php | tek - Chicago, May 16-18, 2007

Exceptional PHP

When Good Code Goes Bad

Jeff Moore
http://www.procata.com/

I Want to Improve How My Code Deals With Errors

What Impression Does this Make?

* **Warning**: mysql_connect(): Too many connections in **/www2/anonymous/php/_Database.inc** on line **18** Too many connections

What Can Go Wrong?

* Programming errors
* Violation of preconditions
* Violation of postconditions
* Violation of invariants
* Invalid input
* Resource failures
* Hardware failures
* Who knows what else?

Formally Defining Correctness

- Precondition
  - Must be true before the code can be Executed
- Invariants
  - Must be true while the code is Executed
- Postconditions
  - Must be true after the code is done executing

Basic Strategies

* Suppress
* Inform
* React
* Propagate

Suppress

- Not everything is an error
- Some error conditions don't matter
- Best practices
  - Don't suppress legitimate errors
  - Document your intention to suppress

Inform

* Some errors are expected
* Inform appropriate party via
  * API
  * UI
  * Log

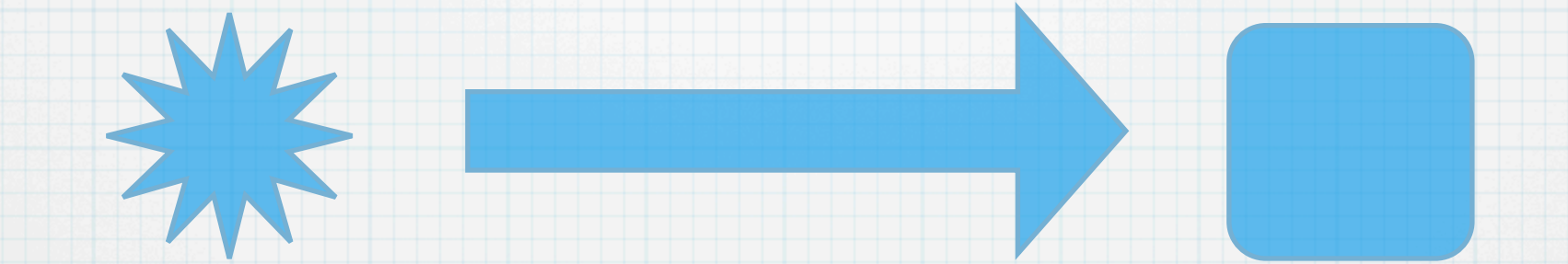
React

* Rollback
* Clean up
* Repair
* Try something else

React

```
ob_start();
try {
    show_feed_sidebar();
    ob_end_flush();
} catch (Exception $e) {
    ob_end_clean();
    echo “Unavailable”;
}
```

Propagate



Context Matters

* Low Level
  * Not enough perspective to know what to do
* High Level
  * Too late and too far away to react

Selecting a Strategy

* Strategies are not mutually exclusive
* Context matters
* Banking versus bookmarking

Detecting Errors

* Its all about the If

Defensive Programming

* Check everything, trust nothing
* Check pre-conditions and invariants
* Most errors conditions never occur
  * More code
  * Slower code
  * Error checking code can have bugs, too

Trusting Your Caller

* Trust your caller to get it right
* Document pre-conditions and invariants
* Faster code
* Leaner code
* Harder to debug

Which Approach?

- Defend
  - Defend yourself at package boundaries
  - Banking
- Trust
  - Trust within your own package
  - Don't trust another organization
  - Bookmarking

Defending Exit Points

* Every function or method call is an exit point
* Do you verify the post conditions and invariants after the call?
  * A.K.A Check return values for error codes?
* How do you indicate an error?

Communicating Errors

* Function return value
* Exceptions
* Pass by reference parameters
* Error callbacks
* User error handlers
* $php_errormsg

Return Values

* FALSE on error
  * `return FALSE;`
* Error codes
  * `return MY_ERROR_CODE;`
* Error Objects
  * `return new PEAR_Error();`

Return Values

* Overloading return values can be confusing
  * strpos FALSE versus 0
* Hard to propagate
  * One omission breaks the chain
  * Extra propagation code inline

Propagating Error Return Values

```
function store($filename, $data) {
  $dir = dirname($filename);
  if ((!@is_dir($dir) && !mkdir($dir, 0777, TRUE)) ||
    !($tmpfile = tempnam($dir, '.tmp')) ||
    !file_put_contents($tmpfile, $data) ||
    !rename($tmpfile, $filename)) {
      return FALSE;
  }
  return TRUE;
}
```

Refactored

```
function store($filename, $data) {
  $dir = dirname($filename);
  if ((!@is_dir($dir)) {
    if (!mkdir($dir, 0777, TRUE) {
      return FALSE;
    }
  }
  $tmpfile = tempnam($dir, '.tmp');
  if ($tmpfile) {
    if (!file_put_contents($tmpfile, $data)) {
      return FALSE;
    }
    return rename($tmpfile, $filename);
  }
  return FALSE;
}
```

Suppressing Returned Error Codes

* Easy to Suppress
* Hard to distinguish intentional from accidental suppression

Error Handlers

* `set_error_handler('handler');`
* Primary PHP error notification mechanism
* Global Variable

Typical Error Handler

```
function handleError(
    $type, $message, $file, $line, $context) {

    if ( ( $type & error_reporting() ) != $type ) {
        return;
        // Ignore this error
        // because of error suppression
    }

    // What do you do here?
}
```

Error Suppression Operator (@)

* `@operation();`
* Same as
* ```
  $level = error_reporting(0);
  @operation();
  error_reporting($level);
  ```
  ```

Be Careful Not to Over Suppress

* `@include_once “file.php”;`
* `@store($filename, $data)`

Prevent Instead of Suppress

* Prevent
  * `$x = isset($var) ? $var : NULL;`
* suppress
  * `$x = @$var;`

Exceptions

* The class is the error code
* Great ability to convey information

Defining An Exception

```
class MyBad extends Exception {
    protected $extraInfo;

    function __construct($message,
            $extraInfo) {
        $this->extraInfo = $extraInfo;
        parent::__construct($message);
    }
    function getExtraInfo() {
        return $this->extraInfo;
    }
}
```

Throwing an Exception

* We only need to define code at the point where the error occurs.
* `throw new Exception("oops.");`

Catching An Exception

* And define code where we will handle the error
* ```
  try {
      ...
  } catch (Exception $e) {
      react_to_problem();
  }
  ```
  ```

Control Flow

* No extra error propagation code required
* Minimizes the distance between the event and the reaction
* Allows the reaction to take place in the appropriate context

Remember This?

```
function store($filename, $data) {
  $dir = dirname($filename);
  if ((!@is_dir($dir)) {
    if (!mkdir($dir, 0777, TRUE) {
      return FALSE;
    }
  }
  $tmpfile = tempnam($dir, '.tmp');
  if ($tmpfile) {
    if (!file_put_contents($tmpfile, $data)) {
      return FALSE;
    }
    return rename($tmpfile, $filename);
  }
  return FALSE;
}
```

Exceptions Reveal the Intent of the Code

```
function store($filename, $data) {
    $dir = dirname($filename);

    if (!is_dir($dir)) {
        mkdir($dir, 0777, TRUE);
    }
    $tmpfile = tempnam($dir, '.tmp');
    file_put_contents($tmpfile, $data);
    rename($tmpfile, $filename);
}
```

Building Firewalls

* Convert errors from low level errors to high level errors
* Add information at each stage
* Build firewalls at system boundaries

Inappropriate Level of Abstraction

* File system driven cache:
  * `throw new DiskFullException();`
* Database driven cache:
  * `throw new DbConnectionException();`
* Should a Feed class that uses this cache API have to know about these exceptions?

Wrapping with PEAR_Exception

```
class CacheWriteException extends
    PEAR_Exception {}

try {
    // Write to the cache with plugin
} catch (Exception $e) {
    throw new CacheWriteException(
     “Item Not Cached”, $e);
}
```

Control Flow

* Don't use exceptions for "normal" control flow
* Consider a find() method:
  * Throw on Not Found?
  * Return FALSE on Not Found?

trigger_error

* Leave this communication channel to PHP

$php_errormsg

* Don't compare it to anything
* Influenced by too many ini parameters
* User error handlers cannot use

Error Levels & Severity

* E_ERROR
* E_WARNING
* E_PARSE
* E_NOTICE
* E_CORE_ERROR
* E_CORE_WARNING
* E_COMPILE_ERROR
* E_COMPILE_WARNING
* E_USER_ERROR
* E_USER_WARNING
* E_USER_NOTICE
* E_STRICT
* E_RECOVERABLE_ERROR
* E_DEPRECATED?

Classifying Errors by Severity

* Makes you determine severity when the error is detected
* Ignores higher level context
* Very hard to get it right

Hide Errors

- Don't reveal your internal error messages to the user
  - `php_flag display_errors off`
  - `php_flag display_startup_errors off`

Log It

* Built in error logging
  * `php_flag log_errors on`
  * `php_value error_log /path/to/log`
* More efficient than your own error handler
* Can log errors that user space error handlers can't

error_reporting

* `error_reporting(E_ALL);`
* `error_reporting(E_ALL | E_STRICT);`
* `error_reporting(E_ALL ^ E_NOTICE);`
* `error_reporting(E_ALL ^ E_NOTICE ^ E_WARNING);`

Fatal Errors

* E_ERROR
* E_PARSE
* E_CORE_ERROR
* E_COMPILER_ERROR

Catch Fatal Errors

* Use register_shutdown_function to detect fatal errors
* Show a 500 page
* Things may be squirrelly in your shutdown handler?
* Thanks, Derick

Questions?