IPC 2019: PHPStan

@OndrejMirtes

hat mit PHPStan angefangen
mussten Java machen, hat PHP schon gemacht
hat in PHP Dinge von Java vermisst

PHPStan geht auch für PHP 5.3 in Grenzen

Dynamic Analysis: inspiziert Program während es läuft

einfachstes Analysator: `php -l` :)

PHP_CodeSniffer: checkt nur die aktuelle Datei
kann ungenutzte Variablen und private Props finden, aber keine pub./prot. Props
arbeitet mit einer flachen Liste an Token

```
$t = token_get_all('<?php class A { private $f; }')
array_map(function($t) { echo is_array($t) ? token_name($t[0]).' '.$t[1]."\n" : $t; }, $t
```

PHP_CodeSniffer kann auch CSS und JS formatieren

Project Scope: phpstan/phpstan od. phpstan/phpstan-shim
empfielt es so zu installieren, weil:

- damit man die gleiche Version wie der Kollege hat
- hat selbst wieder selbst Abhängigkeiten die passen müssen..
  --> evtl das Problem in PfmScraper?

PHPStan CLI braucht die gleichen Extensions wie der auszuführende Code

Baseline Feature wird kommen mit 0.11.20 \o/ - aufrufen mit `--error-format baselineNeon` -> speichert in `phpstan-baseline.neon` - includen in `phpstan.neon` -> bekommt man nur noch neue Fehler

- generiert `ignoreErrors` Einträge :)
- gut bei Upgrades von PHPStan wenn damit ein Haufen neue Fehler gemeldet werden, und man keine Zeit hat die alle zu fixen
- aber nicht 12.000 Fehler unterdrücken, sondern erstmal auf ein paar Hundert herunterarbeiten und dann baselinen
- man könnte Level 1 baselinen und mit Level 2 weitermachen
- man kann auch eine eigene neue Regel ausprobieren und alle anderen Fehler unterdrücken

```
 vendor/bin/phpstan analyse --memory-limit=256M -c phpstan.neon -l 3 ./src --error-
format baselineNeon > phpstan-baseline.neon
 // in phpstan.neon eintragen: "includes: phpstan-baseline.neon"
 vendor/bin/phpstan analyse --memory-limit=256M -c phpstan.neon -l 3 ./src
```

keine Funktionen in includeten Dateien aufrufen - PHPStan führt es sonst aus

für Properties ohne Typen: inferPrivatePropertyTypeFromConstructor: true

es gibt 7 Level, bald noch einen Level 8
starte bei 0, erhöhe wenn Zeit ist, kommt auch auf die Codebasis an
inkrementierend: L 2 enthält auch alle Prüfungen von L 1

`Foo::class` macht keinen Autoload, gibts die Klasse nicht gibt PHP trotzdem den String zurück

Array Keys mit Float werden zu int gecastet
`[1 => 'Foo', '1' => 'woop', 1.9 => 'Meh']` -> `[1 => "Meh"]`

Playground: phpstan.org

ignoreErrors kann man einschränken auf bestimmten Pfad via `paths`

PHPStan möchte dass man auf PHP schaut wie als wäre es eine kompillierende Sprache

Level Konfigurationen hier: vendor/phpstan/phpstan/conf

Level 1: analysiert u.a. undefinierte Variablen
Level 2: analysiert u.a. PHPDocs
Level 3: analyisiert Verletzung von Liskov Substitution Principle, Generatoren
Level 4: always true/false, unerreichbarer Code
Level 5: wrong function argument types

Level 6+7 sind brutal und für die meisten Projekte bedeutet es viel Refactoring

Level 6: checking union types `/** @var int|string */` - es kann aber Situationen geben wo es als String verrabeitet wird - kann man mit `assert($this->foo !== null)` abprüfen - aber wird mit `NullItem` und `WorkingItem` abgebildet

Traits werden besonders behandelt

phpstan/phpstan-strict-rules -- u.a. kein `empty()` !!

Asserts verstehen:

- phpstan/phpstan-webmozart-assert
- phpstan/phpstan-beberlein-assert

neuer Level 6 (zw. jetzigen 5 und 6) wird Vars ohne Typ ziehen

dynamischen Code vermeiden (`new $foo()` , `$foo->$bar()`)

magische Dinge definieren: `@property`, `@method`
`/** @var Foo $foo */` kann Typisierung der Var überschreiben, z.B. bei Nullable -> `/** @var ?string $foo */`

Array Typehint:

- `User[]` -> alte Schreibweise
- `array<User>` bzw. `array<int, User>` -> neue Schreibweise

Typehint für eigene Instanz: `/** @return static */`

`new self();` ist unsicher, sollte nur in `final class` verwendet werden

`@return Collection|Item[]` heißt eine `Collection mit Items` - kommt noch aus Netbeans Zeiten -- PHPSTan wendet versch. Heuristiken an -- besser `Collection<Item>|array<Item>`

Union Type: `Foo|Bar` - entweder der eine oder der andere Typ

- es können sicher Methoden aufgerufen werden die in beiden Typen vorhanden sind

Intersection Type: `Foo&Bar` - kann Methoden aus beiden Methoden vervwwnet werden kann

- für Mock Objects
- um mehrere Interfaces abzufragen

geht auch beides: `(Foo&Bar)|Baz`

Generics

```
/**
 * @template T of \Exception
 * @param T $param
 * @return T
 */
```

gibt Instanz für Klasse zurück

```
/**
 * @template T of \Exception
 * @param class-string<T> $param
 * @return T
 */
```

kann man auch über Klasse platzieren
gut für Collections

da IDE noch nicht so weit ist kann man die Annotations Prefixen
`/** @phpstan-template T */`

kann sein dass der Tag später noch zu `@generic` umbenannt wird

eigene Regeln?

- Dinge die zu opinionated sind
- Codestyle erzwingen
- oft vorkommende Code Issues absichern - Devs können schlecht sauer über Beschwerden von Maschinen sein, bei Menschen in Reviews sieht das anders aus

Expressions haben einen `Type`, können in andere Expressions überführt werden, z.B. Variablen, Zuweisungen, Instanceof, Isset, print (!) ->
Statements sind Sprachkonstrukte, haben keinen Type, u.a. if, while, echo (!)

Type Interface um Typen miteinander zu vergleichen

Eigene Regel: EventData in Event

```
class EventDataRule implements \PHPStan\Rules\Rule
{

  /**
   * @return string Class implementing \PhpParser\Node
   */
  public function getNodeType(): string
  {
    return \PHPStan\Node\InClassMethodNode::class;
  }

  /**
   * @param \PHPStan\Node\InClassMethodNode $node
   * @param \PHPStan\Analyser\Scope $scope
   * @return (string|RuleError)[] errors
   */
  public function processNode(\PhpParser\Node $node, \PHPStan\Analyser\Scope $scope): arr
  {
    $orgNode = $node->getOriginalNode();
    $stmts = $orgNode->stmts;
    $fn = $scope->getFunction();
    if ($fn->getName() !== 'setData') {
      return [];
    }
    $classRfl = $scope->getClassReflection();
    if (!$classRfl->getNativeReflection()->implementsInterface(Event::class)) {
      return [];
    }
    if (count($stmts) === 0) {
      return ['Should be a EventData check'];
```

```
    }
    $firstStmt = $stmts[0];
    if (!$firstStmt instanceof If_)  {
      return ['EventData at first!'];
    }
    $cond = $firstStmt->cond;
    if (!$cond instanceof BooleanNot) {
      return ['Please check EventData check with !'];
    }
    $instanceOf = $cond->expr;
    if (!$instanceOf instanceof Instanceof_) {
      return ['Please check EventData check with instanceof'];
    }
    $class = $instanceOf->class;
    if ($class instanceof Expr) {
      return ['Aaaahr... Expr'];
    }
    $targetName = $classRfl->getName() . 'Data';
    if ($class->toString() !== $targetName ) {
      return ['Name it '.$targetName];
    }

    return [];
  }
}
```

```php
<?php declare(strict_types = 1);
namespace WS;

class ThisEventData implements EventData
{
}

class MyEventData implements EventData
{
}

/**
 * @implements Event<MyEventData>
 */
class MyEvent implements Event
{
  /** @var MyEventData */
  private $data;

  public function setData(EventData $data): void
  {

  }
}

interface EventData
{
```

```
}

/**
 * @template T of EventData
 */
interface Event
{
  /**
   * @phpstan-param T $data
   */
  public function setData(EventData $data): void;
}

$e = new MyEvent();
$e->setData(new MyEventData()); // Yay!
$e->setData(new ThisEventData()); // Boom!
```

Test:

```
class MyEvent1 implements Event
{
  /** @var EventData */
  private $data;

  public function setData(EventData $data)
  {
    if (!$data instanceof MyEvent1Data) {
      throw new \RuntimeException();
    }
  }
}

class MyEvent2 implements Event
{
  /** @var EventData */
  private $data;

  public function setData(EventData $data)
  {
    if (!$data instanceof MyEvent1Data) {
      throw new \RuntimeException();
    }
  }
}
```

```
class EventDataRuleTest extends RuleTestCase
{
  protected function getRule(): Rule
  {
    return new EventDataRule();
  }
```

```
  public function testRule(): void
  {
    $this->analyse([__DIR__.'/MyEvent.php'], [
      [
        'Name it WS\MyEvent2Data',
        23
      ]
    ]);
  }
}
```

Extensions

[https://blog.martinhujer.cz/how-to-configure-phpstan-for-symfony-applications/](How to configure PHPStan for Symfony applications)

phpstan-symfony

`includes` in `phpstan.neon` z.B. `vendor/phpstan/phpstan-symfony/extension.neon`

phpstan/extension-installer - sucht sich automatisch installierte Extensions raus, braucht man keinen include mehr machen

Custom Extensions

wenn man `instanceof ObjectType` nutzt, will man eigentlich `isSuperTypeOf` nutzen

PHP hat Variants ?!

- strtok()
- PDO::query()

ImmutableObjectRule

git@github.com:ondrejmirtes/phpstan-workshop.git

---

gut zu wissen:

- `/** @var Bla $a */ // ein paar Hinweise` sorgt dafür dass PHPStan den Typehint nicht mehr erkennt
- Versionsschema: 0.major.minor -- kein Fix-Level -- gibt noch Abwärtskomp. Probleme -- außerdem mag er keine großen Zahlen