Exceptions einmal anders

Axel T. Schreiner und Bernd Kühl
Fachbereich Mathematik/Informatik, Universität Osnabrück

{axel,bekuehl}@uos.de

Exceptions enthält man Anfängern gerne vor, so nach dem Motto: ‘‘C kam doch auch prima ohne sie aus’’. Andrerseits sind Exceptions ein sehr wesentliches Ausdrucksmittel in Java: Immer wenn sich eine Methode ärgert, reagiert sie mit einer Exception. Wenn diese nicht von `RuntimeException` abstammt, ist der Aufrufer der Methode definitiv gezwungen, sich mit dem Problem auseinanderzusetzen – entweder, indem er die Exception mit `try..catch` abfängt oder, indem er in seinem eigenen Methodenkopf mit `throws` davor warnt, daß er die Exception verursachen könnte.

Exceptions sind allerdings recht endgültig. Schickt man eine mit `throw` ab, kann man unbedingt davon ausgehen, daß man das Problem los ist, denn nach einer `throw`-Anweisung wird die Programmausführung unbedingt dort fortgesetzt, wo die Exception mit `try..catch` aufgefangen wird. Waren zwischen dem `try`-Block und der `throw`-Anweisung noch Methodenaufrufe verschachtelt, dann werden diese bei `catch` entsorgt. Aus dieser Perspektive entsprechen Exceptions in Java der `setjmp`-Konstruktion in C.

Kann man eine Exception probehalber abschicken? Das heißt, kann man von einer Methode aus entlang der Aufrufverschachtelung von innen nach außen eine Exception anbieten, so daß jede beteiligte Methode die Exception selbst abfangen, sie nach außen weiterleiten oder auch verlangen kann, daß die Exception wieder entlang der Aufrufverschachtelung nach innen bis hin zum ursprünglichen Verursacher zurückgeschickt wird? Dabei könnten globalere Randbedingungen von den Beteiligten zwischenzeitlich so modifiziert werden, daß die Exception möglicherweise nicht mehr nötig ist. Im Jargon spricht man von *resume*, Fortsetzung am Punkt, nach dem ein Fehler auftrat, und *retry*, Wiederholung einer Operation, die einen Fehler verursachte. In einem Artikel über das Design von C++ schrieb Bjarne Stroustrup, daß er sich für Terminierung bei Exceptions entschied, da er die korrekte Implementierung der anderen Möglichkeiten für seine mitstreitenden Compiler-Entwickler für zu schwierig hielt...

Zwar hat sich auch James Gosling in Java für Terminierung entschieden, aber man kann die Alternativen relativ leicht selbst schaffen. Will eine Methode die Kontrolle über eine Exception endgültig übernehmen, muß sie natürlich `try..catch` bemühen, um etwaige Trümmer zu entsorgen. Damit man sich zuvor entlang der Aufrufverschachtelung nach außen und zurück nach innen bewegen kann, muß man die Aufrufe mit rückwärts verketteten Objekten modellieren, die entlang und während der Aufrufe aufgebaut werden. Nach außen bewegt man sich dann durch eine Nachricht an ein vorhergehendes Objekt; zurück nach innen bewegt man sich, wenn der Methodenaufruf dieser Nachricht beendet wird. Besteht die Kette aus `Throwable`-Objekten, kann jedes das Auf und Ab mit `throw this` abwürgen. Jedes solche Objekt modelliert einen Methodenaufruf; in den Methoden kann man `try..catch` so organisieren, daß der zum Objekt gehörende Methodenaufruf dann die Kontrolle endgültig wiedergewinnt. Das alles funktioniert auch dann, wenn Rekursion, verschiedene Methoden und verschiedene Objekte gleicher oder verschiedener Klassen beteiligt sind.

```
public abstract class Activation extends Throwable {
```

```
  private final Activation caller;
  protected Activation (Activation caller) { this.caller = caller; }

  public Object raise (Object info)
                        throws Activation, NullPointerException {
    return caller.raise(info);
  }

  protected final void handle (Object info) throws Activation {
    this.info = info;
    throw this;
  }

  private Object info;
  public final Object info () { return info; }
}
```

Abbildung 1: Basisklasse zum Modellieren von Methodenaufrufen

Abbildung 1 zeigt eine Basisklasse `Activation`, mit der Methodenaufrufe als Kette modelliert werden können. Ein `Activation`-Objekt verkapselt einen Zeiger `caller` auf seinen Vorgänger in der Kette. `raise(info)` leitet ein beliebiges Objekt `info`, also zum Beispiel eine Exception, an den Vorgänger in der Kette weiter, sofern einer vorhanden ist, und ist die einzige Zugriffsmöglichkeit zu diesem Vorgänger. `handle(info)` speichert einen Verweis, der mit `info()` wieder abgeholt werden kann; diese beiden Methoden sind dazu vorgesehen, daß der vom Kettenelement modellierte Methodenaufruf endgültig die Kontrolle erhält und dann `info` übernimmt.

```
... method (Activation caller, ...) throws Activation {

  final class MyActivation extends Activation {
    MyActivation (Activation caller) { super(caller); }

    ...  lokale Variablen für method() mit Zugriff für raise()

    public Object raise (Object info) throws Activation {
      ...  entscheidet, wie mit info verfahren werden soll
    }
  };
  final Activation self = new Activation(caller);

   try {

    ...  hier können andere Methoden aufgerufen werden und Probleme auftreten

  } catch (Activation a) {
    if (a != self) throw a; Object info = self.info();

    ...  hier wird info endgültig verarbeitet, wenn raise() das mit handle() verlangt

  }
}
```

Abbildung 2: Struktur zur kontrollierten Verarbeitung von Problemen

Wie eine Methode strukturiert sein muß, damit das funktioniert, ist aus Abbildung 2 ersichtlich. `method()` erhält ein `Activation`-Objekt `caller`, das den Aufrufer beschreibt, und setzt seinerseits die Kette mit `self` fort. `self` gehört zu einer lokalen Unterklasse von `Activation`, in der `raise()` überschrieben wird.

In einer lokal definierten Klasse hat man Zugriff auf Instanzvariablen sowie auf `final`

vereinbarte lokale Variablen und Parameter der Methode, die die Klasse enthält. Macht man die Klasse, wie hier `MyActivation`, nicht anonym, kann man dort lokale Variablen, die `method()` und `raise()` gemeinsam bearbeiten sollen, als paket-öffentliche, aber nur lokal erreichbare Instanzvariablen unterbringen – `final` ist dann nicht nötig.

`try`..`catch` umgibt Bereiche von `method()`, in denen man mit Problemen rechnet. Dort kann man andere Methoden aufrufen, wobei man die Modellierung der Aufrufverschachtelung fortsetzen sollte, indem man den beteiligten Methoden `self` als Argument übergibt. Entdeckt man ein Problem, schickt man die nötige Information mit `raise()` an sich selbst oder seinen Aufrufer; im Falle von `method()` also an `self` oder `caller`.

`raise()` empfängt die Information und entscheidet, was geschehen soll. Im wesentlichen gibt es drei Möglichkeiten: `super.raise()` schickt die Information entlang der Aufrufkette weiter nach außen, `handle()` schickt sie an den zugehörigen Methodenaufruf – in der Abbildung also an den `catch`-Bereich in `method()` – und `return` schickt sie entlang der Aufrufkette wieder zurück nach innen. `handle()` entspricht folglich der konventionellen Bearbeitung von Exceptions mit Terminierung; `return` führt letztlich zu *resume* oder *retry*, je nachdem, wie sich der ursprüngliche Erzeuger der Information zum Schluß selbst entscheidet – per Konvention könnte man das auch von außen steuern, indem man Null statt der Information zurückschickt.

`handle()` verwendet eine `throw`-Anweisung, siehe Abbildung 1. Damit `info` im richtigen `catch`-Block verarbeitet wird, muß jeweils kontrolliert werden, daß nur das eigene `Activation`-Objekt berücksichtigt wird; andere reicht man mit `throw` weiter, siehe Abbildung 2. Da jeder Aufruf von `method()` ein eigenes `Activation`-Objekt `self` erzeugt, wird `info` selbst bei rekursiven Aufrufen der korrekten Aktivierung zugestellt.

```
A       method  A.. or raise, else return: A
AA      method  A.. or raise, else return: B
AAB     method  A.. or raise, else return:
        method  quits
AA      method  A.. or raise, else return: raise
AA      raise   super or handle, else return: super
A       raise   super or handle, else return: super   Fehler!
A       raise   super or handle, else return:
        raise   quits
AA      raise   super or handle, else return: handle
        method  handling AA
AA      method  A.. or raise, else return:
        method  quits
A       method  A.. or raise, else return:
        method  quits
```

Abbildung 3: Ablaufbeispiel am Terminal

Als Test kann man in `method()` und `raise()` anfragen und interaktiv entscheiden lassen, was geschehen soll. Abbildung 3 zeigt ein Beispiel, in dem einige Objekte angelegt wurden und zunächst `method()` für das erste Objekt aufgerufen wurde. Interaktiv wird dann `method()` für das erste Objekt rekursiv aufgerufen und dann für das zweite Objekt aufgerufen und beendet. Im rekursiven Aufruf wird danach mit `self.raise()` ein Problem aufgeworfen und interaktiv beiden verschachtelten Aufrufen angeboten. Damit ist der Anfang der Aufrufkette erreicht, ein weiterer Aufruf von

`super.raise()` endet mit einem Fehler, der hier ignoriert wird. Im weiteren Verlauf reicht `raise()` das Problem zurück, und es wird dann doch durch `handle()` in den `catch`-Bereich des rekursiven Aufrufs von `method()` beim ersten Objekt geliefert. Im Testprogramm enthält `method()` eine Schleife; *resume* und *retry* sind hier nicht zu unterscheiden.

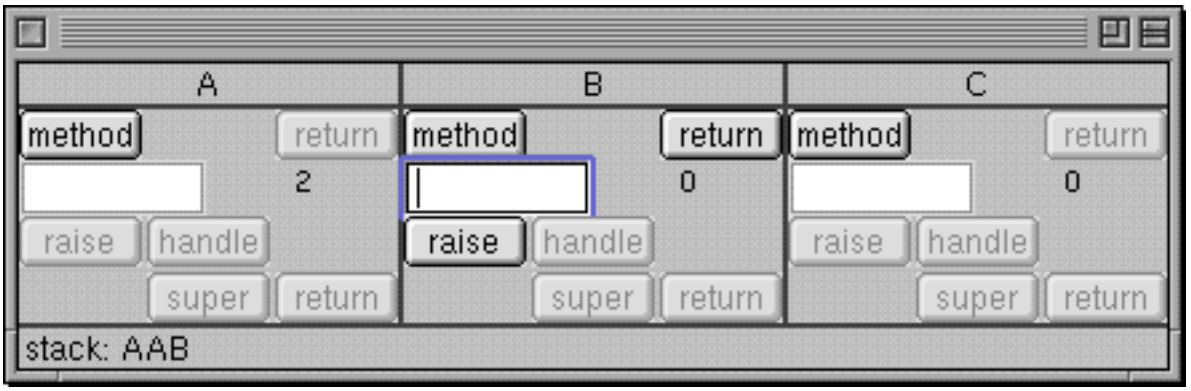

Abbildung 4: Ablaufbeispiel mit Oberfläche

Entwirft man den Test strikt nach dem Model-View-Controller-Schema, kann man ihn unverändert auch im *main*-Thread einer Java-Applikation ablaufen lassen und die Eingaben durch Synchronisation mit dem *event*-Thread von einer Oberfläche beschaffen. Abbildung 4 zeigt ein Beispiel, bei dem in jedem der drei ‘‘Objekte’’ nur die aktuell möglichen Aktivitäten freigeschaltet sind. Durch Druck auf einen Knopf in einem Objektbereich wird die entsprechende Methode bei dem Objekt aufgerufen. `method()` und `raise()` teilen sich hier noch eine lokale Variable: eine laufende Summe, die durch Eingaben im Textfeld innerhalb von beiden Methoden beeinflußt werden kann. Damit kann man zum Beispiel die rekursiven Aufrufe von `method()` beim gleichen Objekt unterscheiden.

Klappt diese Programmiertechnik in anderen Sprachen als Java? Im Prinzip schon, wenn man wenigstens wie in C mit den in *setjmp.h* definierten Makros die Möglichkeit hat, eine Aufrufverschachtelung abzuschneiden. Javas innere Klassen sind allerdings syntaktisch besonders elegant, um gemeinsamen Zugriff auf lokale Variablen zwischen `method()` und `raise()` einzurichten, und Javas Thread-Modell für Applikationen mit grafischer Oberfläche macht es besonders leicht, ein Testprogramm auch grafisch bedienen zu lassen. Java hat bekanntlich viele Aspekte eher von Objective C als von C++ übernommen, aber die Implementierung der analogen Funktionalität ist in Objective C deutlich unschöner. Interessenten können den Code für Java und Objective C von ... beziehen; *class*-Dateien liegen bei, denn auch JDK 1.1.7 kann einen trivialen Aspekt des Tests nicht korrekt übersetzen.