Matthias Noback
@matthiasnoback
info@matthiasnoback.nl

WIFI:
Network: WeWork
Password: P@ssw0rd

Practicing Domain-Driven **Entity** and **Value Object** design

Slides: TODO

Installation

- Go to https://github.com/matthiasnoback/testing-playground
- Clone the project.
- `git checkout practicing_value_object_design`
- Follow the instructions in `README.md`.
- `git pull`

Fetch recent changes!

Object Design Style Guide

Matthias Noback

MEAP

MANNING

Part 1: Value Objects

Values

```
1, 'a', 12.3456, 'test',
 new DateTimeImmutable()
```

Types

```
int, string, float,
DateTimeImmutable
```

Value Object

*In the same category as a value,
but now it's an object*

Primitive types

Data and behavior are separated

Primitive types

```
$string = 'test';
$uppercase = strtoupper($string);
```

Object types

Data and related behaviors are combined

Object types

```
$string = new String('test');
$uppercase = $string->toUpper();
```

Value Object

Wrapper object for a primitive value

Whole Value

Combining multiple values
(prevent "Data Clump" code smell)

Whole Value

```
// before
public function moveTo(int $x, int $y)

// after
public function moveTo(Position $position)
```

Whole Value

Provide context for a certain value

Whole Value

```
public function pay(int $amount, string $currency)

public function pay(Money $amount)
```

Whole Value

The general form: a quantity, and its unit

Guarantee consistency

```
final class EmailAddress
{
   public function __construct(string $emailAddress)
   {
      if (...) {
         throw new InvalidArgumentException(...);
      }
   }
}
```

Guarantee consistency

```
final class GeoLocation
{
   public function __construct(float $latitude, ...)
   {
      if ($latitude < 90.0 || $latitude > 90.0) {
         throw new InvalidArgumentException(...);
      }
   }
}
```

Use standard assertions

```
public function __construct(float $latitude, ...)
{
    Assertion::range($latitude, -90.0, 90.0);
}
```

Offer possibilities

"You can sum a list of amounts"
"You can convert amounts to a different currency"
…

Offer possibilities

```
$converted = $amount / $exchangeRate;

// versus

final class Money
{
    public function convert(
        ExchangeRate $exchangeRate
    ): Money {
        ...
    }
}
```

Offer possibilities

Value objects attract behavior

Correct usage

You can’t add meters to square meters

Correct usage

```
// before: $distance and $area are both floats
$unknownUnit = $distance + $area;


// after: objects need to explicitly support addition
final class Distance
{
    public function plus(Distance $other): self
    {
        // ...
    }
}

$distance->plus($otherDistance);
$distance->plus($area); // fatal error
```

Type conversions

```
public function amount(): float
{
   return $this->tariff * $this->quantity;
}

public function amount(): Money
{
   return $this->tariff->multiply($this->quantity);
}
```

Standard Types

E.g. currencies, country codes (according to standards)

Standard Types

```
public function __construct(string $currency)
{
   if (!in_array($currency, [
      'EUR',
      'GBP',
      'USD',
      ...
   ]) {
      throw new InvalidArgumentException(...);
   }
}
```

Standard Types

```
final class Currency
{
   private function __construct(string $currency)
   {
      // no need for extra validation...

      $this->currency = $currency;
   }

   public static function USD(): self
   {
      return new self('USD');
   }
}
```

Safe to use

Anywhere you see a `Currency` *object,*
you know it's "valid"

Immutable

Just like any other value; a Value Object is an extension of the type system

Immutable

```
$money->setAmount(...);
$money->setCurrency(...);

// versus

$money = $money->add(...) // multiply(), subtract(), etc.
$money = $money->convert(...);
```

Immutable

Replace with a new instance

Immutable

Modifiers return a transformed copy

Immutable

It’s safe to pass Value Objects around, or keep as instance variables

“DDD” value object

Describes some aspect of an entity

Value equality

Allow comparison on the whole object (using ==)

Value equality

```
assertSame('USD', $money->currency()->toString());

// versus

assertEquals(new Currency('USD'), $money->currency());

// or even better

assertEquals(new Money(100, new Currency('USD')), $money);
```

Group exercise

- Look at the `SalesInvoice` and `Line` classes.
- Suggest 10 values that could be promoted to value objects.

Group exercise

- How would you prioritize the list?
- Then: prioritize the list

Example

```
public function addLine(
  string $vatCode,
  /* ... */
): void {
  Assertion::inArray($vatCode, ['S', 'L']);

  /* ... */
}
```

Step 1

```
use Assert\Assertion;

final class VatCode
{
    private function __construct(string $vatCode)
    {
        Assertion::inArray($vatCode, ['S', 'L']);
        $this->code = $code;
    }

    public static function fromString(
        string $vatCode
    ): self {
        return new self($vatCode);
    }
}
```

Step 2

```
public function addLine(
    VatCode $vatCode,
    /* ... */
): void {
    /* ... */
}
```

Group exercise

Start working on designing the value object:

- Work in pairs
- Write unit tests
- Run the tests (don’t break anything)
- Make small commits so you can later share your improvements with the group

Value equality - Pitfall

Don’t just add an `equals()` *method on every Value Object class*

Value object – General pitfall

There shouldn't be an `interface` *for Value Objects*

Side-effect free behavior

Don’t change the behavior of the system in any observable way

Pure functions

Don't use IO (system calls, e.g. current time, randomness, network, file system)

Pure functions

“Juggling data”

Pure functions

```
public function vatRate(): float
{
    if (new DateTime('now') <
        DateTime::createFromFormat('Y-m-d', '2019-01-01')
    ) {
        return 6.0;
    } else {
        return 9.0;
    }
}
```

Trade-offs

Primitive Obsession versus *Lazy Class*

Trade-offs

Specific versus *Generic*
(e.g. should you use a value object library?)

Serialization

Loading/storing

Serialization

```
public function asString(): string
public function asInt(): int
public function asArray(): array

/*
 * Re-constructing should be possible using already
 * existing constructors
 */

public static function fromString(string ...): Type
```

Trade-offs

Evolutionary design versus *Building blocks first*

Some code smells related to value objects

- Shotgun surgery
- Duplicate code
- Data clumps
- Primitive obsession
- Lazy class
- Speculative generality

Part 2: Entity design

Core aspects

- Identity
- Change & Lifecycle
- Invariant protection

Entity Design: Invariant protection

- Invariant: something that's always true
- At creation time (constructor)
- At modification time (methods)

Creating an entity

```
final class SalesInvoice
{
    public function __construct(
        CustomerId $customerId,
        Date $date,
        Currency $currency
    ) {
        // ...
    }
}
```

Creating an entity

- Define the minimum amount of information you need.
- Define specific aspects of each piece of information.
- Define the relations between the information provided.

Creating an entity

- “A sales invoice is always created for a single customer.”
- “Amounts on a sales invoice always have a known and specific currency.”
- “A new sales invoice always has a date”

Creating an entity

- Use a static function ("named constructor") instead of a regular constructor.
- Make the regular constructor private.

Assignment

- Create a proper constructor for the `SalesInvoice` entity.
- What should be minimally provided upon instantiation? Do the provided values need to be validated?
- Use a named constructor.

Entity design: Change & Lifecycle

- An entity gets created,
- Can be modified
- Can be discarded
- *Aim for entities to be the only type of object with mutable state*

Modifying an entity

```
final class SalesInvoice
{
    public function addLine(
        ProductId $productId,
        int $quantity
    ): void {
        // ...
    }
}
```

Modifying an entity

- Define the minimum amount of information you need.
- Define specific aspects of each piece of information.
- Define the relations between the information provided.
- Define valid state transitions.

Modifying an entity

- “You can add a line, which always refers to a product, and has a quantity.”
- “The quantity should be more than 0.”
- “You can’t add the same product more than once.”

Assignment

- Apply the rules for modifying an entity's state to the existing `addLine()` method.

Action-based

- Think of the interactions: what behavior does an entity offer?
- E.g. instead of `setFinalize()`, add a `finalize()` method.
- Finalizing is a new phase in the life of an invoice

Action-based

- Also: you can't just "unfinalize" an invoice.
- Same for `setCancelled()`

State machine

- “You can’t add a line to an already finalized invoice.”
- “You can't finalize a canceled invoice.”
- “You can’t cancel an invoice that has already been finalized”

State machine

- Compare the state before and after calling the method.
- Is the *state transition* allowed? Does it make sense?

Assignment

- Change the setters of `SalesInvoice` to action (command) methods.
- Protect against invalid state transitions and prevent actions that aren't allowed.

Entity Design: Identity

- An object that changes needs to be identifiable
- An entity's identity is part of the object's minimum set of information

Identity

- Provided as constructor argument
- Not dependent on infrastructure (e.g. auto-incremented integer ID)

Identity: type

- Integer (generated from a sequence)
- UUID (generated from time and randomness)
- Wrapped in Value Object

The repository provides the next identity

```php
interface SalesInvoiceRepository
{
    public function getById(SalesInvoiceId $salesInvoiceId): SalesInvoice;

    public function save(SalesInvoice $salesInvoice): void;

    public function nextIdentity(): SalesInvoiceId
}
```

Assignment

- Create a repository for `SalesInvoice` entities. Define an interface and a class.
- It's okay if the implementation simply keeps the entities inside an array.
- Make sure `SalesInvoice` has an ID when it gets instantiated.
- Define the `SalesInvoiceId` as a value object

Testing an entity

- Don’t test the constructor
- Don’t expose all its state
- Test for invariants
- Test state transitions

Don’t test the constructor

- ~~Test that you can get out what you put in~~
- Copy data to a property only when you have interesting behavior that requires this.
- Let a test force you to remember the data inside the object.

Don’t expose all its state

- ~~Add setters and getters for all fields~~
- Only add getters if something other than a test needs it
- Consider using a separate read model.
- Instead of setters, provide more meaningful methods.

Test for invariants

- Expect invalid argument exceptions.
- Test edge cases.
- A happy path will usually be covered by one of the other tests.

Test state transitions

- Expect logic exceptions for invalid state transitions.
- Try different sequences of events.