PEARSON Education

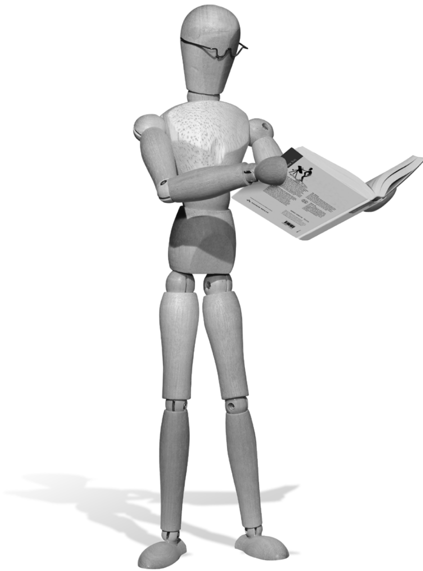
ADDISON-WESLEY

open source library

George Schlossnagle

Professionelle PHP 5-Programmierung

Entwicklerleitfaden für große Webprojekte mit PHP 5



An imprint of Pearson Education

München • Boston • San Francisco • Harlow, England
Don Mills, Ontario • Sydney • Mexico City
Madrid • Amsterdam

3 Fehlerbehandlung

Fehler sind ein Bestandteil des Lebens. Mister Murphy hat – wie Sie wissen – eine detaillierte Sammlung über die Verbreitung und Gesetzmäßigkeit von Fehlern zusammengestellt. Beim Programmieren tauchen Fehler in zweierlei Hinsicht auf:

- **Externe Fehler** – Das sind Fehler, in denen der Programmablauf einen nicht vorhergesehenen Weg einschlägt, weil ein Teil des Programms nicht so agiert wie gedacht. Wenn zum Beispiel eine erforderliche Datenbankverbindung fehlschlägt, ist dies ein externer Fehler.
- **Logische Programmfehler** – Diese Fehler, die üblicherweise als Bugs bezeichnet werden, sind Fehler, bei denen der Code grundsätzliche Fehler aufweist: entweder wegen falscher Logik (»so geht es einfach nicht«) oder einfach aufgrund eines Tippfehlers.

Diese beiden Fehlerkategorien unterscheiden sich in mehrerer Hinsicht:

- Externe Fehler werden immer auftauchen, egal wie fehlerfrei der Code ist. Dabei handelt es sich nicht um Bugs im engen Sinne, da die Ursache außerhalb des Programms liegt.
- Externe Fehler, die im Code nicht abgefangen werden, können Bugs sein. Zum Beispiel blind anzunehmen, dass eine Datenbankverbindung immer erfolgreich aufgebaut werden kann, ist ein Bug, weil das Programm in diesem Fall mit Sicherheit nicht korrekt reagiert.
- Logische Programmfehler sind viel schwerer zu entdecken als externe Fehler, weil – per Definition – ihre Ursache nicht bekannt ist. Allerdings können Sie Konsistenztests implementieren, um diese Fehler aufzuspüren.

PHP unterstützt Fehlerbehandlung und hat ein eingebautes Schweregrad-System, mit dem Sie nur die Fehler sehen, die Sie wirklich ernsthaft betreffen. Dabei unterscheidet PHP die folgenden drei Kategorien:

- `E_NOTICE`
- `E_WARNING`
- `E_ERROR`[1]

1 Anm. d. Fachl.: Seit PHP 5 gibt es noch eine weitere Kategorie, E_STRICT. Diese warnt auch bei der Verwendung von Konstrukten, die nur aus Gründen der Abwärtskompatibilität zu PHP 4 noch in PHP 5 erlaubt sind.

E_NOTICE Fehler sind unbedeutend, aber sie können Ihnen helfen, Hinweise auf eventuell vorhandene Bugs zu liefern. Generell stören solche Fehler nicht den Ablauf des Programms, aber das Resultat entspricht eventuell nicht Ihren Vorstellungen. Ein Beispiel wäre, eine Variable in einem nicht zuweisenden Ausdruck zu verwenden, bevor dieser Variablen ein Wert zugewiesen wurde:

```
<?php
    $variable++;
?>
```

Dieses Beispiel inkrementiert $variable auf 1 (da Variablen als 0/false/Leerstring initialisiert werden), aber es wird einen E_NOTICE-Fehler hervorrufen. Stattdessen sollten Sie den Code so schreiben:

```
<?php
    $variable = 0;
    $variable++;
?>
```

Dieser Test soll Tippfehler in Variablennamen verhindern. Der nächste Codeschnipsel arbeitet fehlerfrei:

```
<?php
    $variable = 0;
    $variabel++;
?>
```

$variable wird nicht inkrementiert, aber $variabel. Eine E_NOTICE-Warnung macht Sie auf diese Art der Fehler aufmerksam. Sie sind vergleichbar mit der Ausführung von use warnings und use strict in Perl oder mit dem Kompilieren eines C-Programms mit -Wall.

In PHP sind E_NOTICE-Fehler standardmäßig ausgeschaltet, da sie ziemlich große Log-Dateien erzeugen. In meinen Programmen ziehe ich es – während der Entwicklung – vor, diese Hinweise einzuschalten, um eine Unterstützung beim Säubern des Codes zu erhalten und sie dann wieder zu deaktivieren.

E_WARNING-Fehler sind keine schweren Laufzeitfehler. Sie stoppen oder ändern den Fluss des Programms nicht, aber sie signalisieren, dass etwas Unfreundliches aufgetreten ist. Viele externe Fehler generieren E_WARNING-Fehler. Ein Beispiel ist, dass man einen Fehler beim Aufruf von fopen() oder mysql_connect() erhält.

E_ERROR-Fehler sind schwere Fehler, die die Ausführung des Codes abbrechen. Sie tauchen beispielsweise bei dem Versuch auf, eine nicht existierende Klasse zu initialisieren.

In PHP gibt es die Funktion trigger_error(), die es dem Programmierer erlaubt, eigene Fehler im Skript zu erstellen. Drei Fehlertypen können ausgelöst werden. Sie besitzen die gleiche Semantik, wie die oben beschriebenen:

open source library

- E_USER_NOTICE
- E_USER_WARNING
- E_USER_ERROR

Sie können diese Fehler folgendermaßen auslösen:

```
while(!feof($fp)) {
  $line = fgets($fp);
  if(!parse_line($line)) {
    trigger_error("Incomprehensible data encountered",
                  E_USER_NOTICE);
  }
}
```

Wenn kein Fehlergrad spezifiziert ist, wird E_USER_NOTICE verwendet.

Zusätzlich zu diesen Fehlern gibt es fünf weitere Kategorien, die eher selten auftreten:

- E_PARSE – Im Skript ist ein Syntaxfehler und konnte nicht verarbeitet werden. Dies ist ein schwerer Fehler.
- E_COMPILE_ERROR – Ein schwerer Fehler ist während des Kompilierens des Skripts aufgetreten.
- E_COMPILE_WARNING – Ein weniger schwerer Fehler in der PHP-Engine ist während des Verarbeitens des Skripts aufgetreten.
- E_CORE_ERROR – Ein schwerer Laufzeitfehler ist in der PHP-Engine aufgetreten.
- E_CORE_WARNING – Ein weniger schwerer Fehler ist in der PHP-Engine aufgetreten.

Zusätzlich verwendet PHP die E_ALL-Fehlerkategorie für alle Stufen des Fehlerberichts. Sie können den Schweregrad der Fehler, die zu Ihrem Skript durchsickern, durch den Eintrag error-reporting in der php.ini steuern. Error-reporting ist ein Bit-Feld, das definierte Konstanten wie die folgende für alle Fehler benutzt:

```
error_reporting = E_ALL
```

Error_reporting verwendet den folgenden Code für alle Fehler, außer für E_NOTICE, das mit einem exklusivem ODER von E_ALL und E_NOTICE erreicht wird:

```
error_reporting = E_ALL ~ E_NOTICE
```

Ganz ähnlich wird error_reporting nur für schwere Fehler eingesetzt (bitweises ODER der beiden Fehlertypen):

```
error_reporting = E_ERROR | E_USER_ERROR
```

Beachten Sie, dass das Entfernen von E_ERROR aus dem error_reporting Ihnen nicht erlaubt, schwere Fehler zu ignorieren. Es verhindert lediglich, dass die Behandlungsroutine für Fehler aufgerufen wird.

3.1 Fehlerbehandlung

Nachdem Sie gesehen haben, welche Fehlerarten PHP generiert, sollten Sie einen Plan entwickeln, wie Sie mit auftretenden Fehlern umgehen. PHP bietet im Rahmen der Fehlerbehandlung vier Möglichkeiten:

- Anzeigen
- Protokollieren
- Ignorieren
- Reagieren

Keine dieser Optionen übertrifft die anderen an Wichtigkeit oder Funktionalität, da jede einen wichtigen Platz in einer soliden Fehlerbehandlung einnimmt. Die Anzeige von Fehlern bietet sich insbesondere Entwicklungsumgebungen an, und Fehler zu protokollieren, ist im echten Einsatz eines Codes sehr wichtig. Einige Fehler können getrost ignoriert werden, andere erfordern Reaktionen. Der richtige Mix der Fehlerbehandlungstechniken hängt von den jeweiligen Bedürfnissen ab.

3.1.1 Fehler anzeigen

Wenn Sie sich dazu entscheiden, Fehler anzuzeigen, wird eine Fehlermeldung an die Standardausgabe geschickt. Im Fall einer Webseite bedeutet dies, dass sie zum Browser gesendet wird. Sie ändern diese Einstellung über die folgende Einstellung in der `php.ini`:

```
display_errors = On
```

`display_error` ist sehr hilfreich bei der Entwicklung, weil Sie ein sofortiges Feedback darüber zu erhalten, was im Skript schief läuft. Es reicht die bearbeitete Webseite aufzurufen, ohne eine Log-Datei durchzuarbeiten

Was für Entwickler wichtig ist zu sehen, ist oftmals nicht für die Augen des End-Users gedacht. Die PHP-Fehlermeldungen für die End-User anzuzeigen ist normalerweise aus drei Gründen nicht wünschenswert:

- Es sieht hässlich aus.
- Es vermittelt den Eindruck, dass die Seite fehlerhaft ist.
- Es kann Einblicke in das Skript gewähren, die ein User eventuell missbrauchen kann

Der dritte Punkt kann nicht genug betont werden. Wenn Sie zusehen wollen, wie Sicherheitslücken in Ihrem Code gefunden und ausgenutzt werden, gibt es keinen schnelleren Weg als `display_errors` im laufenden Betrieb eingeschaltet zu haben. Ich habe einmal miterlebt, wie eine schlechte INI-Datei für einige Stunden auf einer Site mit viel Traffic verwendet wurde. Sobald dies bemerkt wurde, wurde die korrigierte Datei auf den Webserver kopiert und wir alle dachten, dass der Schaden lediglich unsere Eitelkeit verletzt hatte. Anderthalb Jahre später verfolgten und erwischten wir einen Hacker, der bösartig andere Mitgliedsseiten verunstaltet hatte. Als Gegenleis-

open source library

tung dafür, dass wir ihn nicht anzeigten, legte er alle Sicherheitslücken offen, die er gefunden hatte. Zusätzlich zu den bekannten Sicherheitsrisiken von JavaScript (es war eine Seite, die viel Inhalte der User zuließ) waren ihm einige besonders schlaue Hacks gelungen, die er aufgrund der Informationen geschrieben hatte, die für nur wenige Stunden ein Jahr zuvor auf der Webseite erschienen waren.

Wir hatten in diesem Fall Glück. Hauptsächlich hatte er nicht validierte Userangaben und nicht standardisierte Variablen (das war in den Tagen, bevor `register_globals` eingeführt wurde) entdeckt. Alle unsere Datenbankverbindungs-Informationen lagen in Bibliotheken und nicht auf der Webseite. Aber viele Webseiten sind ernsthaft durch eine Kette von Sicherheitslücken angegriffen worden:

- Eingeschaltete `display_errors`
- Informationen zur Datenbankverbindung (`mysql_connect()`) auf der Seite.
- Zulassung von nicht lokalen Verbindungen zur MySQL-Datenbank.

Mit diesen drei Fehlern zusammen landet Ihre Datenbank in den Händen eines jeden, der eine Fehlerseite auf Ihrer Webseite sieht. Sie werden (hoffentlich) geschockt sein, wie oft dies passiert.

Ich lasse `display_errors` während der Entwicklung an, aber danach schalte ich es aus.

Anzeigen von Fehlern im Betrieb

Wie User über Fehler informiert werden, hängt von der jeweiligen Firmenphilosophie ab. Bei allen meiner größeren Kunden gab es strikte Regeln darüber, wie zu verfahren ist, wenn ein User auf einen Fehler stößt. Diese Regeln reichten vom Anzeigen angepasster, themenbezogener Fehlerseiten bis hinzu komplexen Systemen, durch die gespeicherte Seiten angezeigt wurden, die dem vom User gesuchten Inhalt entsprachen. Ihre Webseite ist das Fenster zum Kunden, und jeder Fehler darin trübt das Ansehen Ihres Unternehmens.

Unabhängig von dem genauen Inhalt, der dem User im Falle eines unerwarteten Fehlers angezeigt werden muss, ist das letzte, was er sehen soll, ein wildes Durcheinander von Debugging-Informationen. Je nach Menge der Informationen auf Ihren fehlerhaften Seiten, würde dies zu einer unerwünschten Offenlegung von Informationen führen.

Ein relativ normales Verfahren ist es, einen Fehlercode 500 von der Seite zurückzugeben und den User über eine angepasste Fehlerbehandlung auf eine entsprechend angepasste Fehlerseite zu leiten. Ein Fehlercode 500 in HTTP zeigt an, dass ein interner Serverfehler aufgetreten ist. Um diesen Fehler mit PHP zu senden, verwenden Sie:

```
header("HTTP/1.0 500 Internal Server Error");
```

In der Apache-Konfiguration setzen Sie:

```
ErrorDocument 500 /custom-error.php
```

open source library

Dies führt dazu, dass jede Seite, die den Statuscode 500 zurückgibt, intern auf /custom-error.php umgeleitet wird. Im Abschnitt »Einrichten eines Top-Level-Exception-Handlers« weiter hinten in diesem Kapitel werden Sie für die Fehlerbehandlung eine alternative ausnahmebasierte Technik sehen.

3.1.2 Fehler protokollieren

PHP unterstützt sowohl das Verfahren, Fehler in einer Datei zu protokollieren als auch syslog einzusetzen. Über zwei Einträge in der php.ini steuern Sie das Verhalten. Die folgende Einstellung schaltet das Fehlerprotokoll ein:

```
log_errors = On
```

Die folgenden zwei Einstellungen bestimmen, wohin das Protokoll geschrieben wird: in eine Datei oder in syslog:

```
error_log = /path/to/filename

error_log = syslog
```

Protokollieren bietet eine gut zu verfolgende Spur zu jedem Fehler, den Ihre Webseite ausgibt. Wenn ich einem Problem auf der Spur bin, platziere ich oft debugging-Zeilen in den in Frage kommenden Bereich.

Zusätzlich zu den Fehlern, die vom System oder über trigger_error() protokolliert werden, können Sie manuell einen Eintrag in der Log-Datei erzeugen:

```
error_log("This is a user defined error");
```

Alternativ können Sie eine E-Mail verschicken oder die Ausgabe-Datei bestimmen. Werfen Sie für genauere Informationen einen Blick in das PHP-Manual. error_log() protokolliert die übergebene Nachricht unabhängig von der Einstellung error_reporting. error_log und error_reporting sind zwei komplett unterschiedliche Einstiege in die Möglichkeiten, Fehler zu protokollieren.

Wenn Sie nur einen einzigen Server haben, sollten Sie direkt in eine Datei protokollieren. Der Weg über syslog ist ziemlich langsam, und wenn bei jeder Skriptausführung eine Menge protokolliert werden muss (was sowieso keine gute Idee ist), kann der Aufwand beträchtlich werden.

Bei der Arbeit mit mehreren Servern stellt syslog einen angenehmen Weg bereit, die Log-Dateien von mehreren Maschinen zur Analyse und Archivierung in Echtzeit zusammenzuführen. Sie sollten exzessives Protokollieren jedoch vermeiden, wenn Sie vorhaben syslog zu nutzen.

open source library

3.1.3 Fehler ignorieren

In PHP können Sie selektiv mit der @-Syntax das `error_reporting` unterdrücken. Wenn Sie beispielsweise versuchen, eine eventuell nicht existierende Datei zu öffnen und Sie die Fehlermeldung unterdrücken wollen, können Sie folgende Zeile schreiben:

```
$fp = @fopen($file, $mode);
```

Die Möglichkeiten in PHP auf Fehler zu reagieren, sehen nicht vor, den Programmfluss zu beeinflussen. Daher bietet es sich an, Fehler zu unterdrücken, von denen Sie wissen, dass sie auftreten werden (die aber nicht stören).

Betrachten Sie eine Funktion, die den Inhalt einer Datei, die eventuell nicht existiert, zurückgibt:

```
$content = file_get_content($sometimes_valid);
```

Wenn die Datei nicht existiert, erhalten Sie einen `E_WARNING`-Fehler. Erwarten Sie dies als mögliches Resultat, sollten Sie diese Warnung unterdrücken (weil Sie sie erwartet haben) – es ist nicht wirklich ein Fehler. Auch bei Aufrufen von benutzerdefinierten Funktionen unterdrücken Sie die Fehlermeldung mit @:

```
$content = @file_get_content($sometimes_valid);
```

Wenn Sie in der `php.ini` den Eintrag `track_errors = on` setzen, wird die letzte Fehlermeldung in `$php_errormsg` gespeichert. Dies geschieht unabhängig davon, ob Sie das @-Zeichen zur Fehlerunterdrückung verwendet haben oder nicht.

3.1.4 Auf Fehler reagieren

In PHP können Sie eine selbst gestrickte Fehlerbehandlung über die Funktion `set_error_handler()` erreichen. Definieren Sie dazu folgende Funktion:

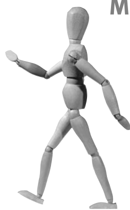
```
<?php
require "DB/Mysql.inc";
function user_error_handler($severity, $msg, $filename, $linenum) {
   $dbh = new DB_Mysql_Prod;
   $query = "INSERT INTO errorlog
             (severity, message, filename, linenum, time)
             VALUES(?,?,?,?, NOW())";
   $sth = $dbh->prepare($query);
   switch($severity) {
   case E_USER_NOTICE:
      $sth->execute('NOTICE', $msg, $filename, $linenum);
      break;
   case E_USER_WARNING:
      $sth->execute('WARNING', $msg, $filename, $linenum);
      break;
   case E_USER_ERROR:
      $sth->execute('FATAL', $msg, $filename, $linenum);
```

```
      print "FATAL error $msg at $filename:$linenum<br>";
      break;
   default:
      print "Unknown error at $filename:$linenum<br>";
      break;
   }
}
?>
```

Folgendermaßen setzen Sie eine Funktion:

```
set_error_handler("user_error_handler");
```

Wenn jetzt ein Fehler auftritt, wird er – anstatt angezeigt oder in der Log-Datei ausgegeben zu werden – in die Datenbanktabelle für Fehler eingefügt. Im Fall eines schweren Fehlers, wird auf dem Bildschirm eine Nachricht ausgegeben. Vergessen Sie nicht, dass die Fehlerbehandlung keine Programmflusskontrolle bereitstellt. Handelt es sich um einen leichten Fehler, wird das Skript nach der Ausführung der Fehlerbehandlung an der Stelle fortgesetzt, an der der Fehler auftrat. Im Fall eines schweren Fehlers, bricht das Skript nach der Fehlerbehandlung ab.



Mails an Sie selbst

Es klingt nach einer guten Idee, in einer selbst gestrickten Fehlerbehandlung die Funktion `mail()` zu verwenden, um eine Nachricht an den Entwickler oder den Systemadministrator zu schicken, wenn ein Fehler auftritt. Aber in der Regel sollten Sie die Finger davon lassen.

Fehler treten meistens gehäuft auf. Es würde funktionieren, wenn garantiert wäre, dass der Fehler nur einmal pro Stunde auftritt (oder irgendeine andere Zeitperiode). Aber in der Regel sind viele Seitenaufrufe betroffen, wenn ein unerwarteter Fehler aufgrund von Fehlern im Skript auftritt. Das bedeutet, dass Ihre E-Mail-Funktion `error_handler()` eventuell 20.000 E-Mails an Ihr Konto schickt, bevor Sie in der Lage sind einzugreifen und die Funktion zu deaktivieren. Nicht anzuraten, oder?

Wenn Sie diese Art der Funktionalität bei der Fehlerbehandlung brauchen, empfehle ich, ein Skript zu schreiben, das die Fehlermeldungen verarbeitet und auf intelligente Art die Anzahl der E-Mails begrenzt.

3.2 Behandlung externer Fehler

Obwohl wir alles bisherige in diesem Kapitel als Fehlerbehandlung bezeichnet haben, haben wir nicht wirklich gehandelt. Wir haben die Meldungen akzeptiert und verarbeitet, aber waren nicht in der Lage, den Programmablauf zu beeinflussen. Das heißt:

open source library

Wir haben die Fehler nicht wirklich behandelt oder abgefangen. Eine angepasste Fehlerbehandlung setzt voraus, dass man sich klar macht, wo der Fehler im Code liegen kann und entscheidet, wie im Falle des Falles der Fehler abgefangen werden soll.

Externe Fehler treten hauptsächlich im Zusammenhang mit der Verbindung zu externen Prozessen oder dem Extrahieren von Daten aus diesen Prozessen auf.

Betrachten Sie die folgende Funktion, die die Details aus der Datei `passwd` (Home-Verzeichnis, verwendete Shell, gecos-Informationen) für einen beliebigen User zurückgeben soll:

```
<?php
function get_passwd_info($user) {
    $fp = fopen("/etc/passwd", "r");
    while(!feof($fp)) {
        $line = fgets($fp);
        $fields = explode(";", $line);
        if($user == $fields[0]) {
            return $fields;
        }
    }
    return false;e
}
?>
```

So wie er ist, enthält der Code zwei Fehler: Einer ist ein reiner Logikfehler, und der andere liegt darin, dass versäumt wurde, mögliche externe Fehler zu bedenken. Wenn Sie dieses Beispiel ausführen, erhalten Sie ein Array mit folgenden Elementen:

```
<?php
    print_r(get_passwd_info('www'));
?>
Array
        (
            [0] => www:*:70:70:World Wide Web Server:/Library/WebServer:/noshell
        )
```

Der erste Fehler ist, dass das Feldtrennzeichen »`:`« ist, und nicht »`;`«. Also muss diese Zeile:

```
$fields = explode(";", $line);
```

folgendermaßen geändert werden:

```
$fields = explode(":", $line);
```

Der zweite Fehler ist etwas komplizierter. Wenn es nicht gelingt die `passwd`-Datei zu öffnen, wird ein `E_WARNING`-Fehler erzeugt, aber der Programmfluss fortgesetzt. Ist ein User nicht in der `passwd`-Datei enthalten, gibt die Funktion `false` zurück. Auch wenn `fopen()` fehlschlägt, gibt die Funktion `false` zurück, was sehr irritierend ist.

Dieses schlichte Beispiel demonstriert eines der Kernprobleme der Fehlerbehandlung in prozeduralen Sprachen (oder auf jeden Fall in Sprachen ohne Ausnahmen): Wie propagiert man einen Fehler der aufrufenden Funktion, die vorbereitet ist, damit umzugehen?

Wenn Sie die Daten lokal bearbeiten, können Sie dort entscheiden, wie ein Fehler behandelt werden soll. Zum Beispiel ließe sich die Funktion `get_passwd_info()` so ändern, dass sie eine Fehlermeldung zurückgibt.

```
<?php
function get_passwd_info($user) {
    $fp = fopen("/etc/passwd", "r");
    if(!is_resource($fp)) {
        return "Error opening file";
    }
    while(!feof($fp)) {
        $line = fgets($fp);
        $fields = explode(":", $line);
        if($user == $fields[0]) {
            return $fields;
        }
    }
    return false;
}
?>
```

Alternativ können Sie einen bestimmten Wert zurückgeben, der normalerweise nicht gültig ist bzw. vorkommen kann:

```
<?php
function get_passwd_info($user) {
    $fp = fopen("/etc/passwd", "r");
    if(!is_resource($fp)) {
        return "Error opening file";
    }
    while(!feof($fp)) {
        $line = fgets($fp);
        $fields = explode(":", $line);
        if($user == $fields[0]) {
            return $fields;
        }
    }
    return false;
}
?>
```

Sie können diese Art der Logik verwenden, um Fehler nach oben zu den aufrufenden Funktionen durchzureichen:

open source library

```
<?php
function is_shelled_user($user) {
    $passwd_info = get_passwd_info($user);
    if(is_array($passwd_info) && $passwd_info[7] != '/bin/false') {
        return 1;
    }
    else if($passwd_info === -1) {
        return -1;
    }
    else {
        return 0;
    }
}
?>
```

Bei diesem Verfahren müssen Sie alle möglichen Fehler entdecken:

```
<?php
$v = is_shelled_user('www');
if($v === 1) {
    print "Your Web server user probably shouldn't be shelled.\n";
}
else if($v === 0) {
    print "Great!\n";
}
else {
    print "An error occurred checking the user\n";
}
?>
```

Wenn Ihnen dies hässlich und verwirrend erscheint, dann deswegen, weil es das ist. Der Aufwand, per Hand Fehler durch mehrere aufrufende Funktionen durchzureichen, ist einer der Hauptgründe für die Implementierung von Ausnahmen in Programmiersprachen. Und jetzt unterstützt auch PHP in der Version 5 Ausnahmen. Irgendwie werden Sie dieses spezielle Beispiel schon zum Laufen bringen, aber was ist, wenn die betreffende Funktion jede Zahl als gültigen Rückgabewert haben darf? Wie können Sie den Fehler klar und deutlich nach oben durchreichen? Das größte Problem in dieser verworrenen Fehlerbehandlung liegt darin, dass die Fehlerbehandlung nicht in der Funktion enthalten ist, in der der Fehler auftritt, sondern dass sie in allen aufrufenden Funktionen der Hierarchie verstanden und behandelt werden muss.

3.3 Ausnahmen

Alle bisher abgehandelten Verfahren waren bereits vor PHP 5 verfügbar und Sie werden bemerken, dass dies ein kritischer Punkt ist, speziell wenn Sie größere Applikationen schreiben. Das grundlegende Problem besteht darin, Fehler an die aufrufende

Funktion zurückzugeben. Beachten Sie den Fehlertest der Funktion `get_passwd_info()`. Als Sie dieses Beispiel geschrieben haben, standen Ihnen zur Behandlung eines Verbindungsfehlers zwei Möglichkeiten zur Verfügung:

- Den Fehler lokal behandeln und ungültige Daten (z.B. `false`) zurückgeben.
- Den Fehler festhalten und der aufrufenden Funktion über den Rückgabewert bekannt geben, anstatt das Ergebnis zurückzugeben.

In der Funktion `get_passwd_info()` haben Sie sich nicht für die erste Möglichkeit entschieden, da sie voraussetzt, dass die Bibliothek weiß, wie die Applikation den Fehler behandelt haben möchte. Wenn Sie z.B. eine Datenbanktest-Suite schreiben, soll der Fehler vielleicht an die oberste aufrufende Funktion zurückgegeben werden; in einer Web-Anwendung möchten Sie den User auf eine Fehlerseite umleiten.

In dem vorausgegangenen Beispiel wird die zweite Methode angewendet, aber sie ist nicht viel besser als die erste. Das Problem besteht darin, dass Vorausschau und Planung erforderlich sind, damit der Fehler immer richtig durch die Applikation gegeben wird. Wenn beispielsweise das Ergebnis einer Datenbank ein String ist, wie differenzieren Sie dann zwischen diesem und einem Fehler-String? Außerdem muss die Weitergabe per Hand vorgenommen werden. Auf jeder Stufe muss der Fehler manuell zur aufrufenden Funktion gegeben, als Fehler erkannt und entweder weitergereicht oder behandelt werden. Im letzten Abschnitt haben Sie gesehen, wie schwierig diese Schritte sind.

Ausnahmen sind dafür gedacht, solche Situationen zu behandeln. Eine Ausnahme ist eine Kontrollflussstruktur, über die es möglich ist, den aktuellen Pfad der Ausführung des Skriptes zu stoppen und an einem bestimmten Punkt wieder aufzunehmen. Der auftretende Fehler wird durch ein Objekt repräsentiert, das als Ausnahme gesetzt wird.

Ausnahmen sind Objekte. Um bei dem Umgang mit einfachen Ausnahmen zu helfen, besitzt PHP die eingebaute Klasse `Exception`, die speziell für Ausnahmen entworfen wurde. Obwohl es nicht unbedingt notwendig ist, dass Ausnahmen Instanzen der Klasse `Exception` sind, bietet es sich jedoch an, jede Klasse, der man Ausnahmen übergeben möchte, von der Klasse `Exception` abzuleiten – was wir gleich diskutieren werden.

Um eine neue Ausnahme zu erstellen, initialisieren Sie eine Instanz der Klasse `Exception` und rufen diese mit dem Befehl `throw` auf.

Wird eine Ausnahme ausgelöst, wird das `Exception`-Objekt gespeichert und die Ausführung im derzeitigen Codeblock sofort angehalten. Sofern es einen Codeblock zur Ausnahmebehandlung im aktuellen Bereich gibt, wird dort hingesprungen und die Ausnahmebehandlung durchgeführt. Ist das nicht der Fall, wird im Bereich der aufrufenden Funktion nach einem Codeblock zur Ausnahmebehandlung gesucht. Dies wiederholt sich solange bis ein Block gefunden wird oder die oberste Ebene erreicht ist. Das Ausführen dieses Codes:

open source library

```
<?php
throw new Exception;
?>
```

führt zu folgendem Ergebnis:

```
> php uncaught-exception.php

Fatal error: Uncaught exception 'exception'! in Unknown on line 0
```

Eine nicht abgefangene Ausnahme ist ein schwerer Fehler, sodass Ausnahmen ihren eigenen Pflegeaufwand mitbringen. Wenn Ausnahmen als Warnung für mögliche Fehler verwendet werden, muss jeder aufrufenden Funktion bekannt sein, dass eventuell eine Ausnahme ausgelöst wird und bereit sein, diese zu verarbeiten.

Die Fehlerbehandlung besteht aus einem Anweisungsblock, der zunächst ausprobiert und einem zweiten, der ausgeführt wird, wenn ein Fehler aufgetreten ist. Nachfolgend zeigt ein einfaches Beispiel, wie eine Ausnahme ausgelöst und abgefangen wird:

```
try {
    throw new Exception;
    print "This code is unreached\n";
}
catch (Exception $e) {
    print "Exception caught\n";
}
```

In diesem Fall wird eine Ausnahme ausgelöst, aber sie steckt im `try`-Block. Die Ausführung wird angehalten und es wird zum `catch`-Block gesprungen. `Catch` fängt eine Ausnahme-Klasse (die Klasse, die ausgelöst wurde), sodass der Block ausgeführt wird. Im `catch`-Block werden normalerweise »Aufräumarbeiten« durchgeführt, die eventuell durch den aufgetretenen Fehler notwendig wurden.

Ich habe bereits erwähnt, dass es nicht notwendig ist, eine Instanz der Klasse `Exception` zu werfen. Hier folgt ein Beispiel:

```
<?php

class AltException {}

try {
        throw new AltException;
}
catch (Exception $e) {
        print "Caught exception\n";
}
?>
```

Die Ausführung dieses Beispiels ergibt das folgende Resultat:

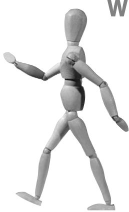
```
> php failed_catch.php
Fatal error: Uncaught exception 'altexception'! in Unknown on line 0
```

In dieses Beispiel gelingt es nicht, die Ausnahme abzufangen, da ein Objekt der Klasse `AltException` ausgelöst, aber nur versucht wurde, ein Objekt der Klasse `Exception` aufzufangen.

Als Nächstes sehen Sie in einem weniger trivialen Beispiel, wie Sie eine einfache Ausnahme für die Fehlerbehandlung einsetzen können. Die Fakultätsfunktion ist nur für natürliche Zahlen gültig (Zahlenwerte > 0). Sie können diesen Test in die Applikation einbinden, indem Sie eine Ausnahme auslösen, falls falsche Daten übergeben werden:

```
<?php
// factorial.inc
// Eine einfache Fakultätsfunktion
function factorial($n) {
    if(!preg_match('/^\d+$/',$n) || $n < 0 ) {
        throw new Exception;
    } else if ($n == 0 || $n == 1) {
        return $n;
    }
    else {
        return $n * factorial($n - 1);
    }
}
?>
```

Ein solider Test der übergebenen Funktionsparameter ist der Schlüssel zu defensiver Programmierung.



Warum ein regulärer Ausdruck?

Es mag Ihnen zunächst merkwürdig vorkommen, einen regulären Ausdruck anstelle die Funktion `is_int` zu verwenden, um herauszufinden, ob `$n` ein Integer ist. Die Funktion `is_int` macht aber nicht, was Sie wünschen. Sie bewertet nur, ob `$n` als String oder Integer eingegeben wurde und nicht, ob der Wert von `$n` ein Integer ist. Dies ist ein kleiner aber feiner Unterschied, der Sie einholen wird, wenn Sie `is_int` verwenden, um (unter anderem) Formulardaten zu validieren. Wir werden die dynamische Typenvergabe in PHP in Kapitel 20, PHP und die Zend Engine, behandeln.

Wenn Sie die Funktion `factorial()` aufrufen, müssen Sie sicherstellen, dass dies in einem `try`-Block geschieht. Andernfalls riskieren Sie, dass die Applikation beendet wird, falls ungültige Daten übergeben werden:

open source library

```
<html>
<form method="POST">
Berechne die Fakultät von
<input type="text" name="input" value="<?= $_POST['input'] ?>"><br>
<?php
include "factorial.inc";
if($_POST['input']) {
    try {
        $input = $_POST['input'];
        $output = factorial($input);
        echo "$_POST[input]! = $output";
    }
    catch (Exception $e) {
        echo "Only natural numbers can have their factorial
              computed.";
    }
}
?>
<br>
<input type=submit name=posted value="Submit">
</form>
```

3.3.1 Hierarchien von Ausnahmen verwenden

Sie können mit `try` mehrere `catch`-Blöcke verwenden, wenn Sie unterschiedliche Fehler unterschiedlich behandeln möchten. Z.B. können Sie die Funktion `factorial` so ergänzen, dass der Fall abgefangen wird, dass `$n` zu groß für die mathematischen Funktionen von PHP wird:

```
class OverflowException {}
class NaNException {}
function factorial($n)
{
    if(!preg_match('/^\d+$/', $n) || $n < 0 ) {
        throw new NaNException;
    }
    else if ($n == 0 || $n == 1) {
        return $n;
    }
    else if ($n > 170 ) {
        throw new OverflowException;
    }
    else {
        return $n * factorial($n - 1);
    }
}
```

open source library

Jetzt können Sie jeden Fehler differenziert behandeln:

```
<?php
if($_POST['input']) {
    try {
        $input = $_POST['input'];
        $output = factorial($input);
        print "$_POST[input]! = $output";
    }
    catch (OverflowException $e) {
        print "The requested value is too large.";
    }
    catch (NaNException $e) {
        print "Only natural numbers can have their factorial
              computed.";
    }
}
?>
```

In dieser Variante müssen Sie nun jeden Fall separat bedenken. Dies ist sowohl mühselig zu schreiben als auch potenziell gefährlich. Da mit wachsender Bibliothek auch die Anzahl der möglichen Ausnahmen wächst und somit die Wahrscheinlichkeit, versehentlich einen Fall zu vergessen. Um damit umgehen zu können, können Sie die Ausnahmen in Familien gliedern und einen Vererbungsbaum kreieren:

```
class MathException extends Exception {}
class NaNException extends MathException {}
class OverflowException extends MathException {}
```

Jetzt können Sie den `catch`-Block folgendermaßen umstrukturieren:

```
<?php
if($_POST['input']) {
    try {
        $input = $_POST['input'];
        $output = factorial($input);
        print "$_POST[input]! = $output";
    }
    catch (OverflowException $e) {
        print "The requested value is too large.";
    }
    catch (MathException $e) {
        print "A generic math error occurred";
    }
    catch (Exception $e) {
        print "An unknown error occurred";
    }
}
?>
```

open source library

In diesem Skript wird bei Auslösung eines `OverflowException`-Fehlers dieser durch den ersten `catch`-Block aufgefangen. Wenn irgendein anderer Abkömmling von `Math Exception` (z.B. `NaNException`) ausgelöst wird, wird er durch den zweiten `catch`-Block aufgefangen. Schließlich wird jeder Abkömmling von `Exception`, der nicht durch die vorhergehenden Fälle aufgefangen wurde, aufgefangen. Deswegen ist es günstig, alle Ausnahmen durch Vererbung von `Exception` abzuleiten. Es ist möglich, einen generellen `catch`-Block zu schreiben, der alle Ausnahmen behandelt ohne sie individuell aufzulisten. Eine Ausnahmebehandlung für alle Ausnahmen zu haben ist wichtig, da darüber Fehler abgefangen werden, die Sie nicht erwartet haben.

3.3.2 Beispiel für eine typisierte Ausnahme

Bisher waren in diesem Kapitel alle Ausnahmen ohne Attribute (zumindest soweit wir es wissen). Müssen Sie nur den Typ der Ausnahme identifizieren, die ausgelöst wird, und haben Sie die Hierarchie mit Bedacht gewählt, werden diese Ausnahmen den meisten Ihrer Anforderungen gerecht werden. Wenn die einzige interessante Information, die Sie über Ausnahmen weiterreichen möchten, immer nur Strings wären, hätte es ausgereicht, Strings anstelle von ganzen Objekten zu implementieren. Aber Sie sollten in der Lage sein, der aufrufenden Funktion, die die Ausnahme abfängt, beliebige, eventuell hilfreiche Informationen zu übergeben. Die Basisklasse `Exception` ist tatsächlich umfangreicher als bisher angedeutet. Es handelt sich um eine eingebaute Klassen, d.h. sie ist in C implementiert statt in PHP. In PHP würde sie in groben Zügen so aussehen:

```
class Exception {
    public function __construct($message=false, $code=false) {
        $this->file = __FILE__;
        $this->line = __LINE__;
        $this->message = $message; // Die Fehlermeldung als String
        $this->code = $code;  // Hier kann eine Fehlernummer
                              // eingesetzt werden
    }
    public function getFile() {
        return $this->file;
    }
    public function getLine() {
        return $this->line;
    }
    public function getMessage() {
        return $this->message;
    }
    public function getCode() {
       return $this->code;
    }
}
```

__FILE__ und __LINE__ sind für die letzte aufrufende Funktion oft nutzlose Informationen. Stellen Sie sich vor, Sie lösen eine Ausnahme aus, wenn Sie ein Problem mit einer Abfrage in der Klasse DB_Mysql haben:

```
class DB_Mysql {
    // ...
    public function execute($query) {
      if(!$this->dbh) {
        $this->connect();
      }
      $ret = mysql_query($query, $this->dbh);
      if(!is_resource($ret)) {
        throw new Exception;
      }
      return new MysqlStatement($ret);
    }
}
```

Wenn Sie diese Ausnahme nun durch Ausführen einer syntaktisch ungültigen Abfrage auslösen:

```
<?php
        require_once "DB.inc";
        try {
        $dbh = new DB_Mysql_Test;
        // Ausführen einiger Abfragen mit dieser Datenbankverbindung
        $rows = $dbh->execute("SELECT * FROM")->fetchall_assoc();
        }
        catch (Exception $e) {
                print_r($e);
        }
?>
```

erhalten Sie Folgendes:

```
exception Object
(
    [file] => /Users/george/Advanced PHP/examples/chapter-3/DB.inc
    [line] => 42
)
```

Zeile 42 der DB.inc ist die Methode execute() selber. Wenn Sie eine Anzahl von Abfragen innerhalb des try-Blocks ausführen, können Sie nicht erkennen, welche dieser Abfragen diesen Fehler verursacht. Es wird noch schlimmer. Wenn Sie Ihre eigene Klasse zur Ausnahmebehandlung verwenden und manuell $file und $line setzen (oder über parent::__construct den Konstruktor aufrufen), werden Sie mit den Angaben der ersten aufrufenden Funktion in __FILE__ und __LINE__ enden. Aber eigentlich soll es möglich sein, den Fehler ab dem Moment seines Auftretens vollständig zurückzuverfolgen.

open source library

Jetzt können Sie anfangen, die DB-Bibliothek mit Ausnahmen zu füllen. Zusätzlich zum Sammeln der Rückverfolgungsdaten können Sie die Attribute `message` und `code` mit den MySQL-Fehlerinformationen setzen:

```
class MysqlException extends Exception {
  public $backtrace;
  public function __construct($message=false, $code=false) {
    if(!$message) {
      $this->message = mysql_error();
    }
    if(!$code) {
      $this->code = mysql_errno();
    }
    $this->backtrace = debug_backtrace();
  }
}
```

Wenn Sie jetzt die Bibliothek für die Verwendung dieser Ausnahmetypen anpassen:

```
class DB_Mysql {
  public function execute($query) {
    if(!$this->dbh) {
      $this->connect();
    }
    $ret = mysql_query($query, $this->dbh);
    if(!is_resource($ret)) {
      throw new MysqlException;
    }
    return new MysqlStatement($ret);
  }
}
```

und den Test wiederholen:

```
<?php
        require_once "DB.inc";
        try {
        $dbh = new DB_Mysql_Test;
        // Ausführen einiger Abfragen mit dieser Datenbankverbindung
        $rows = $dbh->execute("SELECT * FROM")->fetchall_assoc();
        }
        catch (Exception $e) {
                print_r($e);
        }
?>
```

erhalten Sie als Ergebnis:

```
mysqlexception Object
(
  [backtrace] => Array
```

open source library


```
    (
      [0] => Array
        (
          [file] => /Users/george/Advanced PHP/examples/chapter-3/DB.inc
          [line] => 45
          [function] => __construct
          [class] => mysqlexception
          [type] => ->
          [args] => Array
            (
            )
         )
      [1] => Array
        (
          [file] => /Users/george/Advanced PHP/examples/chapter-3/test.php
          [line] => 5
          [function] => execute
          [class] => mysql_test
          [type] => ->
          [args] => Array
            (
              [0] => SELECT * FROM
            )
        )
    )

  [message] => You have an error in your SQL syntax near '' at line 1
  [code] => 1064
)
```

Verglichen zur vorherigen Ausnahme enthält diese Ausnahme ein Kaleidoskop an Informationen:

- Wo ist der Fehler aufgetreten?
- Wie ist die Applikation zu diesem Punkt gelangt?
- Die MySQL-Fehlerinformationen

Sie können jetzt die komplette Bibliothek entsprechend anpassen:

```
class MysqlException extends Exception {
  public $backtrace;
  public function __construct($message=false, $code=false) {
    if(!$message) {
      $this->message = mysql_error();
    }
    if(!$code) {
     $this->code = mysql_errno();
    }
    $this->backtrace = debug_backtrace();
  }
```

```
}
class DB_Mysql {
  protected $user;
  protected $pass;
  protected $dbhost;
  protected $dbname;
  protected $dbh;

  public function __construct($user, $pass, $dbhost, $dbname) {
    $this->user = $user;
    $this->pass = $pass;
    $this->dbhost = $dbhost;
    $this->dbname = $dbname;
  }
  protected function connect() {
    $this->dbh = mysql_pconnect($this->dbhost, $this->user,
                               $this->pass);
    if(!is_resource($this->dbh)) {
      throw new MysqlException;
    }
    if(!mysql_select_db($this->dbname, $this->dbh)) {
      throw new MysqlException;
    }
  }
  public function execute($query) {
    if(!$this->dbh) {
      $this->connect();
    }
    $ret = mysql_query($query, $this->dbh);
    if(!$ret) {
      throw new MysqlException;
    }
    else if(!is_resource($ret)) {
      return TRUE;
    } else {
      return new DB_MysqlStatement($ret);
    }
  }
  public function prepare($query) {
    if(!$this->dbh) {
      $this->connect();
    }
    return new DB_MysqlStatement($this->dbh, $query);
  }
}
class DB_MysqlStatement {
  protected $result;
  protected $binds;
  public $query;
```

open source library

```
  protected $dbh;
  public function __construct($dbh, $query) {
    $this->query = $query;
    $this->dbh = $dbh;
    if(!is_resource($dbh)) {
      throw new MysqlException("Not a valid database connection");
    }
  }
  public function bind_param($ph, $pv) {
    $this->binds[$ph] = $pv;
  }
  public function execute() {
    $binds = func_get_args();
    foreach($binds as $index => $name) {
      $this->binds[$index + 1] = $name;
    }
    $cnt = count($binds);
    $query = $this->query;
    foreach ($this->binds as $ph => $pv) {
      $query = str_replace(":$ph", "'".mysql_escape_string($pv)."'",
                          $query);
    }
    $this->result = mysql_query($query, $this->dbh);
    if(!$this->result) {
      throw new MysqlException;
    }
  }
  public function fetch_row() {
    if(!$this->result) {
      throw new MysqlException("Query not executed");
    }
    return mysql_fetch_row($this->result);
  }
  public function fetch_assoc() {
    return mysql_fetch_assoc($this->result);
  }
  public function fetchall_assoc() {
    $retval = array();
    while($row = $this->fetch_assoc()) {
      $retval[] = $row;
    }
    return $retval;
  }
}

?>
```

open source library

3.3.3 Mehrstufige Ausnahmen

Mitunter möchten Sie einen Fehler behandeln, aber ihn dennoch zur Fortsetzung der Fehlerbehandlung weiterreichen. Das erreichen Sie durch Auslösung einer neuen Ausnahme im `catch`-Block:

```
<?php
try {
        throw new Exception;
}
catch (Exception $e) {
        print "Exception caught, and rethrown\n";
        throw new Exception;
}
?>
```

Der `catch`-Block fängt die Ausnahme ab und druckt seine Meldung. Anschließend löst er eine neue Ausnahme aus. Im vorgehenden Beispiel gibt es keinen weiteren `catch`-Block, der diese neue Ausnahme abfängt, sodass sie nicht verarbeitet wird. Beachten Sie, was passiert, wenn Sie den Code ausführen:

```
> php re-throw.php
Exception caught, and rethrown

Fatal error: Uncaught exception 'exception'! in Unknown on line 0
```

De facto ist es nicht notwendig, eine neue Ausnahme zu erstellen. Wenn Sie möchten, können Sie das aktuelle `Exception`-Objekt mit identischem Ergebnis neu auslösen:

```
<?php
try {
        throw new Exception;
}
catch (Exception $e) {
        print "Exception caught, and rethrown\n";
        throw $e;
}
?>
```

Eine Ausnahme neu auslösen zu können, ist wichtig, weil Sie unter Umständen nicht mit Sicherheit wissen, ob Sie die Ausnahme behandeln wollen, wenn Sie sie auffangen. Wenn Sie z.B. Verweise auf Ihrer Webseite beobachten möchten, benötigen Sie eine solche Tabelle:

```
CREATE TABLE track_referrers (
    url varchar2(128) not null primary key,
    counter int
);
```

open source library

Das erste Mal beim Auftreten einer URL als Referrer müssen Sie folgende Abfrage ausführen:

```
INSERT INTO track_referrers VALUES('http://some.url/', 1)
```

Beim wiederholten Auftreten der URL benötigen Sie folgende Abfrage:

```
UPDATE track_referrers SET counter=counter+1 where url = 'http://some.url/'
```

Sie können nun zunächst abfragen, ob die betreffende URL bereits in der Tabelle existiert, um die richtige Abfrage auszuwählen. Dieses Verfahren birgt die seltene Gefahr, dass ein `Insert` fehlschlägt, wenn zweimal die gleiche URL von zwei verschiedenen Prozessen gleichzeitig verarbeitet wird.

Eine sauberere Lösung besteht darin, die `Insert`-Abfrage blind auszuführen und die `Update`-Abfrage auszuführen, wenn die `Insert`-Abfrage fehlschlägt und eine Verletzung des eindeutigen Schlüssels (primary key) zurückgibt. So können Sie alle `Mysql Exeption`-Fehler abfangen und die Abfrage ausführen, sofern notwendig:

```
<?php
include "DB.inc";

function track_referrer($url) {
     $insertq = "INSERT INTO referrers (url, count) VALUES(:1, :2)";
     $updateq = "UPDATE referrers SET count=count+1 WHERE url = :1";
     $dbh = new DB_Mysql_Test;
     try {
             $sth = $dbh->prepare($insertq);
             $sth->execute($url, 1);
     }
     catch (MysqlException $e) {
             if($e->getCode == 1062) {
                     $dbh->prepare($updateq)->execute($url);
             }
             else {
                     throw $e;
             }
     }
}
?>
```

Alternativ können Sie eine Lösung mit typisierten Ausnahmen verwenden, wobei die Methode `execute()` abhängig vom auftretenden Fehler verschiedene Ausnahmen auslöst:

```
class Mysql_Dup_Val_On_Index extends MysqlException {}
//...
class DB_Mysql {
  // ...
```

open source library

```
  public function execute($query) {
    if(!$this->dbh) {
      $this->connect();
    }
    $ret = mysql_query($query, $this->dbh);
    if(!$ret) {
      if(mysql_errno() == 1062) {
        throw new Mysql_Dup_Val_On_Index;
      else {
        throw new MysqlException;
      }
    }
    else if(!is_resource($ret)) {
       return TRUE;
    } else {
      return new MysqlStatement($ret);
    }
  }
}
```

Dann kann der Test wie folgt durchgeführt werden:

```
function track_referrer($url) {
  $insertq = "INSERT INTO referrers (url, count) VALUES('$url', 1)";
  $updateq = "UPDATE referrers SET count=count+1
              WHERE url = '$url'";
  $dbh = new DB_Mysql_Test;
  try {
    $sth = $dbh->execute($insertq);
  }
  catch (Mysql_Dup_Val_On_Index $e) {
    $dbh->execute($updateq);
  }
}
```

Beide Wege sind möglich, es ist nur eine Frage des Geschmacks und des Programmierstils. Wenn Sie typisierte Ausnahmen verwenden, können Sie etwas Flexibilität gewinnen, indem Sie ein Fabrikmuster einsetzen, um die Fehler zu erzeugen:

```
class MysqlException {
  // ...
  static function createError($message=false, $code=false) {
     if(!$code) {
       $code = mysql_errno();
     }
     if(!$message) {
       $message = mysql_error();
     }
     switch($code) {
```

open source library


```
      case 1062:
        return new Mysql_Dup_Val_On_Index($message, $code);
        break;
      default:
        return new MysqlException($message, $code);
        break;
     }
   }
}
```

Durch die verbesserte Lesbarkeit ergibt sich ein zusätzlicher Vorteil. Anstelle von kryptischen Konstanten werden suggestive Klassennamen verwendet. Der Wert besserer Lesbarkeit sollte nicht unterschätzt werden.

Anstatt spezifische Fehler in Ihrem Code auszulösen, verwenden Sie einfach den Befehl:

```
throw MysqlException::createError();
```

3.3.4 Fehler im Konstruktor behandeln

Die Behandlung von Fehlern im Konstruktor eines Objekts ist eine schwierige Angelegenheit. Ein Konstruktor einer Klasse muss in PHP eine Instanz der Klasse zurückgeben, sodass die Möglichkeiten begrenzt sind:

- Sie können ein initialisiertes Attribut verwenden, um anzuzeigen, dass das Objekt korrekt initialisiert wurde.
- Sie können keine Initialisierungen im Konstruktor durchführen.
- Sie können eine Ausnahme im Konstruktor auslösen.

Die erste Möglichkeit ist nicht sehr elegant, und wir werden sie daher nicht ernsthaft in Betracht ziehen. Die zweite Möglichkeit ist ein weit verbreiteter Weg zur Fehlerbehandlung im Konstruktor. Tatsächlich ist sie in PHP 4 die zu bevorzugende Variante.

Um diesen Weg zu implementieren, schreiben Sie in etwa folgenden Code:

```
class ResourceClass {
  protected $resource;
  public function __construct() {
    // Benutzername, Kennwort etc. setzen
  }
  public function init() {
    if(($this->resource = resource_connect()) == false) {
      return false;
    }
    return true;
  }
}
```

open source library

Wenn ein neues Objekt der Klasse ResourceClass erstellt wird, werden keine Aktionen durchgeführt, die fehlschlagen könnten. Zur Durchführung von Initialisierungen, die potenziell Fehler erzeugen können, wird die Methode init() aufgerufen. Diese kann Fehler generieren, allerdings ohne größere Auswirkung.

Die dritte Möglichkeit ist normalerweise zu bevorzugen, insbesondere da dieser Weg das Standardverfahren bei traditionellen OO-Sprachen wie C++ ist. In C++ sind die Aufräumarbeiten im catch-Block um den Konstruktor wichtiger als in PHP, da eventuell Speichermanagement bedacht werden muss. Zum Glück wird Ihnen das Speichermanagement in PHP abgenommen, zu sehen in folgendem Beispiel:

```
class Stillborn {
  public function __construct() {
    throw new Exception;
  }
  public function __destruct() {
    print "destructing\n";
  }
}
try {
  $sb = new Stillborn;
}
catch(Stillborn $e) {}
```

Das Ausführen dieses Codes erzeugt keine Ausgabe:

```
>php stillborn.php
>
```

Die Klasse Stillborn demonstriert, dass der Destruktor nicht aufgerufen wird, wenn eine Ausnahme im Konstruktor ausgelöst wird. Dies liegt daran, dass das Objekt tatsächlich nicht existiert, bis der Konstruktor komplett abgearbeitet ist.

3.3.5 Einrichten eines Top-Level-Exception-Handlers

Ein interessantes Feature in PHP ist die Möglichkeit, einen Standard-Exception-Handler einzurichten, der aufgerufen wird, wenn eine Ausnahme den obersten Level erreicht hat und noch nicht abgefangen worden ist. Dieser Handler unterscheidet sich von einen catch-Block insofern, als dass es sich um eine einzige Funktion handelt, die unabhängig vom Typ (inklusive Ausnahmen, die nicht von Exception vererbt wurden) mit jeder noch nicht abgefangenen Ausnahme umgehen wird.

Der Standard-Exception-Handler ist besonders in solchen Web-Applikationen sinnvoll, in denen Sie verhindern möchten, dass ein User im Fall einer nicht abgefangenen Ausnahme eine halbfertige oder fehlerhafte Seite erhält. Wenn Sie Output Buffering (Ausgabezwischenspeicher) von PHP dazu verwenden, den Inhalt einer Seite erst dann zu senden, wenn sie vollständig generiert wurde, können Sie elegant jeden Fehler umgehen und den User zur gewünschten Seite umleiten.

Um den Standard-Exception-Handler zu setzen, definieren Sie eine Funktion mit einem einzigen Parameter:

```
function default_exception_handler($exception) {}
```

Sie setzen diese Funktion folgendermaßen:

```
$old_handler = set_exception_handler('default_exception_handler');
```

Der vorher definierte Standard-Exception-Handler wird zurückgegeben (wenn er existiert).

Benutzerdefinierte Exception-Handler werden in einem Stack (Stapel) gehalten, sodass Sie den alten Handler entweder durch Schieben einer Kopie in den Stack wiederherstellen

```
set_exception_handler($old_handler);
```

oder indem Sie im Stack einen Eintrag weiterrutschen:

```
restore_exception_handler();
```

Ein Beispiel für die gewonnen Flexibilität hängt damit zusammen, dass Sie bei Fehlern, die während des Generierens der Seite auftreten, Umleitungen festlegen können. Anstatt jede in Frage kommende Möglichkeit in einen einzelnen try-Block einzufügen, können Sie einen Standard-Exception-Handler setzen, der die Umleitungen vornimmt. Da ein Fehler auftreten kann, nachdem die Seite bereits partiell erstellt ist, müssen Sie dafür sorgen, dass der Output Buffer eingeschaltet ist. Entweder dadurch, dass Sie am Anfang jeder Datei:

```
ob_start();
```

schreiben oder indem Sie in der php.ini-Datei folgenden Eintrag setzen:

```
output_buffering = On
```

Der Vorteil der ersten Variante besteht darin, dass Sie das Verhalten einfach von Seite zu Seite ändern können. Außerdem ist Ihr Code dadurch einfacher portierbar (weil das Verhalten durch das Skript bestimmt wird, und es keine Nicht-Standard-Einstellung in der php.ini erfordert). Die zweite Methode besitzt den Vorteil, dass der Output Buffer in jedem Skript mit nur einer Einstellung aktiviert wird und Sie nicht in jedes Skript den Code für den Output Buffer hinzufügen müssen. Wenn ich ein Programm schreibe, von dem ich weiß, dass es nur auf meinen Computern laufen wird, ändere ich lieber .ini-Einstellungen. Sas macht das Leben einfacher. Wenn ich eine Software schreibe, die auf den Servern der Kunden läuft, steht eine portierbare Lösung im Vordergrund. Normalerweise ist es am Beginn eines Projektes klar, welche Richtung das Projekt nehmen wird.

open source library

Das nächste Beispiel eines Standard-`Exception`-Handlers generiert automatisch eine Fehlerseite für jede nicht abgefangene Ausnahme:

```
<?php
function redirect_on_error($e) {
  ob_end_clean();
  include("error.html");
}
set_exception_handler("redirect_on_error");
ob_start();
// ... der Code der Seite kommt hier hin
?>
```

Dieser Handler verlässt sich darauf, dass Output Buffering aktiviert ist, sodass er – wenn eine nicht abgefangene Ausnahme bis auf die oberste Ebene durchgereicht wurde – den bis dahin generierten Inhalt verwerfen und stattdessen eine HTML-Fehlerseite zurückgeben kann.

Sie können diesen Handler weiterhin verbessern, indem Sie ihn mit der Fähigkeit ausstatten, bestimmte Fehler unterschiedlich zu behandeln. Wenn Sie z.B. eine `AuthException`-Ausnahme durchreichen, können Sie zur Login-Seite umleiten anstatt eine Fehlerseite anzuzeigen:

```
<?php
function redirect_on_error($e) {
  ob_end_clean();
  if(is_a($e, "AuthException")) {
    header("Location: /login.php");
  }
  else {
    include("error.html");
  }
}
set_exception_handler("redirect_on_error");
ob_start();
// ... der Code der Seite kommt hier hin
? >
```

3.3.6 Daten überprüfen

Einer der Hauptfehlerquellen bei der Web-Programmierung steckt in der ungenügenden Überprüfung der vom Client übergebenen Daten. Im Rahmen der Datenüberprüfung muss sichergestellt werden, dass die Daten vom Client tatsächlich in der Form vorliegen, wie Sie sie erwartet hatten. Nicht validierte Daten treten auf als:

- Müll-Daten
- Bösartig geänderte Daten

Müll-Daten sind Informationen, die einfach nicht der geforderten Spezifikation entsprechen. Wenn die Benutzer zum Beispiel in einem Anmeldeformular geografische Informationen eingeben und dabei das Bundesland als freien Text eintippen können, dann haben Sie Ergebnisse wie die folgenden selbst provoziert:

- Bandenburg (Tippfehler)
- Lalalala (absichtlich falsch)

Eine übliche Lösung besteht darin, Auswahllisten mit den Bundesländern bereit zu stellen. Dies löst allerdings nur die Hälfte des Problems. Sie haben zwar verhindert, dass User unkorrekte Namen eingeben, aber es bietet keinen Schutz vor bösartig geänderten POST-Daten, mit denen nicht existierende Optionen übergeben werden können.

Um sich davor zu schützen, sollten Sie die eingehenden Daten immer im Skript validieren (bevor Sie irgendwas mit diesen Daten anstellen):

```
<?php
$STATES = array('sh' => 'Schleswig Holstein',
                 /* ... */,
                '??' => 'Brandenburg');
function is_valid_state($state) {
  global $STATES;
  return array_key_exists($STATES, $state);
}
?>
```

Ich füge gern eine Methode zur Validierung der Klasse hinzu, um den Validierungscode zu kapseln, und damit ich nicht vergesse, ein Attribut zu überprüfen. Hier ein Beispiel dafür:

```
<?php

class User {
  public id;
  public name;
  public city;
  public state;
  public zipcode;
  public function __construct($attr = false) {
    if($attr) {
      $this->name = $attr['name'];
      $this->email = $attr['email'];
      $this->city = $attr['city'];
      $this->state = $attr['state'];
      $this->zipcode = $attr['zipcode'];
    }
  }
  public function validate() {
```

open source library

```
    if(strlen($this->name) > 100) {
      throw new DataException;
    }
    if(strlen($this->city) > 100) {
      throw new DataException;
    }
    if(!is_valid_state($this->state)) {
      throw new DataException;
    }
    if(!is_valid_zipcode($this->zipcode)) {
      throw new DataException;
    }
  }
}

?>
```

Die Methode `validate()` überprüft alle Attribute des Objektes `User` inklusive der folgenden Punkte:

- Übereinstimmung mit der Länge des Datenbankfeldes.
- Behandlung von Beschränkung durch foreign keys (z.B., dass das Bundesland gültig ist).
- Behandlung von Einschränkungen durch die Form der Daten, z. B. PLZ oder Zip-Code).

Um die Methode `validate()` zu nutzen, können Sie einfach ein neues Objekt der Klasse `user` mit nicht vertrauenswürdigen Daten initialisieren:

```
$user = new User($_POST);
```

Und dann `validate()` aufrufen:

```
try {
    $user->validate();
}
catch (DataException $e) {
    /* Mache, was immer gemacht werden muss, wenn die Daten des
       Users ungültig sind */
}
```

Nochmal: Hier eine Ausnahme zu verwenden, anstatt `validate()` nur `true` oder `false` zurückgeben zu lassen, ist deswegen vorteilhaft, weil Sie eventuell an dieser Stelle keinen `try`-Block haben und Sie die Ausnahme lieber etwas weiter oben in der Aufrufhierarchie behandeln möchten.

Mit bösartig veränderten Daten lässt sich natürlich Schlimmeres bewirken als – wie im Beispiel – nicht existierende Namen von Bundesländern zu übergeben. Die bekanntesten Angriffe, die sich mangelhafte Datenvalidierung zu Nutze machen,

open source library

werden als »siteübergreifende Skriptingangriffe« (Cross-Site Scripting)[2] bezeichnet. »Siteübergreifende Skriptingangriffe« beinhalten bösartig verändertes HTML (normalerweise Tags für clientseitig ausgeführte Skriptsprachen wie z.B. JavaScript), das über Benutzerformulare übermittelt wird.

Der folgende Fall ist ein einfaches Beispiel. Wenn Sie Ihren Usern erlauben, einen Link auf Ihre Homepage zu setzen und diesen Link folgendermaßen anzeigen:

```
<a href="<?= $url ?>">Click on my home page</a>
```

Die URL kann aus beliebige Daten bestehen, die der User übermitteln kann. Er könnte also auch das Folgende übermitteln:

```
$url ='http://example.foo/" onClick=bad_javascript_func foo="';
```

Wenn die Seite erzeugt wird, resultiert dies in folgende Anzeige für den User:

```
<a href="'http://example.foo/" onClick=bad_javascript_func foo="">
  Click on my home page
</a>
```

Dies führt dazu, dass der Browser des Users die JavaScript-Funktion `bad_javascript_func` ausführt, wenn auf den Link geklickt wird. Nicht nur das: Da die Seite von Ihrer Webseite geliefert wurde, hat das JavaScript beim User vollen Zugriff auf die Cookies Ihrer Domain. Das ist natürlich alles andere als wünschenswert, da es bösartigen Usern erlaubt, Daten anderer User zu manipulieren, zu stehlen oder auszuspionieren. Überflüssig zu sagen: Angemessene Validierung jeglicher Art von Daten, die von Usern eingegeben werden können und auf der Webseite angezeigt werden, ist essentiell für die Sicherheit der Seite. Die Tags, die Sie filtern sollten, sind natürlich abhängig von Ihren Geschäftsregeln. Ich persönliche bevorzuge eine drakonische Herangehensweise an die Validierung und weise jeden Text ab, der nur ansatzweise aussieht wie JavaScript. Hier ist ein Beispiel:

```
<?php
$UNSAFE_HTML[] = "!javascript\s*:!is";
$UNSAFE_HTML[] = "!vbscri?pt\s*:!is";
$UNSAFE_HTML[] = "!<\s*embed.*swf!is";
$UNSAFE_HTML[] = "!<[^>]*[^a-z]onabort\s*=!is";
$UNSAFE_HTML[] = "!<[^>]*[^a-z]onblur\s*=!is";
$UNSAFE_HTML[] = "!<[^>]*[^a-z]onchange\s*=!is";
$UNSAFE_HTML[] = "!<[^>]*[^a-z]onfocus\s*=!is";
$UNSAFE_HTML[] = "!<[^>]*[^a-z]onmouseout\s*=!is";
$UNSAFE_HTML[] = "!<[^>]*[^a-z]onmouseover\s*=!is";
$UNSAFE_HTML[] = "!<[^>]*[^a-z]onload\s*=!is";
$UNSAFE_HTML[] = "!<[^>]*[^a-z]onreset\s*=!is";
$UNSAFE_HTML[] = "!<[^>]*[^a-z]onselect\s*=!is";
```

2 Anm. d. Fachl. Abgekürzt mit XSS.

```
$UNSAFE_HTML[] = "!<[^>]*[^a-z]onsubmit\s*=!is";

$UNSAFE_HTML[] = "!<[^>]*[^a-z]onunload\s*=!is";
$UNSAFE_HTML[] = "!<[^>]*[^a-z]onerror\s*=!is";
$UNSAFE_HTML[] = "!<[^>]*[^a-z]onclick\s*=!is";

function unsafe_html($html) {
  global $UNSAFE_HTML;
  $html = html_entities($html, ENT_COMPAT, 'ISO-8859-1')
  foreach ( $UNSAFE_HTML as $match ) {
    if( preg_match($match, $html, $matches) ) {
      return $match;
    }
  }
  return false;
}
?>
```

Wenn Sie es zulassen, Text direkt in Tags zu integrieren (wie im vorhergehenden Beispiel), möchten Sie vielleicht sogar jeden Text zurückweisen, der aussieht wie eine clientseitige Skriptsprache:

```
$UNSAFE_HTML[] = "!onabort\s*=!is";
$UNSAFE_HTML[] = "!onblur\s*=!is";
$UNSAFE_HTML[] = "!onchange\s*=!is";
$UNSAFE_HTML[] = "!onfocus\s*=!is";
$UNSAFE_HTML[] = "!onmouseout\s*=!is";
$UNSAFE_HTML[] = "!onmouseover\s*=!is";
$UNSAFE_HTML[] = "!onload\s*=!is";
$UNSAFE_HTML[] = "!onreset\s*=!is";
$UNSAFE_HTML[] = "!onselect\s*=!is";
$UNSAFE_HTML[] = "!onsubmit\s*=!is";
$UNSAFE_HTML[] = "!onunload\s*=!is";
$UNSAFE_HTML[] = "!onerror\s*=!is";
$UNSAFE_HTML[] = "!onclick\s*=!is";
```

Es mag verführerisch sein, `magic_quotes_gpc` in der `php.ini`-Datei zu aktivieren. `magic_quotes` fügt eingehenden Daten automatisch Anführungszeichen hinzu. Ich aktiviere `magic-quotes` allerdings nicht, da der Eindruck von Sicherheit entsteht – der aber trügerisch ist, wie einfache Beispiele (z.B. das vorhergehende) zeigen.

Bei der Datenvalidierung (speziell bei Daten, die angezeigt werden sollen) stehen Sie oft vor der Alternative, Daten beim Speichern (Inbound) oder der Anzeige (Outbound) zu filtern und zu konvertieren. Generell ist das Filtern der eingehenden Daten effizienter und sicherer. Das Filtern eingehender Daten muss nur einmal geschehen, und man läuft nicht Gefahr es zu vergessen, wenn die Daten an verschiedenen Stellen angezeigt werden. Nachfolgend aber auch zwei Gründe, die für die Filterung von ausgehenden Daten sprechen können:

- Sie brauchen sehr anpassbare Filter.
- Die Filter ändern sich schnell.

Im zweiten Fall ist es vermutlich besser, bekannte bösartige Daten beim Eintreffen (Inbound) zu filtern und einen weiteren Filter einzubauen, wenn die Daten ausgegeben werden.

Weitergehende Datenvalidierung

Die Anzeige von unvalidierten Daten auf Webseiten ist nicht der einzige Ort, an dem Sicherheitslücken auftreten können. Alle Daten, die vom User empfangen werden, sollten vor der Weiterverarbeitung getestet und gesäubert werden. In Datenbankabfragen z.B. ist es wichtig, sauber Anführungszeichen zu setzen, bevor ein `Insert` durchgeführt wird. Es gibt bequeme Funktionen, die Sie bei dieser Arbeit unterstützen.

Ein Beispiel sind so genannte SQL-Infiltrierungsattacken (SQL Injection, SQL-Injektionsangriffe). Diese Attacken funktionieren ungefähr so: Sie haben beispielsweise folgende Abfrage:

```
$query = "SELECT * FROM users where userid = $userid";
```

Falls `$userid` nicht validierte eingehende Daten enthält, könnte ein bösartiger User Folgendes infiltrieren:

```
$userid = "10; DELETE FROM users;";
```

Da MySQL (wie viele anderen Datenbanksysteme) mehrere Abfragen in einer Zeile unterstützt, haben Sie – wird dieser Wert ungetestet verwendet – die Tabelle `users` verloren. Dies ist nur ein Beispiel von vielen möglichen Variationen dieser Angriffe. Die Moral von der Geschichte ist: Sie sollten immer alle Daten in Abfragen validieren.

3.4 Wann nutzt man Ausnahmen?

Es gibt verschiedene Ansichten darüber, wann und wie Ausnahmen eingesetzt werden sollten. Einige Programmierer sind der Meinung, dass Ausnahmen nur für schwere oder potenziell schwere Fehler eingesetzt werden sollten. Andere Programmierer nutzen Ausnahmen als grundlegende Komponente zur Beeinflussung des Programmablaufs. Die Sprache Python ist ein guter Repräsentant der zweiten Technik: In Python werden Ausnahmen üblicherweise zur normalen Ablaufkontrolle eingesetzt.

Letztendlich ist dies lediglich eine Frage des Stils. Ich bin von Natur aus Sprachen gegenüber misstrauisch, die versuchen, einen bestimmten Stil zu erzwingen. Die folgende Liste hilft Ihnen vielleicht bei der Entscheidung, wo und wann Sie Ausnahmen verwenden möchten:

- Ausnahmen sind eine Syntax der Kontrollstruktur, genau wie `if`, `else`, `while` und `foreach`.
- Die Anwendung von Ausnahmen für nicht lokale Flusskontrolle (z. B. weite Sprünge aus einem Block in einen anderen Bereich) resultiert in nicht intuitiven Codes.
- Ausnahmen sind etwas langsamer als traditionelle Formen der Ablaufkontrolle.
- Mit Ausnahmen läuft man Gefahr, Speicherlecks zu erhalten.

3.5 Lesetipps

Eine maßgebende Quelle über XSS und bösartige HTML-Tags ist CERT Advisory CA-2000-02, zu erhalten auf der Seite *www.cert.org/advisories/CA-2000-02.html*.

Da Ausnahmen ein neues Feature von PHP sind, sind die besten Referenzen über ihren Einsatz wahrscheinlich Bücher über Java und Python. Die Syntax in PHP ist vergleichbar zu der in Java und Python (obwohl es leichte Unterschiede insbesondere zu Python gibt).