Testbarer Code in PHP

Well, there are no tricks to writing tests, there are only tricks to writing testable code.

Miško Hevery

- Trennen von Zuständigkeiten
- Dekomposition
- Dependency Injection
- globaler Status
- Entwurfsprinzipien
- Heuristiken

Trennen von Zuständigkeiten

- Single-Responsibility Principle
- Separation-of-Concerns
- Vermeide Klassen mit mehr als einer Aufgabe
- Trennen von Service-Objects und Value-Objects
  - Value-O.: erzeugen mit `new`, werden *nicht* gemocked, oft *imutable* (Bsp. EmailAddress, CreditCard)
  - Service-O.: nicht `new`, erzeugen durch Factory, können in Tests gemocked werden (Bsp. MailServer, CreditCardProcessor)

Trennen von Zuständigkeiten

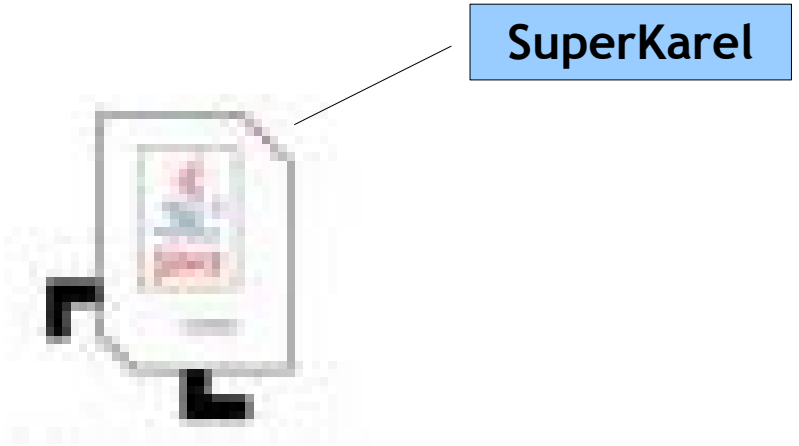
Beispiele für Zuständigkeiten:

- etwas wissen: Daten/Konzept kennen
- etwas können: Steuerung od. Kontrolle
- etwas erzeugen

Trennen von Zuständigkeiten

Dieses Prinzip lässt sich auch auf Methoden und Funktionen anwenden.

Dekomposition



Stanford University (YouTube):
Course Programming Methodology (CS106A)

Dekomposition

```
abstract class SuperKarel {
    abstract public function run();

    public function beepersPresent() { /* ... */ }

    public function pickBeeper() { /* ... */ }

    public function putBeeper() { /* ... */ }

    public function move() { /* ... */ }

    public function turnArraound() { /* ... */ }
}
```

(keine) Dekomposition

```
class DoYourThing extends SuperKarel {
    public function run() {
        $this->move();
        while ($this->beepersPresent()) {
            $this->pickBeeper();
            $this->move();
            $this->putBeeper();
            $this->putBeeper();
            $this->turnArraound();
            $this->move();
            $this->turnArraound();
        }
        $this->move();
        while ($this->beepersPresent()) {
            $this->pickBeeper();
            $this->turnArraound();
            $this->move();
            $this->putBeeper();
            $this->turnArraound();
            $this->move();
        }
    }
}
```

(mit) Dekomposition

```
class OurDoubleBeepers extends SuperKarel {
    public function run() {
        $this->move();
        $this->doubleBeepersInPile();
        $this->moveBackward();
    }
}
```

(mit) Dekomposition

```
class OurDoubleBeepers extends SuperKarel {
    public function run() {
        $this->move();
        $this->doubleBeepersInPile();
        $this->moveBackward();
    }

    public function doubleBeepersInPile() {
        while ($this->beepersPresent()) {
            $this->pickBeeper();
            $this->putTwoBeepersNextDoor();
        }

        $this->movePileNextDoor();
    }

    public function moveBackward() {
        $this->turnAround();
        $this->move();
        $this->turnAround();
    }
}
```

(mit) Dekomposition

```
class OurDoubleBeepers extends SuperKarel {
    public function run() { /* … */ }

    public function doubleBeepersInPile() {/* … */ }

    public function moveBackward() { /* … */ }

    public function putTwoBeepersNextDoor() {
        $this->move();

        while ($this->beepersPresent()) {
            $this->moveOneBeeperBack();
        }

        $this->moveBackward();
    }

    public function movePileNextDoor() {
        $this->move();
        $this->putBeeper();
        $this->putBeeper();
        $this->moveBackward();
    }
}
```

(mit) Dekomposition

```
class OurDoubleBeepers extends SuperKarel {
    public function run() { /* … */ }

    public function doubleBeepersInPile() {/* … */ }

    public function moveBackward() { /* … */ }

    public function putTwoBeepersNextDoor() { /* … */ }

    public function movePileNextDoor() { /* … */ }

    public function moveOneBeeperBack() {
        $this->pickBeeper();
        $this->moveBackward();
        $this->putBeeper();
        $this->move()
    }
}
```

Dependency Injection

Goldene Regel des `new`-Operators:

- Ok für Domänen-Klassen, nicht für Services
- Ok in Tests und spezialisierten Erzeuger-Klassen, nicht in der Business-Logik

Dependency Injection

```
class FriendFinder {

   public function __construct() {
      $this->search   = new Search();
      $this->strategy = Strategy::create();
   }

}
```

- schwer zu testen (2 Abhängikeiten)
- Don't do work in contructor!
- keine Test-Doubles möglich
- wird bei jedem Test ausgeführt
- Verkompliziert Test-Setup

Dependency Injection

```
class FriendFinder {

   public function __construct(Search $search, Strategy $strategy) {
      $this->search   = $search;
      $this->strategy = $strategy;
   }

}
```

- ermöglicht Test-Doubles
- nicht jeder Test braucht das volle Brett
- „Ask for things, don't look for things.“

Dependency Injection

- DI by Constructor (voriges Bsp.)
- DI by Setter
  - `$search->setSearch(Search $s)`
  - `$search->setStrategy(Strategy $s)`
- DI by Interface
  - http://martinfowler.com/articles/injection.html

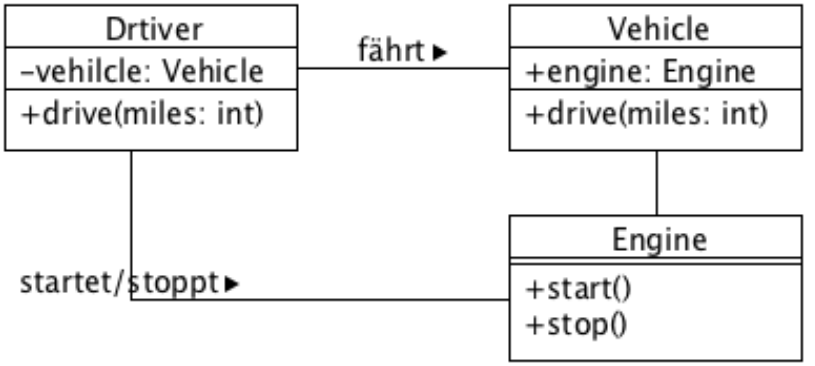
Gesetz von Demeter

Objekte sollten nur mit Objekten in ihrer unmittelbaren Umgebung kommunizieren.

Wikipedia

Gesetz von Demeter

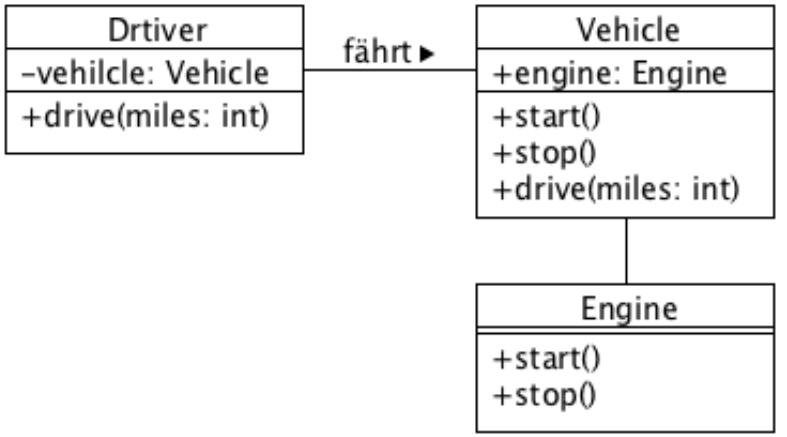
```
class Driver {

    public function drive($miles) {
        $this->vehicle->engine->start();
        $this->vehicle->drive($miles);
        $this->vehicle->engine->stop();
    }

}
```



- schwer testbar, braucht immer Engine-Objekt
- Driver eng an Engine gekoppelt
- interner Status von Vehicle offen gelegt
- zirkuläre Abhängigkeit

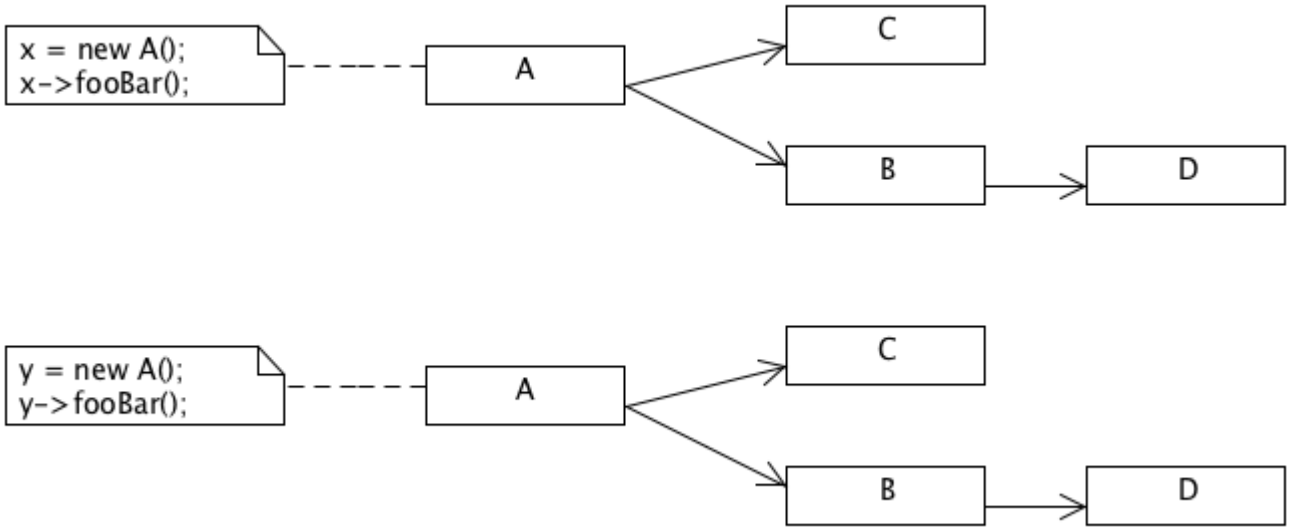
Gesetz von Demeter

```
class Driver {

    public function drive($miles) {
        $this->vehicle->start();
        $this->vehicle->drive($miles);
        $this->vehicle->stop();
    }

}
```



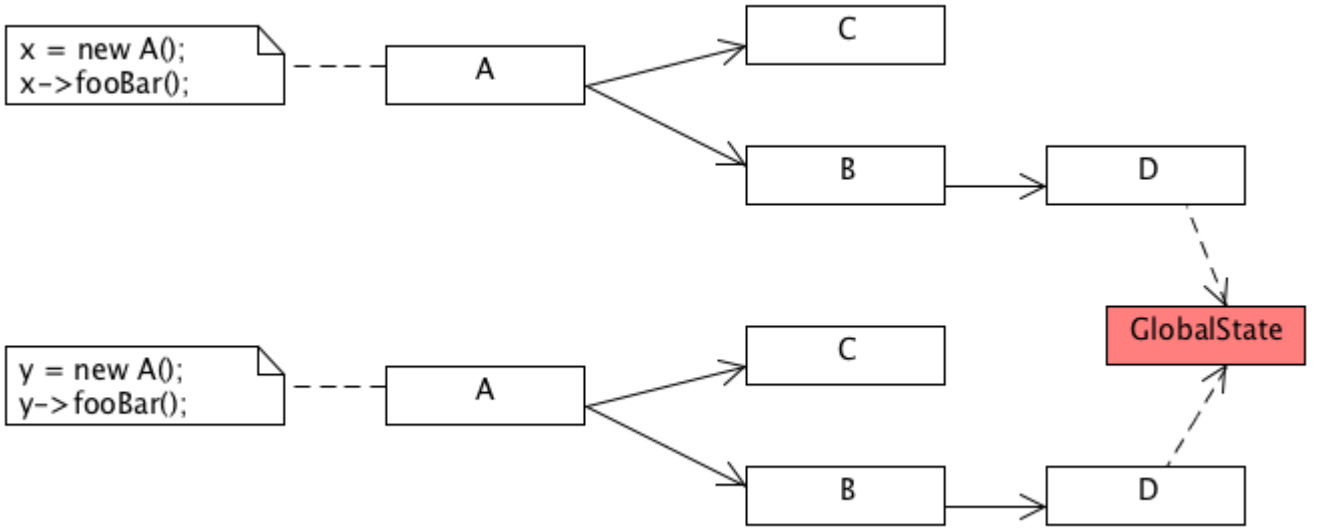
- leichter testbar
- Driver und Engine entkoppelt → leichter wartbar
- weniger fehleranfällig

globaler Status



x == y

globaler Status



x != y

globaler Status

- gleiche Operation mit gleichem Input → immer gleiches Ergebnis!
- nicht notwendigerweise mit globalen Stati
- versteckte globale Stati:
  - $_GET, $_POST, $_SESSION, ...
  - statische Methoden
  - Singleton
  - Registry

$_GET, $_POST, $_SESSION, …

- sehr schlecht zu testen
  - gibt's auf CLI einfach nicht :-(
- nie direkter Zugriff in der Business-Logik
- möglichst weit weg abstrahieren
  - keine Singletons verwenden!
- Interfaces definieren :-)
  - das kann *gemocked* werden

statische Methoden

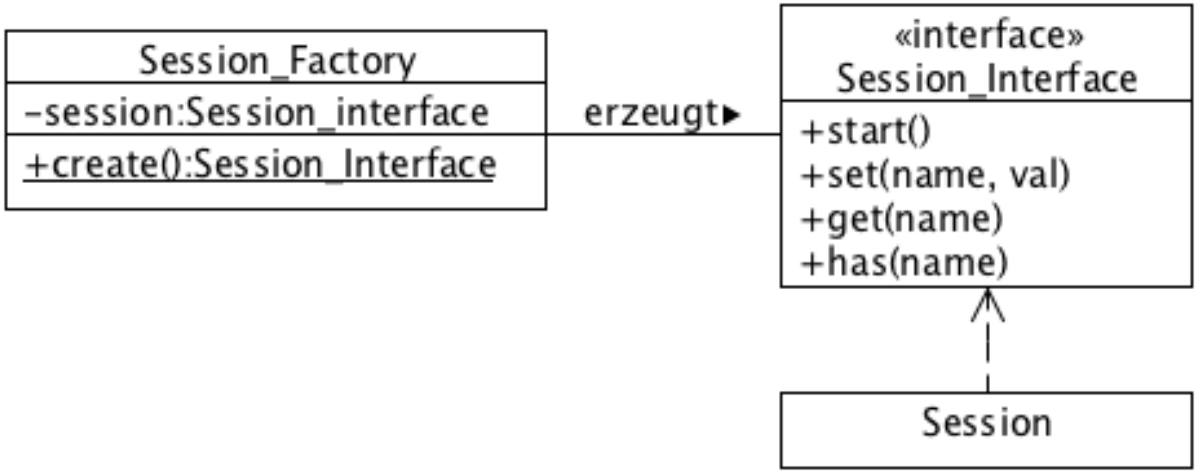
- nicht immer böse → `Math::abs($int)`
  Ok für einfache Operationen ohne Abhängigkeiten:
- Immer böse, wenn State involviert →
  Rückgabe von verschiedenen Werten bei gleicher Eingabe! `Auth::isLoggedIn($userId)`
- Klassen nur mit statischen Methoden →
  prozeduraler Code

Singleton

- sind globale Variabeln im Schafspelz
- transitiv rekursiv global!
- vermeide Singletons wann immer möglich
- meist die Folge von schlechtem Design
- sehr schlecht testbar

Singleton

Alternative: Factory



- nicht Objekt erzwingt eine Instanz
- Applikation erzeugt nur eine Instanz
- jeder Test *kann* eigene Instanz erzeugen

Registry

- Nur für Configurations-Werte benutzen!
- Sollte kein Singleton sein!

  statische Methoden reichen völlig :-)

Entwurfsprinzipien

- Einfachheit vor Allgemeinanwendbarkeit → mache normale Dinge einfach, besondere möglich
- Prinzip der minimalen Verwunderung → erstaunliches meist schwer verständlich
- Don't repeat yourself / Once and once only
- Prinzip der einzelnen Zuständigkeit

Entwurfsprinzipien

- keien zirkulären Abhängigkeiten (s. Bsp. Driver)
- Liskov'sches Substitutionsprinzip → Unterklassen anstelle Oberklasse einsetzbar
- Prinzip der Schnittstellen-Abtrennung → nicht von unbenötigten Diensten abhängen
- Konvention vor Konfiguration

Heuristiken

- Klassen & Objekte
  - genau eine Abstraktion (Zuständigkeit)
  - zusammengehörige Daten u. deren Verhalten kapseln
  - wenn unschönes, dann nur in einer Klasse
  - keine Gott-Klassen
  - Klassen sollten nicht von deren Benutzern abhängen
  - Klassen sollten nicht wissen worin sie enthalten sind
  - modelliere möglichst nah an der realen Welt
  - vermeide lange Argumentlisten → aggregieren

Heuristiken

- Vererbung & Delegation
  - Vererbung nur zur Spezialisierung
  - Oberklasse sollte nichts über Unterklasse wissen
  - Vermeide explizites untersuchen eines Typs
  - abstrakte Klassen nur als Basis der Hierarchie
  - Methoden nicht mit leerer Implementierung überschreiben (s. Liskov!)
  - bevorzuge Interfaces vor abstrakten Klassen

Mehr Stoff

- Writing Testable Code
- Guide to Writing Testable Code
- Programming Methodology Course Stanford
- Entwurfsmuster [Gamma et al.]
- PHP Design Patterns [Stephan Schmidt]
- Patterns kompakt [Karl Eilebrecht et al.]


Singletons are really, really
(and I mean really)
EVIL
CAUTION
DANGER
EXTREMELY FLAMMABLE
VAPOR MAY EXPLODE IF HEATED