den zusätzlichen Aufwand zur Instrumentierung des Codes mehr als wett.

Der verringerte Aufwand der Fehleranalyse ermöglicht aber auch verkürzte Turn-Around-Zeiten bei Incidents in der Produktion. Dies wiederum stärkt das Vertrauen des Kunden in das Produkt und in die Entwickler der Software.

Ableitungen der RsIt Klasse

Die am häufigsten gebrauchten Klassen, die ein Resultat darstellen, wurden gleich dem RsIt-Paket mitgegeben:

RsItString

RsItString ist ein String, dem man entnehmen kann, wie sein Zustand zu bewerten ist. Mit „setString(String p_String)“, „appString(String p_String)" und "getString()" erreicht man den Inhalt des Strings. Sein Ausgangszustand ist "isEmpty" und da die Frage nach diesem Zustand recht häufig gebraucht wird, wurde die Funktion "isEmpty()" implementiert.

Mit der RsItString gibt es nun kein Rätselraten mehr über die Bedeutung eines Leer-Strings (Wurde nichts gefunden? Oder ist das der gespeicherte Wert?).

RsItLst

RsItLst ist eine Ableitung der ArrayList. Sie implementiert das Interface IRsIt. Ansonsten genießt sie die gleichen Vorzüge wie alle RsIt Klassen.

SQLRsIt

Das SQLRsIt ist eine spezielle Klasse für Aktionen und Ergebnisse von SQL Datenbanken. Neben dem ererbten RsIt-Status führt sie einen separaten Fehlercode mit, der aus der SQL Exception ermittelt wird, wenn der Status per setRsItByExcp(java.sql.SQLException p_Excp) gesetzt wird. Um diesen Status auszulesen steht getLastErrorCode() zur Verfügung.

XerrorHandler

Die XErrorHandler-Klasse implementiert das org.xml.sax.ErrorHandler Interface. Die Standard SAX ErrorHandler Methoden werden zur Verfügung gestellt und wandeln die Ergebnisse in RsIt-Werte um.

Alternative Anwendungen des RsIt-Konzepts

Da die RsIt-Klasse auf keiner Sprachen-spezifischen Eigenheit basiert, kann sie praktisch in jeder Objekt-orientierten Sprache verwendet werden. Aber die Prinzipien der RsIt Klasse sind nicht darauf beschränkt, mittels OO Sprachen umgesetzt zu werden. Ebenso erfolgreich wie in C# und Java

wurde das Konzept in JavaScript und SilkTest, aber auch in XML- und JSON-Nachrichten genutzt.

Download

Der Source-Code der Rslt-Klasse ist frei verfügbar und steht unter der GNU Lesser General Public License. Er kann unter http://www.qa-navigation.com/de/Downloads.html heruntergeladen werden.