__studio_krijst

```
1
2
3  <?inhoudsopgave
4     $abstract_classes_en_interfaces
5         ->interfaces()
6         ->abstract_classes();
7
8     $traits;
9
10    $design_patterns
11        ->dependency_injection()
12
13
14        ->object_pool()
15        ->singleton()
16
17        ->super_global_singleton()
18        ->adapter()
19
20        ->decorator()
21
22        ->factory()
23
24        ->observer()
25
26
27        ->strategy()
28
29        ->lazy_loading();
30
31     $test_driven_development()
32
33        ->php_unit()
34        ->setup()
35
36
37        ->tests_uitvoeren()
38
39        ->assertions();
40
41     $case;
42
43
```

https://dev-master.ninja/php-master

3?>

<?abstract_classes_en_interfaces Abstract classes en interfaces zijn middelen die bedoeld zijn om de programmeur te “dwingen” bepaalde methods of properties te gebruiken. Op die manier kun je vooraf bepalen hoe bepaalde classes gebruikt dienen te worden of in een applicatie opgenomen dienen te worden.

<?interfaces Een interface kun je zien als een soort van Inhoudsopgave voor de afgeleide klasse(n). Alle methodes die in een interface gespecificeerd staan **MOETEN** in de afgeleide klasse geïmplementeerd worden. Met andere woorden: een interface is een overeenkomst tussen ongerelateerde objecten voor het uitvoeren van dezelfde functionaliteit. Een interface stelt je in staat om aan te geven dat een object een bepaalde functionaliteit moet bezitten, maar het bepaalt niet hoe het object dat moet doen. De afgeleide klasse is dus vrij om de hele implementatie te doen, zolang hij maar voldoet aan de beschrijving die de interface afdwingt. Met name als je in team-verband werkt, kunnen interfaces handig zijn om de juiste implementaties af te dwingen.

<?abstract_classes Een abstract class is een klasse met of zonder eigen properties en een aantal methodes die gedeeltelijk de functionaliteit van de klasse bepalen maar tegelijkertijd een deel van de functionaliteit onbepaald laat. Het onbepaalde gedeelte zijn de abstracte methodes en deze dienen uitgewerkt te worden in de afgeleide klasse die deze abstracte klasse “extend”.

Oké, dat was een hele mond vol, maar wat betekent het nu? Dat kunnen we het beste aan de hand van een aantal voorbeelden onderzoeken.

<?interface

Voer onderstaande code uit en kijk wat er gebeurt. Refactor vervolgens de _construct en de getNaam() functies in de klasse en kijk wat er nu gebeurt.

```php
<?php


interface MijnInterface {
    public function getNaam();
    public function __construct($naam);
}


class MijnClass implements MijnInterface {

    private $naam;

    public function getNaam() {
        return($this->naam);
    }

    public function __construct($naam) {
        $this->naam = $naam;
    }

}

$goed = new MijnClass(“Hans van der Code”);
echo $goed->getNaam();

```

<?samengevat

- Alle methodes moeten public zijn.
- Alle methodes die beschreven zijn in de interface MOETEN geïmplementeerd worden
- Er kan eventueel een __construct gedeclareerd worden (niet altijd even handig)

<?opdrachten

Schrijf een interface UserInterface die een afgeleide User class “dwingt” om public getters ensetters te implementeren op de (private) velden: firstname, lastname, email, password.

<?abstract_classes Voer onderstaande code uit en kijk wat er gebeurt. Refactor vervolgens de _construct en de getBandName() functies in de klasse en kijk wat er nu gebeurt.

```php
<?php

abstract class OldSkoolRockBand {

    abstract protected function getBandName();
    abstract public function getBandMembers();

    public function showBand() {
        echo “<pre>”;
        echo “<h2>”. $this->getBandName() .“</h2>”;
        var_dump( $this->getBandMembers() );
        echo “</pre>”;
    }

}

class Beatles extends OldSkoolRockBand {

    protected function getBandName() {
        return(“The Beatles”);
    }

    public function getBandMembers() {
        return([“John”, “Paul”, “George”, “Ringo”]);
    }

}

class RollingStones extends OldSkoolRockBand {

    protected function getBandName() {
        return(“The Rolling Stones”);
    }

  //  public function getBandMembers() {
  //    return([“Mick”, “Keith”, “Charlie”,
  //              “Ron”, “Bill”]);
  //  }
}

$beatles = new Beatles();
$beatles->showBand();

$stones = new RollingStones();
$stones->showBand();
```

<?samengevat

- Alle methodes die als “abstract” beschreven zijn in de abstract class **MOETEN** geïmplementeerd worden
- een abstract class mag **NIET** geïnstantieerd worden
- De zichtbaarheid van een abstract functie moet dezelfde of een minder restrictieve zijn (bijvoorbeeld: abstract: protected, implementatie: protected of public)

<?opdrachten

Herschrijf de interface UserInterface naar een abstract class die de afgeleide User class nog steeds”dwingt” om public getters en setters te implementeren op de
(private) velden: firstname, lastname, email, password
Maak ook een public functie faciliteert die de voor- en achternaam van de gebruiker toont.

<?interface_vs_abstract_class

De verschillen tussen de interaface en de abstract class nog even op een rijtje:

<?interface

- Multiple inheritance
- Geen props/variabelen
- Alleen “signature functions” geen uitgewerkte functies
- Alleen public members

<?abstract_class

- Single inheritance
- Kan props/variabelen bevatten
- Naast “abstract signature functions” ook uitgewerkte functies
- Alle protectie niveau’s zijn toegestaan

<?traits Traits zijn constructies die het mogelijk maken code te hergebruiken die in principe door het "single inheritance" concept van PHP mogelijk in meerdere klasses opgenomen zou moeten worden. In diverse andere talen (oa. C#) is het mogelijk om een klasse meerdere basisklassen te laten overerven:

```
C#: class DataClass: TableClass, DatabaseClass {
```

Dit is onmogelijk in PHP. In PHP zou je dan snel geneigd zijn om "class chaining" te gaan toepassen:

```
class DataClass extends TableClass {
      class TableClass extends DatabaseClass {
```

etc..

Maar dit is een vreselijk slecht idee. Als het al nodig is om je code op een dusdanige manier te construeren, waar er dan voor dat je niet meer dan 3 niveau's diep gaat. Je code wordt namelijk dusdanig onoverzichtelijk dat je onbedoeld rare situaties kunt creëren (denk aan circulaire referenties e.d.).
Om toch een zekere mate van flexibiliteit en code-reusability te implementeren, zijn de zogenaamde "traits" in het leven geroepen. Deze constructies zijn erg handig, echter waak ervoor dat je niet te veel op traits gaat leunen, want **GEEN ENKELE** andere taal heeft deze tools.
Heel eigenlijk is het een beetje valsspelen van PHP en maakt het gebruik van de flexibiliteit van een scripttaal t.o.v. een gecompileerde taal. Ook past het bijna niet in de (huidige) filosofie van PHP om toch strikter om te gaan met oa. Datatypes en functies. Maar goed, vooralsnog werken ze prima en zijn ze zeer zeker de moeite van het bestuderen en implementeren waard.

<?gebruiken? Primair als een technische laag: vermijd zoveel
mogelijk het gebruik van functionele componenten in de traits. Functionele componenten horen thuis in classes.

<?of_niet? Don't trait away... Omdat het concept wellicht iets te handig blijkt te zijn, kan het al heel vlug leiden tot stevig onderling verweven code

```php
<?php


 trait generic {

    public function log($message) {
        echo $message . “<br>”;
    }

 }


 class actionOne {
     use generic;

     public function something() {
         $this->log(“something”);
     }
 }

 class actionTwo {
     use generic;

     public function somethingElse() {
        $this->log(“somethingElse”);
    }

 }


 $one = new actionOne();
 $two = new actionTwo();

 $one->log(“Hi, dit is direct”);
 $one->something();

 $two->somethingElse();

```

<?opdrachten

Schrijf 2 classes die informatie uit een mysql db ophalen (Client en Order), maar de daadwerkelijke interacties met de database (select etc) lopen via een (dezelfde) trait.

<?design_patterns

Design patterns zijn (standaard) oplossingen voor veel voorkomende software problemen. De term komt van het boek "*Design Patterns: Elements of Reusable Object Oriented Software*" (Pearson Education, oktober 1994) van de zogenaamde "Gang of Four" (Erich Gamma, Richard Helm, Ralph Johnson en John Vlissides) waarin deze oplossingen uitgebreid beschreven worden.

Tegenwoordig zijn er een heleboel van deze patterns gedefinieerd, en het is een beetje onduidelijk wie of wat er de scepter over zwaait. Verder zijn ze vaak het onderwerp van verwoede discussies op diverse fora voor software engineering over het juiste gebruik en wat nou wel of niet een "pattern" genoemd kan en mag worden.

Ik ben hier heel pragmatisch in: of je ze strikt en juist gebruikt, vind ik niet eens zo heel belangrijk, het primaire doel is: "Wordt mijn probleem opgelost?" En als een pattern, of een deel van een pattern, daarbij kan helpen, dan is dat een prima en mogelijk charmante oplossing. Ik gebruik zelf regelmatig een aantal patterns, maar beslist niet naar de letter van het boek. Ik vind ze vooral inspirerend van aard: soms een ietwat andere approach of visie op een specifiek probleem bestuderen kan er voor zorgen dat je coding-skills naar een hoger plan getild worden, en dat is nooit verkeerd.

We zullen in dit hoofdstuk een aantal patterns bestuderen en uitwerken - lang niet alle patterns zijn heel duidelijk in wat ze nou precies doen en wat ze nou eigenlijk oplossen. De meest bruikbare heb ik hier op een rijtje gezet en uitgewerkt aan de hand van een (eenvoudig) voorbeeld.

Soms gebruik je ze zelfs al zonder het te weten. Een "dependency injection" bijvoorbleed. Dat is een methode die je eigenlijk al zonder nadenken implementeert. De theorie overstijgt ook hier de taal: als je ze in PHP kunt toepassen, kun je ze ook in andere talen gebruiken.

Grofweg kun je de patterns in vier categorieën indelen:

- **Creational** - design patterns die gericht zijn op het "maken" van objecten
- **Structural** - patterns die zich richten op de structuur van je applicatie
- **Behavioral** - patterns die in de 'uitvoering' genesteld zijn
- **De rest** - alle andere varianten

De volgende patterns zijn mijns inziens de meest bruikbare in PHP: *Dependency Injection, Object Pool, Singleton (en een alternatieve Singleton), Adapter, Decorator, Factory, Observer, Strategy* en *Lazy Loading*.

<?dependency_injection dit pattern helpt je om zogenaamde "tight coupling" te voorkomen. In principe is het zo dat je in je klasse-structuur onderlinge afhankelijkheden zoveel mogelijk moet zien te voorkomen. De theorie is natuurlijk veel mooier dan de praktijk en zul je vaak toch afhankelijkheden tegenkomen. Dan kun je de depency injection toepassen: je "injecteert" de afhankelijkjheden vanuit de ene klasse in de andere - naar alle waarschijnlijkheid gebruik je dit al dagdagelijks (zonder dat je het weet).

Onderstaand voorbeeld overtreedt het "Single Reposibility" principe:

```
1    <?php
2
3
4     class Author {
5
6      private $firstname;
7      private $lastname;
8
9      public function __construct($firstname, $lastname)
10     {
11         $this->firstname = $firstname;
12         $this->lastname = $lastname;
13     }
14
15    }
16
17
18    class Book {
19
20      private $title;
21      private $author;
22
23      public function __construct($first,$last, $title)
24      {
25          $this->title = $title;
26          $this->author = new Author($first, $last);
27      }
28    }
```

We zien op regel 26 dat we de klassen Book en Author ineens samenvoegen, waardoor we eigenlijk nooit meer een boek zonder auteur kunnen hebben. Het "Single Responsibility" principe verwacht dat we Book en Author volledig lost van elkaar benoemen. Maar functioneel horen die twee natuurlijk wel bijelkaar.

Veel betere code krijgen we door “dependency injection”, kijk maar:

```
<?php


 class Author {

  private $firstname;
  private $lastname;

  public function __construct($firstname, $lastname)
  {
      $this->firstname = $firstname;
      $this->lastname = $lastname;
  }

 }


 class Book {

   private $title;
   private $author;

   public function __construct($title,
                               Author $author)
   {
       $this->title = $title;
       $this->author = $author;
   }
 }

$author = new Author(“Stephen”, “King”);
$book = new Book(“Eng boek”, $author);
```

Door de code zo te refactoren, bereiken we weliswaar hetzelfde, maar we “injecteren” het kant-en-klare Author-object nu in Book.

Het zou nog mooiere en duidelijkere code opleveren als we gebruik maken van een zogenaamde setter.

```php
<?php


 class Author {

  private $firstname;
  private $lastname;

  public function __construct($firstname, $lastname)
  {
      $this->firstname = $firstname;
      $this->lastname = $lastname;
  }

 }


 class Book {

   private $title;
   private $author;

   public function setAuthor(Author $author) {
     this->author = $author;
   }

   public function __construct($title)
   {
       $this->title = $title;
   }
 }

$author = new Author(“Stephen”, “King”);
$book = new Book(“Eng boek”);
$book->setAuthor($author);

```

Nu hebben we de functionaliteiten en verantwoordelijkheden van de diverse code fragmenten keurig gesplitst en zien we dat de code een stuk eenduidiger wordt. Programmeren is orde scheppen, en dat hebben we nu mooi gedaan.

<?opdrachten

Neem de Client en de Order-classes uit de trait opdracht en vervaardig een dependency injection zodat alle Orders bij de specifieke Client worden opgenomen.

<?object_pool Het Object Pool pattern wordt gebruikt om object-instanties voor hergebruik te bewaren. Meestal zijn deze niet direct data gerelateerd maar operationeel van aard (bijvoorbeeld configuratie, vertalingen oid.) Je kunt de objectpool als een soort "pre-loader" van klassen beschouwen waarbij je je objecten "parkeert" voor later gebruik. Hoewel zeer bruikbaar, is het praktischer in een niet-web omgeving!
Analyseer onderstaande code:

```
1    <?php
2
3    class ObjectPool {
4        private $instances = [];
5
6        public function get($key) {
7            return $this->instances[$key];
8        }
9
10        public function add($object, $key) {
11            $this->instances[$key] = $object;
12        }
13   }
14
15   class Mail {
16       public function sendMail() {
17           echo “Stuur email”;
18       }
19   }
20
21   class PDF {
22       public function createPDF() {
23           echo “Maak PDF”;
24       }
25   }
26
27   // Client code
28   $pool = new ObjectPool();
29   $mailObject = new Mail();
30   $pdfObject = new PDF();
31
32   $pool->add($mailObject, ‘mail’);
33   $pool->add($pdfObject, ‘pdf’);
34
35   $m = $pool->get(‘mail’);
36   $m->sendMail();
```

Middels een objectpool kunnen we een hele set aan objecten in één keer doorgeven aan volgende methodes of functies. Objectpools zijn handig om bijvoorbeeld een hele workflow in één keer uit te voeren.

<?singleton Volgens velen is een Singleton een “anti pattern” (oftewel niet gebruiken) maar het is in sommige gevallen ontzettend handig.
Je kunt slechts 1 instantie van een klasse maken, waardoor je een beperking op het gebruik kunt opleggen, bijvoorbeeld: je mag maar 1 database connectie openen.
Ook kun je de singleton als een soort van “super global” gebruiken. Op zich is het gebruik van globale variabelen natuurlijk “not done”, maar bijvoorbeeld bij het vervaardigen van een Android applicatie in Xamarin (C#) is het vrijwel onmogelijk om op een transparante manier informatie tussen schermen te delen. Dan biedt een “super global” variabele een prima alternatief. Onderdstaand voorbeeld is een klasse Database die maar eenmalig geïnstantieerd kan worden:

```
1    class Database {
2
3        private static $instance = null;
4        private $conn;
5
6        public static function connect()
7        {
8          if(!self::$instance) {
9            echo “New Connection!! -> “;
10           self::$instance = new self;
11         } else {
12           echo “No need to connect..\n”;
13         }
14
15         return self::$instance;
16       }
17
18       public function getConnection()
19       {
20         return $this->conn;
21       }
22
23       public function closeConnection() {
24          echo “Closing connection!!\n\n”;
25          $this->conn = null;
26          self::$instance = null;
27       }
28
29       /// ...
```

De private constructor zorgt ervoor dat we geen “ = new Databasse();” kunnen uitvoeren, terwijl de code op regel 10 juist wél een nieuwe instantie van het object aanmaakt - maar alleen maar als het object nog niet in memory bestaat.

```
29   /// ...
30
31       private function __construct()
32       {
33          $host = ‘127.0.0.1’;
34          $user = ‘root’;
35          $passwd = ‘root’;
36          $dbn = ‘php_master’;
37
38          $this->conn =  new PDO(“mysql:host=$host;
39                                 dbname=$dbn;
40                                 charset=utf8”,
41                                 $user, $passwd);
42          echo “ exec __construct!\n”;
43
44       }
45
46   }
```

Door de constructor private te maken, is het niet mogelijk om `$db = new Database();` uit te voeren. We kunnen een connectie maken met de database door de static functie: `$instance1 = Database::connect();` aan te roepen.

```
47   // $werkt_niet = new _database();
48
49   $instance1 = _database::connect();
50   $db = $instance1->getConnection();
51   $sql = “SELECT * FROM country WHERE Capital = 1”;
52   $stmt = $db->prepare($sql);
53   $stmt->execute();
54   $results = $stmt->fetchAll(PDO::FETCH_ASSOC);
55   var_dump($results);
56
57   // $instance1->closeConnection();
58
59   $instance2= _database::connect();
60   $db = $instance2->getConnection();
61   $sql = “SELECT * FROM country WHERE Capital = 5”;
62   $stmt = $db->prepare($sql);
63   $stmt->execute();
64   $results = $stmt->fetchAll(PDO::FETCH_ASSOC);
65   var_dump($results);
```

<?opdrachten

Neem de Database Singleton en pas deze aan zodat er ook maar één Object Pool aangemaakt kan worden!

<?super_global_singleton De “super global” is iets eenvoudiger qua opzet en gebruik. Theoretisch is het niet eens een echte Singleton, maar bij gebrek aan een betere term noem ik het maar even zo. Wees voorzichtig met het gebruik van welk type globale variabele dan ook. Zodra je ze gebruikt, moet je ook een methode verzinnen om deze data goed te managen. Als ik al een dusdanige constructie gebruik, is dit meestal met configuratie data en/of styling objecten o.i.d. en zelden tot nooit operationele data. Zodra je met JavaScript tools als React en Vue werkt, kom je een fenomeen tegen dat “Redux” genoemd wordt. In principe is dat vergelijkbaar met dit type Singleton.

```
<?php

   class StorageSingleton {

        public static $data = “”;

        public static function showData() {
            echo StorageSingleton::$data;
        }

        private function __construct() {}
   }


   class something {

     public function __construct() {
      StorageSingleton::showData();
     }

     public function changeData() {
      StorageSingleton::$data = “Nu anders...”;
     }
   }


StorageSingleton::$data = “Tijdelijke opslag!!”;
$s = new something();
$s->changeData();
StorageSingleton::showData();

$werktNiet = new StorageSingleton();
```

<?adapter Met een adapter wordt het mogelijk om ongelijke objecten om te zetten naar gelijk(w)aardige code. Je creëert zodoende een transparante laag zodat de aanroepende laag zich verder niet druk hoeft te maken over de implementatie van de "onderkant". Je kunt het zien als een soort van proxy die een extra interface biedt voor de methodes uit de basis klasse. Een voorbeeld verheldert ook hier weer het theoretische kader:

```
<?php

  interface uInterface {
    public function getFirstname();
    public function getLastname();
  }

  class user implements uInterface {

    public function getFirstname() {
        return("From");
    }

    public function getLastname() {
        return("User");
    }

  }

  class person implements uInterface {

     public function getFirstname() {
        return("From");
    }

    public function getLastname() {
        return("Person");
    }

  }

/// ...
```

```
32  /// ...
33
34    class userAdapter {
35
36      private $user;
37
38      public function __construct($user) {
39          $this->user = $user;
40      }
41
42      public function getUserFullname() {
43          return(
44             $this->user->getFirstname() . " " .
45             $this->user->getLastname());
46      }
47
48    }
49
50    $u = new user();
51    $p = new person();
52
53    $ua = new userAdapter($u);
54    $ub = new userAdapter($p);
55
56    echo $ua->getUserFullname() . "<br>";
57    echo $ub->getUserFullname() . "<br>";
```

Door de adapter zijn we nu in staat dezelfde methodes voor verschillende klassen te construeren. Dit is uiteraard alleen zinvol zodra het functioneel gelijksoortige/gelijk(w)aardige objecten betreft. Het heeft weinig toegevoegde waarde om het merk van een auto en de naam van een huisdier middels een adapter op hetzelfde niveau te trekken.

<?decorator Een Decorator maakt het mogelijk een object uit te breiden met nieuwe code. Op zich vind ik het een heel lelijke oplossing en zijn er tal van andere mogelijkheden om dit voor elkaar te krijgen. Maar... ook deze wordt regelmatig gebruikt, dus hij verdient zeker aandacht... Analyseer onderstaande code:

```
<?php

 interface userInterface {
     public function addFeature();
     public function getAccount();
 }


 class User implements userInterface {

    private $u = array();

    public function addFeature() {
        $this->u[“login”] = “user@server.com”;
    }

    public function getAccount() {
        // ...
        return( $this->u );
    }
 }

abstract class userFeature implements userInterface
{
    protected $user;

    abstract function addFeature();
    abstract function getAccount();

    function __construct(userInterface $user)
    {
      $this->user = $user;
    }

}

/// ...
```

Vervolgens kunnen er nieuwe “Feature”-klassen worden toegevoegd waarmee we de User-klasse gaan “docereren”, door het object door te geven aan de diverse “Feature”-klassen. Zodoende wordt het uiteindelijk object steeds uitgebreider.

```
37  /// ...
38
39  class userApp extends userFeature {
40
41      private $u;
42
43      public function addFeature() {
44          $this->u = $this->user->getAccount();
45          $this->u[“cursus”] = “PHP Masterclass”;
46      }
47
48      public function getAccount() {
49          return($this->u);
50      }
51   }
52
53   class userLocation extends userFeature {
54      private $u;
55
56      public function addFeature() {
57          $this->u = $this->user->getAccount();
58          $this->u[“location”] = “Utrecht”;
59      }
60
61      public function getAccount() {
62          return($this->u);
63      }
64   }
65
67
68   $u = new User();
69   $u->addFeature();
70
71   echo “<h1>Decorator</h1><pre>User<br>”;
72   var_dump( $u->getAccount() );
73
74   $ua = new userApp($u);
75   $ua->addFeature();
76
77   echo “<pre>User Cursus<br>”;
78   var_dump( $ua->getAccount() );
79
80   $ul = new userLocation($ua);
81   $ul->addFeature();
82
83
84   echo “<pre>User Cursus Locatie<br>”;
85   var_dump( $ul->getAccount() );
```

<?factory De Factory is mogelijk na MVC het meest gebruikte pattern - en ook een van de meest bruikbare. Het is relatief eenvoudig te begrijpen en te implementeren en kent eigenlijk alleen maar voordelen. Het is eigenlijk een soort van proxy voor je klassen. De Factory neemt het instantiëren van klassen uit handen van de basis code en retourneert het kant en klare object. Het grote voordeel van het gebruik van Factories is dat je een Single Point of Entry krijgt voor je applicatie structuur. Zodra er ergens iets wijzigt in de applicatie klassen, hoef je in theorie alleen maar je factory te wijzigen en niet alle aanroepen links en rechts door je systeem.

```
1    <?php
2
3
4    class User {
5
6        private $username;
7        private $password;
8
9        public function show() {
10           echo $this->username .
11                " with password: " .
12                $this->password ."<br>";
13       }
14
15       public function __construct($username,
16                                   $password) {
17           $this->username = $username;
18           $this->password = sha1($password);
19       }
20   }
21
22   class Course {
23
24       private $name;
25
26       public function show() {
27           echo $this->name . "<br>";
28       }
29
30       public function __construct($name) {
31           $this->name = $name;
32       }
33   }
34
35   /// ...
36
37
```

```
35  /// ...
36
37  class Factory {
38
39      public static function createUser($email) {
40         $password = “geheim@” . date(“dmYHi”);
41         return( new User($email, $password) );
42      }
43
44      public static function createCourse($name) {
45          return( new Course($name) );
46      }
47
48  }
49
50  $user = Factory::createUser(“hans@code.com”);
51  $course = Factory::createCourse(“PHP Master Class”);
52
53  $user->show();
54  $course->show();
```

Door het creëren van nieuwe instanties van objecten naar de factory te delegeren, kunnen we in die factory in heleboel zaken regelen alvorens het object te creëren. Moeten we deze klassen op diverse plekken aanroepen, dan kunnen we dat via de static methodes van een en dezelfde factory doen. Hierdoor blijft het eerder genoemde “**Single Responsibility**” principe bewaard en sterker nog: onze code wordt steeds “droger” (**DRY**: Don’t Repeat Yourself).

<?observer Het Observer pattern is een set van "losse" transacties die op basis van een specifieke status uitgevoerd moeten worden. Het lijkt nog het meest op een soort van Worklow systeem. De Observer stuurt een "notificatie" naar het geabonneerde object en op basis van die notificatie wordt een specifieke methode uitgevoerd. Ook hier is een praktisch voorbeeld weer duidelijker dan de theorie:

```
1    <?php
2
3    abstract class AbstractObserver {
4        abstract public function update(
5                AbstractTransaction $transaction_in);
6    }
7
8    abstract class AbstractTransaction {
9        abstract public function attach(
10                   AbstractObserver $observer_in);
11       abstract public function detach(
12                   AbstractObserver $observer_in);
13       abstract protected function notify();
14   }
15
16   /// ...
```

Allereerst tuigen we een abstract class definitie op waaraan de Observer klassen moeten voldoen. Deze nemen we op in een "Transactie" object.
Wat er moet gebeuren is dat er bij een update van een specifieke status zowel het User-, als het App-, als het Location-object bijgewerkt moeten worden (zonder dat deze zich van elkaar bewust zijn). Deze objecten zijn als het ware "geabonneerd" op de transactie.

```
16   /// ...
17
18   class User extends AbstractObserver {
19     public function update(
20              AbstractTransaction $transaction) {
21          echo "User: " . $transaction->getData();
22      }
23   }
24
25   class App extends AbstractObserver {
26     public function update(
27              AbstractTransaction $transaction) {
28          echo "App: " . $transaction->getData();
29     }
30   }
31   ///...
```

```
16   /// ...
17
18   class Location extends AbstractObserver {
19     public function update(
20              AbstractTransaction $transaction) {
21          echo “Location: “ . $transaction->getData();
22       }
23   }
24
25   ///...
```

Dit zijn de “Data-klassen” die reageren moeten op een specifieke status. Uiteraard zal de code “in het echt” natuurlijk veel uitgebreider zijn, maar zo kunnen we laten zien wat wanneer uitgevoerd wordt.

Vervolgens gaan we de “Abonnementen”-service coderen. We zorgen ervoor dat we abonnementen kunnen toevoegen (`attach`), maar ook weer kunnen opzeggen (`detach`):

```
25   /// ...
26
27   class Transaction extends AbstractTransaction {
28
29       private $observers = array();
30       private $data = “”;
31
32       protected function notify() {
33         foreach($this->observers as $obs) {
34           $obs->update($this);
35         }
36       }
37
38       public function attach(
39              AbstractObserver $observer_in) {
40         $this->observers[] = $observer_in;
41       }
42
43       public function detach(
44              AbstractObserver $observer_in) {
45         foreach($this->observers as $key => $val) {
46           if ($val == $observer_in) {
47             unset($this->observers[$key]);
48           }
49         }
50       }
51
52
53   /// ...
54
```

Zodra de status bijgewerkt moet worden (de `updateData`-functie) worden alle abonnementen middels de `notify()` functie ingelicht en kunnen deze hun update routine uitvoeren.

```
25   /// ...
26
27
28       public function updateData($data) {
29         $this->data = $data;
30         $this->notify();
31       }
32
33       public function getData() {
34         return $this->data;
35       }
36   }
37
38   $transaction = new Transaction();
39   $user = new User();
40   $app  = new App();
41   $location = new Location();
42
43   $transaction->attach($user);
44   $transaction->attach($app);
45   $transaction->attach($location);
46
47   $transaction->updateData(“Actie #1”);
48   echo “<hr>”;
49
50   $transaction->updateData(“Actie #2”);
51   echo “<hr>”;
52
53   $transaction->detach($user);
54   $transaction->updateData(“Actie #3”);
```

Als je goed kijkt zijn er natuurlijk enorm veel overeenkomsten tussen de Object-pool en de Observer patterns - hoewel ze in principe andere situaties oplossen, zou je ze ook prima kunnen samenvoegen.

<?opdrachten

Schrijf een Observer-systeem dat ervoor zorgt dat zodra er een bestelling binnenkomt in een webshop dat er automatisch een gebruikers-account wordt gecontrolleerd en/of aangemaakt, een factuur gemaakt wordt, de vooraad van het artikel gecontrolleerd wordt en een transport bij een externe firma besteld wordt. Je hoeft geen feitelijke database handelingen te schrijven: de nadruk ligt op de implementatie van het Observer pattern.

<?strategy Een strategy is eigenlijk een combinatie van een “gedelegeerde IF” en een “Factory” pattern. Bijvoorbeeld: “ik ga op vakantie en ga met: **a)** de bus, **b)** de auto, **c)** de trein en **d)** het vliegtuig”. We laten het aan de strategy over om te bepalen welk object we nodig hebben en laten dit meteen door de strategy maken:

```
1   <?php
2
3   interface StrategyInterface {
4       public function showType();
5   }
6
7   class Modem implements StrategyInterface {
8       public function showType() {
9           return(“Uitleveren nieuw Modem”);
10      }
11  }
12
13  class SetTopBox implements StrategyInterface {
14      public function showType() {
15          return(“Uitleveren nieuwe SettopBox”);
16      }
17  }
18
19  class Phone implements StrategyInterface {
20      public function showType() {
21          return(“Uitleveren nieuwe Telefoon”);
22      }
23  }
24
25  /// ...
```

Met ons systeempje kunnen we 3 types hardware uitleveren aan de klant: een modem, een settopbox en een telefoon. Middels de interface zorgen we ervoor dat elk van die klassen een functie `showType` heeft waardoor we kunnen zien wat er uitgeleverd wordt. Hoewel niet direct noodzakelijk, kunnen we door middel van een `Request`-klasse de zaak nog wat meer stroomlijnen:

```
25  /// ...
26
27  class Request {
28      public $hardware;
29
30      public function __construct($hardware) {
31          $this->hardware = $hardware;
32      }
33  }
34
35  /// ...
```

De uiteindelijke Strategy klasse heeft het Request-object als argument om zo het 'aftrappen' van de Strategy eenduidig te houden:

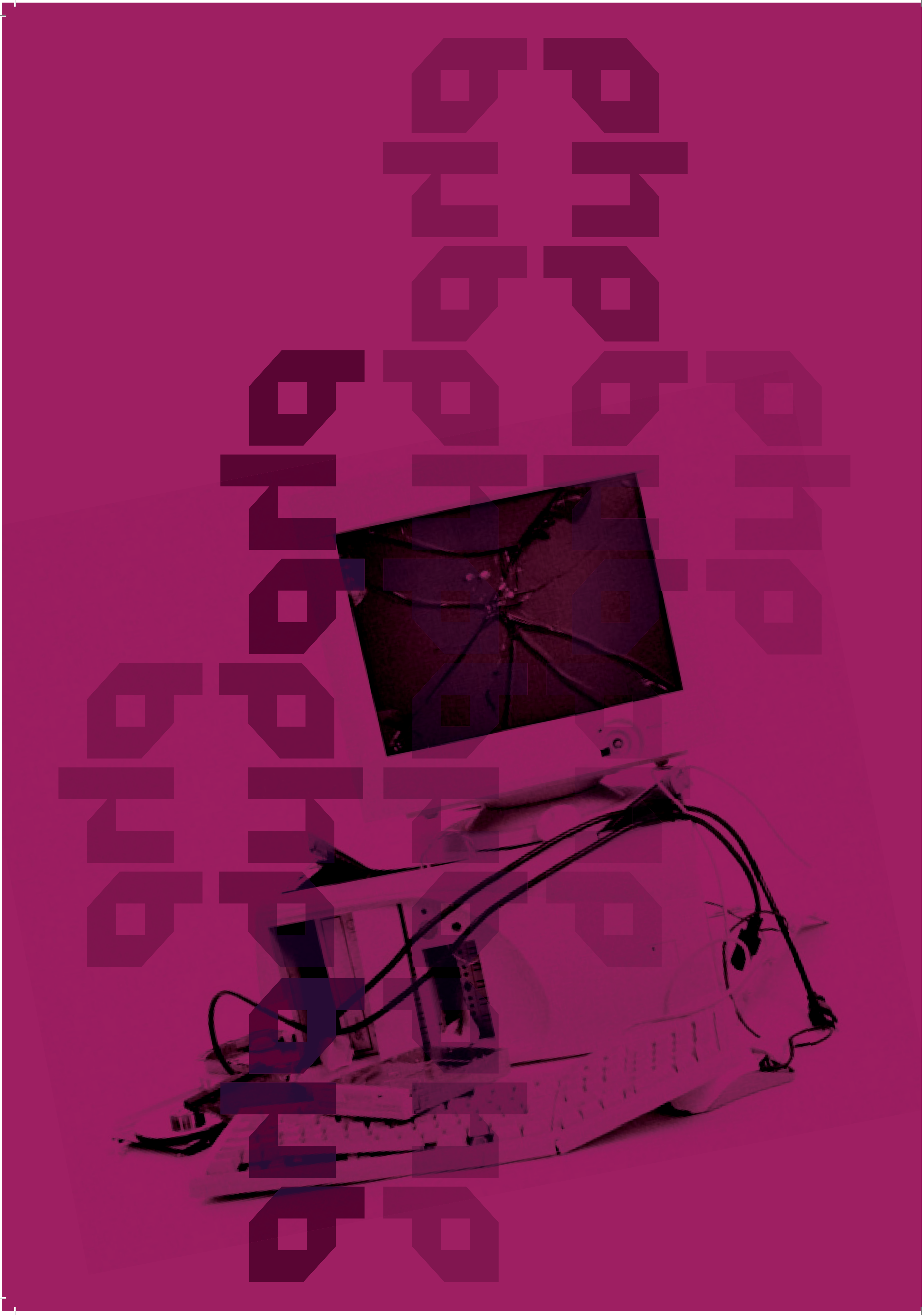
```
35  /// ...
36
37  class StrategyContext {
38      private $strategy = NULL;
39
40      public function __construct($request) {
41          switch ($request->hardware) {
42              case "M": {
43                  $this->strategy = new Modem();
44                  break;
45              }
46
47              case "S": {
48                  $this->strategy = new SetTopBox();
49                  break;
50              }
51
52              case "P": {
53                  $this->strategy = new Phone();
54                  break;
55              }
56          }
57
58      }
59
60      public function showType() {
61          return $this->strategy->showType();
62      }
63  }
64
65  /// ...
```

Je ziet hier de "gedelegeerde IF" - in de vorm van een 'switch' - die bepaald welk object er aangemaakt moet worden (de Factory implementatie). We kunnen nu de Strategy aftrappen:

```
65  /// ...
66
67  $modem = new Request("M");
68  $stb = new Request("S");
69  $phone = new Request("P");
70
71  $strategy = new StrategyContext($modem);
72  echo $strategy->showType() . "<br>";
73  $strategy = new StrategyContext($stb);
74  echo $strategy->showType() . "<br>";
75  $strategy = new StrategyContext($phone);
76  echo $strategy->showType() . "<br>";
```

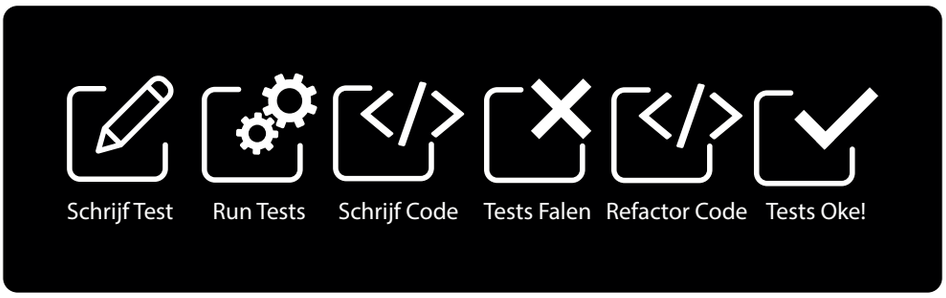
<?lazy_loading Lazy Loading is een eenvoudig pattern dat ervoor kan zorgen dat het laden van een object alleen uitgevoerd wordt als het echt noodzakelijk is - omdat een bepaald proces lang duurt bijvoorbeeld, zorg je ervoor dat het maar 1x uitgevoerd wordt:

```
<?php

class LazyLoader {

    private $onlyOnce = false;

    public function init()
    {
        if (!$this->onlyOnce) {
            $this->onlyOnce = $this->takesLong();
        }
        return $this->onlyOnce;
    }

    private function takesLong()
    {
        sleep(5);
        echo “Deze functie duurt lang...<br>”;
        return(true);
    }
}

$l = new LazyLoader();
echo “First call!!<br>”;
$first = $l->init();

echo “Second call!!<br>”;
$second = $l->init();
echo $second;
```



<?test_driven_development

Test-Driven Development (TDD) is een ontwikkelmethode voor software waarbij eerst tests worden geschreven en daarna pas de code.



De ontwikkelcyclus is gebaseerd op het boek "*Test-driven development by example*" van Kent Beck (Pearson Education, november 2002). Deze cyclus kan zo vaak herhaald worden als nodig is om de code volledig werkend te krijgen volgens de requirements:

- **Test maken:** In Test-Driven Development maakt de programmeur eerst een test, gebaseerd op een requirement, en schrijft pas dan de code.
- **Alle tests uitvoeren en kijken of de nieuwe test faalt**: Deze test moet initieel in principe falen, aangezien het stuk code waarop deze test van toepassing is nog niet bestaat. Het laten falen van de nieuwe test is belangrijk om na te gaan of de test effectief is (geen fouten bevat en daardoor bijvoorbeeld altijd slaagt).
- **Code schrijven:** In deze stap wordt de daadwerkelijke code geschreven om de zojuist gemaakte test te laten slagen. Deze code mag alleen functionaliteit bevatten die nodig is om de test te laten slagen. De code hoeft niet perfect geschreven te zijn, zolang deze maar werkt. In een latere stap zal de code nog herschreven worden volgens de standaard.
- **Tests uitvoeren en kijken of deze slagen:** In deze stap draaien alle tests nogmaals en wordt gekeken of alle tests slagen. Is dit het geval dan is dit een goed uitgangspunt om aan de volgende stap te beginnen.
- **Code herschrijven:** In deze stap wordt de code opgeschoond en volgens de standaard herschreven. Ook zal eventuele dubbele code, nodig om de onderlinge afhankelijkheid van de componenten te minimaliseren, verwijderd worden.

<?voordelen

Er wordt bij Test-Driven Development gekeken vanuit het perspectief van de gebruiker. De testcases waarvoor de code wordt geschreven zullen namelijk gebaseerd zijn op de requirements, die vanuit het oogpunt van de gebruiker zijn opgesteld. Problemen met de bruikbaarheid van de interface kunnen zo eerder worden gevonden en gecorrigeerd in een stadium waarin het gemakkelijkst is.
Er wordt specifiek gelet op de requirements, aangezien de tests daarop gebaseerd zijn.

Extra, wellicht overbodige, functionaliteiten zullen hierdoor niet geïmplementeerd worden, waardoor de hoeveelheid programmacode binnen de perken blijft.
Doordat alle code van het begin af aan wordt getest, wekt dit meer vertrouwen bij het ontwikkelteam en bij de klant.
Ondanks de extra code die nodig is voor het schrijven van de tests, zal de ontwikkelingstijd toch korter worden, aangezien fouten al in een zeer vroeg stadium gevonden zullen worden.
Aangezien alle onderdelen van de code los van elkaar getest worden is de onderlinge afhankelijkheid kleiner en daarom minder complex.

<?nadelen De programmeur schrijft bij deze methode zowel de tests als de code voor de applicatie. Dit heeft tot gevolg dat wanneer de programmeur iets over het hoofd ziet, dit in zowel de test als in de code over het hoofd gezien zal worden.
Wanneer er een groot aantal tests succesvol is, kan dit de indruk geven dat de applicatie volledig getest is. Een valkuil is dat een integratie- en systeemtest niet of nauwelijks uitgevoerd worden.

<?toepassing Deze methode valt onder extreme programming (XP), aangezien het gebruikmaakt van korte ontwikkelcycli met het continu testen van de software. Voor het ontwikkelen/upgraden van toolkits, zoals bijvoorbeeld de Dojo Toolkit, is Test-Driven Development een toepasbare manier van ontwikkeling, aangezien integratie- en systeemtests hierbij nog niet ter zake doen.

<?soorten_test Er zijn een aantal verschillende tests die uitgevoerd kunnen worden door ervaren test-engineers:

1. **De unit-test,** dit is de kleinst mogelijke te testen eenheid in een applicatie (bijvoorbeeld een losse functie);
2. **Integratie test:** werkt alles “end-to-end” in technisch opzicht;
3. **Acceptatie test:** werkt alles “end-to-end” in functioneel opzicht;
4. **UI-test:** werkt de user-interface naar behoren.

We richten ons hier verder op unit-testing in PHP. Voor alle andere tests kunnen test methodieken als TMap geïmplementeerd worden.

<?test_automation De unit test kunnen deel uit maken van je development pipeline - eventueel met behulp van CI/CD (Continuous Integration & Continuous Delivery) tools als Jenkins. Unit testing van je PHP applicatie kan volledig geautomatiseerd uitgevoerd worden.

<?richtlijnen_unit_testing_in_php

- **LET OP!!!** Ga niet unit-testen omwille van het unit-testen!!
- Het is een andere manier van denken over je code!
- Schrijf je test en definieer het verwachtingspatroon van je output *VÓÓR* dat je je echte code schrijft!
- **TDD** kan ook zorgen voor een verbeterd kwaliteitsgevoel en verbeterde perceptie van je project bij de teamleden en de (externe) stakeholders
- Eindgebruikers moeten bij het opstellen van de tests betrokken worden (requirements) Basis principe: **KISS** (Keep it simple, stupid!)
- Hou je test eenvoudig en voorspelbaar
- Ondanks dat het misschien niet altijd even handig is, schrijf je tests toch zo dat *ALLE* transacties richting database of (interne) webservices toch fysiek uitgevoerd worden (gebruik zo min mogelijk mock functies)
- UI Testing is niet echt mogelijk
- Formulier input zul je moeten "faken"
- Begin met de implementatie van PHPUnit en later eventueel een uitgebreider en/of professioneler test framework implementeren.
- Codeception (**https://codeception.com/**) is bijvoorbeeld een prima testing framework dat gebaseerd is op PHPUnit.
- Er bestaan tal van "losse" ui-testing tools, de meeste betaald. Zoek naar een geschikte.

<?php_unit PHPUnit is een zeer uitgebreid "open source" test framework waarin je een groot aantal test scenario's kunt opnemen. Je kunt met eenvoudige "assertions" beginnen en langzaam maar zeker uitgebreidere test schrijven. Het framework gaat er wel min of meer vanuit dat je aan een nieuw project begint. Het is namelijk erg lastig om een bestaand project "test klaar" te maken. Over het algemeen kun je ervan uitgaan dat de basis-assertions ongeveer 90% van je tests zullen zijn.

<?installatie PHPUnit kun je globaal of door middel van composer per project installeren (**https://getcomposer.org**).

```
// OSX & Linux Globaal
$> wget -O phpunit https://phar.phpunit.de/phpunit-8.phar
$> chmod +x phpunit
$> mv phpunit /usr/local/bin

// Package (per project)
$ (in je project-root)> composer require phpunit/phpunit
```

<?phpunit_starten Afhankelijk van je installatie type en OS kan phpunit op de volgende manier gestart worden:

```
// MacOSX & Linux Globaal
$> phpuit --version

// Package (per project) Windows & Mac/Linux
$ (in je project-root)> php ./vendor/bin/phpunit --version
```

<?vooraf Voordat we iets kunnen testen zullen we eerst een project moeten aanmaken. Maak een voor je webserver toegankelijke diretory aan (in je document root bijvoorbeeld) en installeer phpunit met behulp van composer in je project.
Maak vervolgens de subdirectories `lib`, `log` en `tests` aan in je project. Je directory structuur ziet er nu zo uit:

```
-- project
 |
 +- lib
 +- log
 +- test
 +- vendor
```

Maak met je code editor 2 bestanden in de lib directory: `App.php` en `User.php` en neem de volgende code op in deze files:

```
1     <?php
2     /// App.php
3
4     class App {
5        public function __construct() {
6
7         }
8
9         public function getCounter() {
10            return(10);
11        }
12
13    }
```

```php
<?php
/// User.php

class User {

    private $email;
    private $password;

    public function __construct(string $email,
                                string $password) {

        $this->email = $email;
        $this->password = $password;

    }

}
```

Maak nu in de root van je project een index.php aan, en kijk of de User-klasse werkt:

```php
<?php
/// index.php

require_once('lib/User.php');

$u = new User("code@dev-master.ninja", "password");

```

Nu we ons (minimale) projectje hebben opgetuigd, kunnen we beginnen met het schrijven van onze eerste unit-test. Zoals eerder gezegd zullen de zogenaamde "basis assertions" het grootste gedeelte van je unit-test gaan vormen. We zullen in onze eerste test dan ook gaan kijken of we eigenlijk wel een object van de User-klasse kunnen aanmaken. Maak in de directory `tests` een subdirectory `user` aan. We gaan onze test functioneel groeperen. In deze map maak je nu een PHP file `UserTest.php` aan. In deze file zullen onze test opgenomen worden, maar daar moeten we even iets voor doen:

```php
<?php
/// User.php

require_once('lib/User.php');
use PHPUnit\Framework\TestCase;

/// ...
```

Composer heeft ervoor gezorgd dat we middels autoloading de juiste PHPUnit namespaces aan kunnen spreken. Voor het gemak "requiren" we nog gewoon onze User klasse.

Neem nu onderstaande code over in `UserTest.php`:

```
1    <?php
2
3    require_once('lib/User.php');
4    use PHPUnit\Framework\TestCase;
5
6    class UserTest extends TestCase
7    {
8        private $u;
9
10       public function setUp():void {
11           $this->u = new User("code@dev-master.ninja",
12                               "password");
13       }
14
15       public function testCanCreateInstanceOfUser() {
16           $this->assertIsObject($this->u);
17       }
18
19   }
```

De `TestCase` class kan een methode `setUp():void` overerven waarmee we het voorbereidende werk voor de test kunnen opzetten.
LET OP! Ga hier geen uitgebreide programma code opnemen. Een test moet precies dat doen wat de naam zegt: **TESTEN** en geen volledige workflow automatiseren!!

Vervolgens zie je de methode `testCanCreateInstanceOfUser()` - waarom die naam? Nou, **PHPUnit** gebruikt straks die naam in de output en als deze met "test" begint en verder netjes van "Camel Casing" gebruik maakt, komt in de rapportage een keurige zin te voorschijn:

```
User
 ✓ Can create instance of user  3 ms
```

Als je een heleboel tests gaat uitvoeren, is het natuurlijk wel fijn dat de rapportage ook goed leesbaar is.

<?tests_uitvoeren Je kunt test op een aantal manieren uitvoeren. Of "gewoon" de testfile aanroepen, of door middel van een configuratie-file (XML).
De meest recht-toe-recht-aan manier is om de unit-test direct aan te roepen. Dat doe je als volgt (vanuit de root-directory van je project):

```
// Globaal
$> phpunit ./tests/user/UserTest.php

// Package (per project)
$> php ./vendor/bin/phpunit ./tests/user/UserTest.php
```

Als je geen typefouten hebt gemaakt, zal de output ongeveer zo uitzien:

```
$> php ./vendor/bin/phpunit ./tests/user/UserTest.php
PHPUnit 8.3.4 by Sebastian Bergmann and contributors.

.                   1 / 1 (100%)

Time: 34 ms, Memory: 4.00 MB

OK (1 test, 1 assertion)
```

Deze kunnen we nog enigszins verfraaien door de optie `--testdox` mee te geven:

```
$> php ./vendor/bin/phpunit ./tests/user/UserTest.php --testdox
PHPUnit 8.3.4 by Sebastian Bergmann and contributors.

User
 ✓ Can create instance of user

Time: 42 ms, Memory: 4.00 MB

OK (1 test, 1 assertion)
```

Als het project wat omvangrijker wordt is het natuurlijk niet heel erg praktisch om afzonderlijke test op de CLI te specificeren. Daartoe kent PHUnit een configuratie file waarin we kunnen configureren hoe de tests uitgevoerd moeten worden. Als je in de project root een berstand `phpunit.xml` aanmaakt, dan wordt dit de default configuratie zodra je phpunit op de CLI uitvoert. Middels de -c flag kan ook een ander configuratie bestand opgegeven worden:

```
// Gaat uit van een bestand phpunit.xml in de root-dir
$> php ./vendor/bin/phpunit

// Specifiek configuratie bestand
$> php ./vendor/bin/phpunit -c [configfile.xml]
```

Een typisch configuratie bestand ziet er als volgt uit:

```
<?xml version="1.0" encoding="UTF-8"?>
<phpunit
     colors="true"
     convertErrorsToExceptions="true"
     convertNoticesToExceptions="true"
     convertWarningsToExceptions="true"
     forceCoversAnnotation="false"
     stopOnError="false"
     stopOnFailure="false"
     stopOnIncomplete="false"
     stopOnSkipped="false"
     verbose="true"
    >

    <testsuites>
        <testsuite name="UserTests">
            <directory>./tests/user</directory>
        </testsuite>
    </testsuites>

</phpunit>
```

Alle instellingen en opties zijn te vinden op via **https://phpunit.de**

<?opdrachten

Maak de test klasse voor de klasse in App.php en voeg de "assertIsObject" test toe.

Run op basis van bovenstaande XML configuratie je tests.

<?assertions Test driven development gaat ervan uit dat je eerst je test schrijft, en vervolgens pas de bijbehorende code. Een functionele vraag zou bijvoorbeeld kunnen zijn: *"Haal het email adres van de gespecificeerde gebruiker op".*

Een test zou dan bijvoorbeeld beschreven kunnen worden als: *"Komt het email adres code@dev-master.ninja overeen met het aangemaakte mail adres?"*

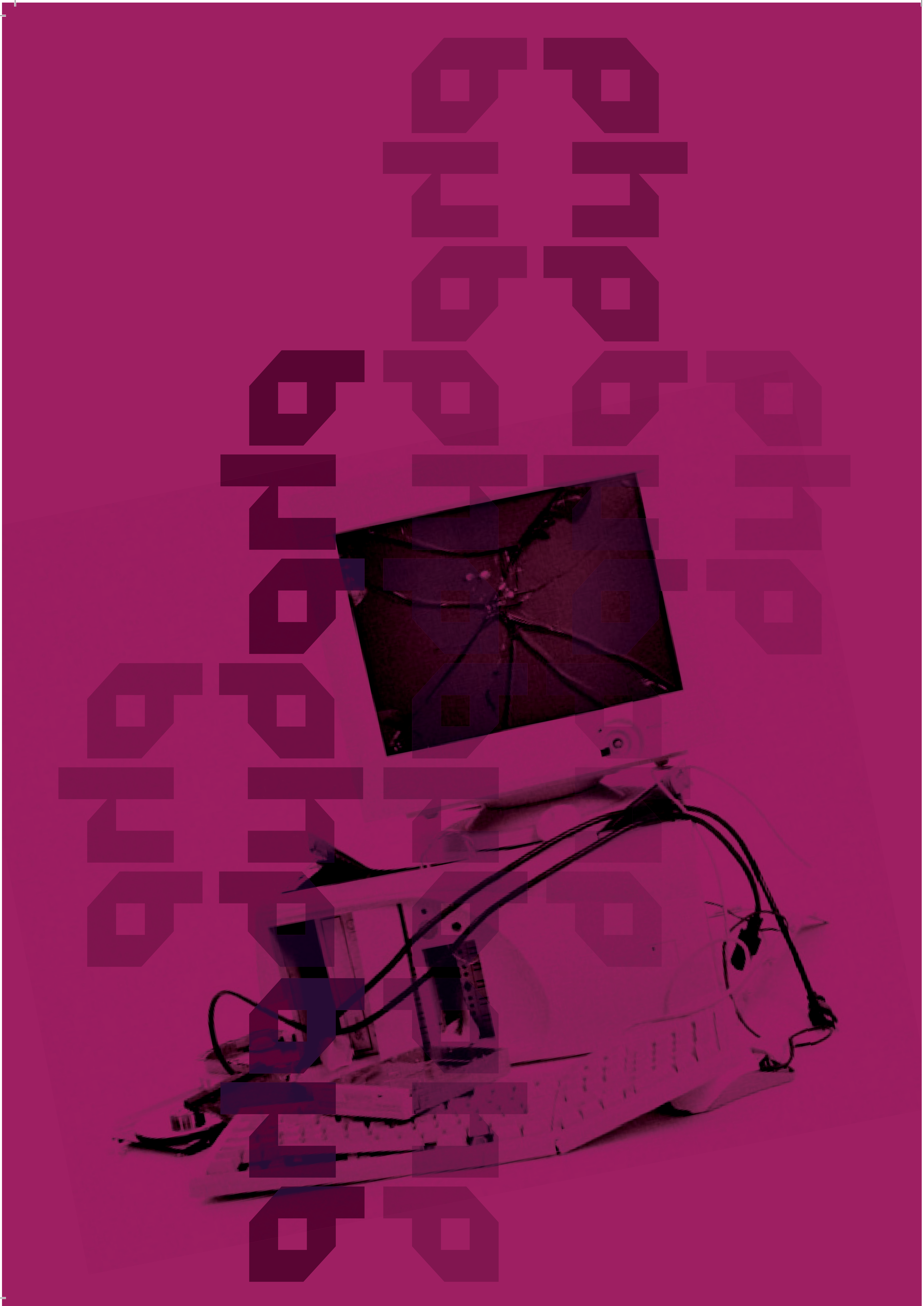
De bijbehorende unit-test wordt vervolgens:

```
1     <?php
2
3     require_once('lib/User.php');
4     use PHPUnit\Framework\TestCase;
5
6     class UserTest extends TestCase
7     {
8         private $u;
9
10        public function setUp():void {
11            $this->u = new User("code@dev-master.ninja",
12                                "password");
13        }
14
15        public function testCanCreateInstanceOfUser() {
16            $this->assertIsObject($this->u);
17        }
18
19        public function testUserEmailAddressEquals() {
20            $this->assertEquals("code@dev-master.ninja",
21                                $this->u->getEmail());
22        }
23
24    }
25
```

Als we nu de tests uitvoeren, falen deze natuurlijk direct: we hebben nog geen methode `getEmail()` in de user klasse in gebouwd. Voeg deze nu toe aan de user-klasse!

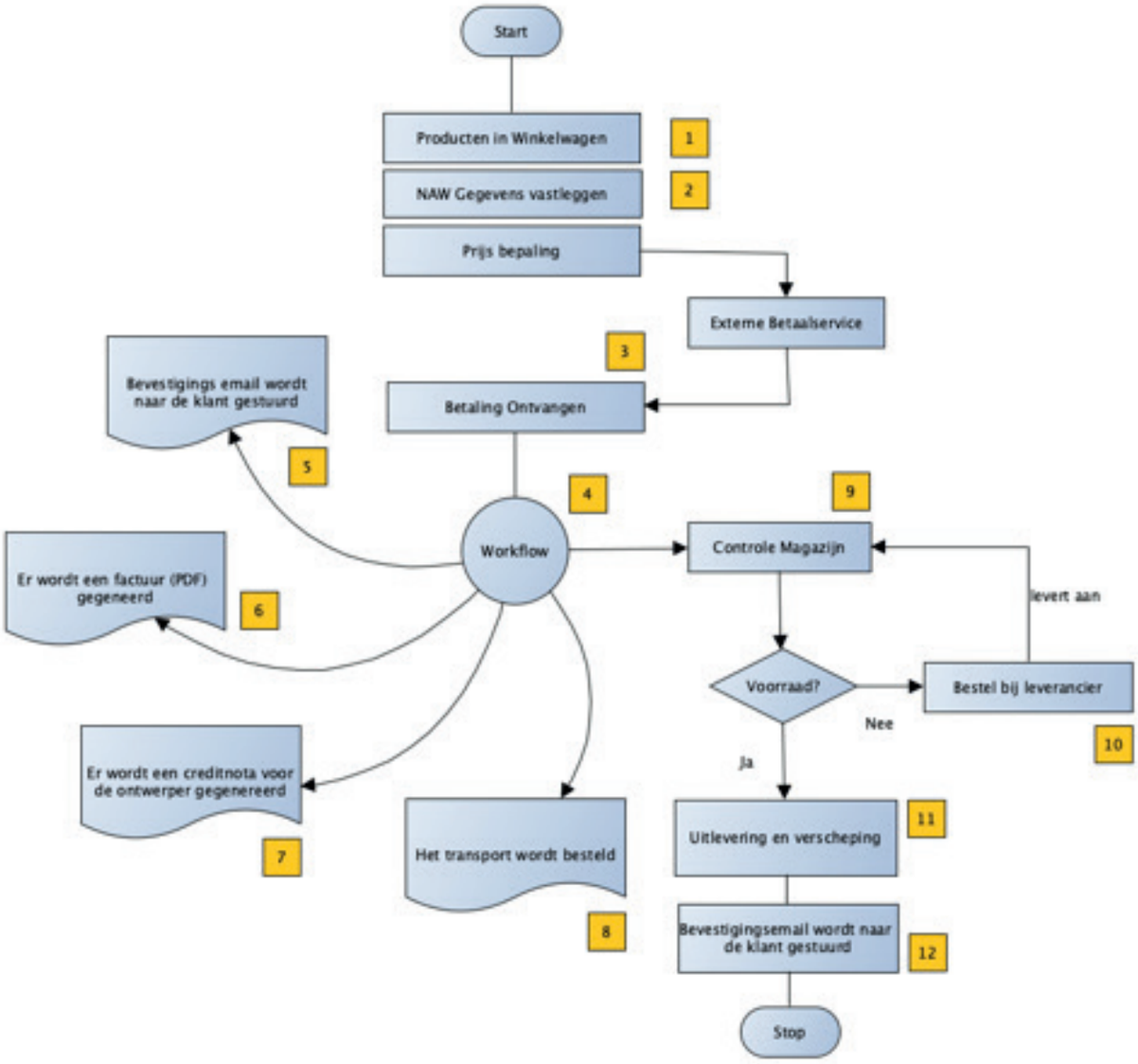
Deze manier van "test first code later" is precies de werkwijze die je bij TDD (Test Driven Development) moet hanteren. Bepaal je test aan de hand van de gebruikers specificaties en schrijf de code pas als de specs (en de tests) "rond" zijn.

Een uitgebreide lijst van alle assertions vind je op **https://phpunit.de** op de workshop site **https://dev-master.ninja/php-master** vind je een uitgewerkte user-klasse en een userTest klasse met een aantal uitgebreide assertions. Aangezien deze assertions per versie wijzigen ongeveer, is het niet zinvol om deze in deze reader op te nemen.



<?case_php_masterclass De webshop “Tremendous Tees” verkoopt t-shirts met opdruk. Deze t-shirts worden gemaakt door grafisch ontwerpers die als ZZP-ers verbonden zijn aan de webshop. De webshop heeft zelf een magazijn waarin een kleine voorraad van de populairste shirts ligt opgeslagen.

De webshop heeft een betalingsmodule gekoppeld en zodra een betaling ontvangen is, krijgt de shop via een zogenaamde webhook verificatie van de betaling en kan tot levering worden overgegaan. Onderstaande flow-chart toont hoe het proces in hoofdlijnen verloopt.



<?specificaties

- De NAW gegevens van de klant worden vastgelegd. Gegevens: voornaam, achternaam, tussenvoegsel, straat, huisnummer, verdieping, woonplaats, postcode, telefoonnummer, email adres.
- De bestelling wordt vastgelegd: een of meerdere gegevensverzamelingen van (product-code, aantal, maat)
- De prijs wordt bepaald en een externe betaalservice wordt aangeroepen. Deze service geeft een betalingsstatus terug aan de webshop. In dit geval is de status altijd "BETAALD" en kan het proces verder afgehandeld worden.
- Zodra de betaling ontvangen is wordt de workflow opgestart.
- Er wordt een bevestigingsmail naar de klant gestuurd.
- Er wordt een factuur gegenereerd en vervolgens wordt deze naar de klant gemaild.
- Er wordt een credit nota voor de ontwerper gegenereerd en een betalingsrecord aan zijn account toegevoegd
- Het transport wordt bij een onafhankelijke transporteur besteld middels een webservice.
- Per product dat de klant besteld heeft, wordt in het magazijn gecontroleerd of het aanwezig is. Zo ja, dan wordt het in de levering opgenomen, zo nee, dan moet het product besteld worden bij de ontwerper.
- De ontwerper krijgt via een bericht in zijn account een verzoek tot levering
- Voordat de producten naar de klant verscheept kunnen worden zal gecontroleerd moeten worden of de transporteur de klus aangenomen heeft en of de leverancier al dan niet het product geleverd heeft.
- Pas als aan alle voorwaarden voldaan is, krijgt de klant een email met verzendbevestiging en wordt de bestelling fysiek verscheept.

<?opdrachten

Herleid uit bovenstaande beschrijving de informatie behoefte en normaliseer deze zo ver als mogelijk. Bouw dit systeem vanaf het moment dat de workflow gestart wordt (stap 4). Sla de informatie op in een MySQL database. Het versturen van de emails, het aanmaken van de PDF's en het aanspreken van eventuele webhooks en/of webservices mag je met een zogenaamde stub oplossen - deze functionaliteit hoef je dus niet te maken, eventuele output of "service calls" kunnen in een ASCII of HTML-bestand opgeslagen worden.

Het systeem hoeft ook geen fraaie UI te hebben, de opdracht richt zich primair op het inzetten van een aantal van de besproken design patterns.
De docent speelt de rol van de niet-technische opdrachtgever.

<?beoordelingscriteria

- Uitgewerkt gegevensmodel (ERD) met de juiste entiteiten en PK/FK relaties
- Implementatie van minimaal 4 van de besproken Design Patterns, met een beetje creativiteit kun je alle besproken patterns implementeren. Er zijn extra punten te verdienen voor elk correct gebruikt pattern.
- Middels Unit tests kan het hele proces gevolgd worden.

Elke opdracht wordt beoordeeld op een schaal van 1 t/m 5:
1: onvoldoende, 2: matig, 3: voldoende, 4: goed, 5: uitstekend