ABSTRACTION

Jorard Andringa, David Veen

21.5

AGENDA

- Philosophise about abstraction
- Why functions are contravariant in their argument
- Portals to different worlds
- Zen Mastery

AGENDA

- Philosophise
- ~~Why functions are contravariant in their argument~~
- Cooking
- Portals to different worlds
- Zen Mastery

WHY ABSTRACTION?

- fundamental to programming
- will change the way you view your craft
- (or not)

DIFFICULTY LEVEL


42
DON'T
PANIC

PHILOSOPHISE

WHAT IS ABSTRACTION?

> *Our ability to group things by the way in which they differ, while regarding them as the same in every other respect.*

- cardinality ('4')
- size
- price
- position
- ...

ABSTRACTION VS PRECISION

> *The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise.*
>
> *-- Dijkstra*

Cogs Inc.

20705 Valley Green Drive

Cupertino, California

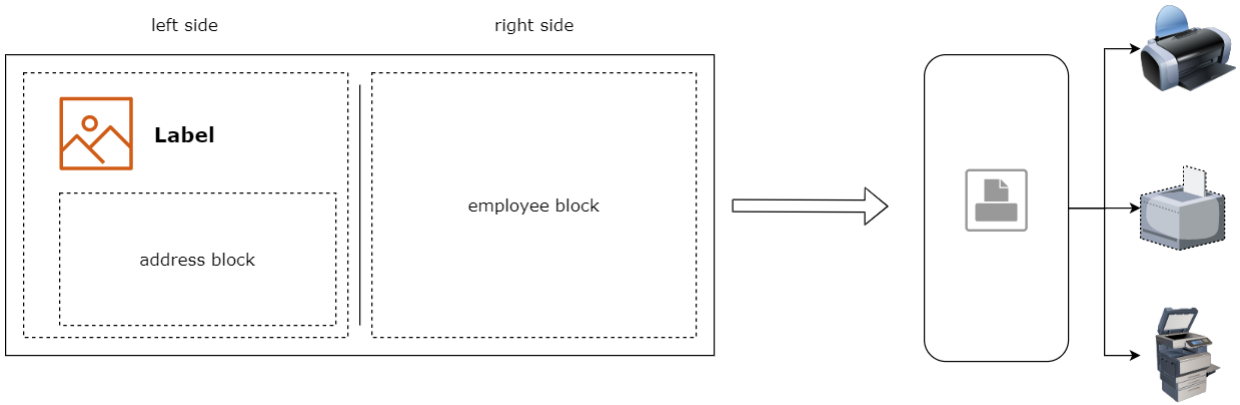
MAIN 408 794 0058

FAX 408 794 0074

<name> <surname>

<job title>

Xerox 9700 Electronic Printing System


left side
right side
Label
address block
employee block

COMPOSITIONALITY

> *Deconstruct a problem into elements that can be rearranged.*

Program:

- components
- interfaces

Code:

- functions
- data structures

ABSTRACTION RECIPES

TIMES TWO

.js

```js
const twoTimesTwo = () => {
  return 2 * 2
}

const threeTimesTwo = () => {
  return 3 * 2
}
```

.py

```py
def two_times_two():
    return 2 * 2


def three_times_two():
    return 3 * 2

```

PARAMETRISE 1: DOUBLE/TRIPLE

.js

```js
const double = (n) => {
  return n * 2
}

const triple = (n) => {
  return n * 3
}
```

.py

```py
def double(n):
    return n * 2


def triple(n):
    return n * 3

```

PARAMETRISE 2: MULTIPLICATION

.js

```js
const multiply = (a, b) => {
  return a * b
}

const double = (a) => {
  return multiply(a, 2)
}

const triple = (a) => {
  return multiply(a, 3)
}
```

.py

```py
def multiply(a, b):
    return a * b


def double(a):
    return multiply(a, 2)


def triple(a):
    return multiply(a, 3)

```

FUNCTION ABSTRACTION RECIPE 1

1. **compare**
   lookalike functions
2. **connect**
   specify essential differences
3. **change**
   parametrise, replace, redefine, eliminate
4. **check**
   test!
5. **challenge**
   use cases, edge cases, existing abstractions

CATS & DOGS

.js

```js
const animals = [
  'giraffe', 'cat', 'elephant'
]

const hasDog = (animals) => {
  for (let animal of animals) {
    if (animal == 'dog')
      return true
  }
  return false
}

const hasCat = (animals) => {
  for (let animal of animals) {
    if (animal == 'cat')
      return true
  }
  return false
}
```

.py

```py
animals = [
    'giraffe', 'cat', 'elephant'
]

def hasDog(animals):
    for animal in animals:
        if animal == 'dog':
            return True
    return False



def hasCat(animals):
    for animal in animals:
        if animal == 'cat':
            return True
    return False


```

CATS & DOGS

.js

```
const animals = [
  'giraffe', 'cat', 'elephant'
]

const hasDog = (animals) => {
  for (let animal of animals) {
    if (animal == 'dog')
      return true
  }
  return false
}

const hasCat = (animals) => {
  for (let animal of animals) {
    if (animal == 'cat')
      return true
  }
  return false
}
```

.py

```
animals = [
    'giraffe', 'cat', 'elephant'
]

def hasDog(animals):
    for animal in animals:
        if animal == 'dog':
            return True
    return False



def hasCat(animals):
    for animal in animals:
        if animal == 'cat':
            return True
    return False
```

CATS & DOGS

```.js
const animals = [
  'giraffe', 'cat', 'elephant'
]

const hasAnimal = (kind, animals) => {
  for (let animal of animals) {
    if (kind == animal)
      return true
  }
  return false
}

const hasDog = (animals) =>
  hasAnimal('dog', animals)

const hasCat = (animals) =>
  hasAnimal('cat', animals)
```

```.py
animals = [
    'giraffe', 'cat', 'elephant'
]

def hasAnimal(kind, animals):
    for animal in animals:
        if animal == kind:
            return True
    return False



def hasDog(animals):
    return hasAnimal('dog', animals)

def hasCat(animals):
    return hasAnimal('cat', animals)
```

CATS & DOGS

```.js
const animals = [
  'giraffe', 'cat', 'elephant'
]

const hasDog = (animals) =>
  animals.includes('dog')

const hasCat = (animals) =>
  animals.includes('cat')
```

```.py
animals = [
    'giraffe', 'cat', 'elephant'
]

def hasDog(animals):
    return 'dog' in animals

def hasCat(animals):
    return 'cat' in animals
```

OLDEST VERTEBRATE

Given a set of animals, find out which animal is oldest (relative to human years).
We are interested only in vertebrates.

```
# Goldfish
age * 5

# Dog
16 * ln(age) + 31

# Cat
1 = 15
2 = 25
(age - 2) * 4 + 25
```

NAIVE IMPLEMENTATION

.js

```
const getRelativelyOldestVertebrate = (animals) => {
  let eldest = { animal: null, relativeAge: 0 }

  for (const animal of animals) {
    if (animal.isVertebrate) {
      let relativeAge = 0

      switch (animal.genus) {
        case 'carassius': {
          relativeAge = animal.age * 5
        } case 'canis': {
          relativeAge =
            Math.round(16 * Math.log(animal.age) + 31)
        } case 'felis': {
          switch (animal.age) {
            case 1: relativeAge = 15
            case 2: relativeAge = 25
            default: {
              relativeAge = (animal.age - 2) * 4 + 25
            }
          }
        }
      }

      // compare relative age to eldest so far
      if (relativeAge > eldest.relativeAge) {
        eldest = { animal, relativeAge }
      }
    }
  }

  return eldest
}
```

NAIVE IMPLEMENTATION

```js
const getRelativelyOldestVertebrate = (animals) => {
  let eldest = { animal: null, relativeAge: 0 }

  for (const animal of animals) {
    if (animal.isVertebrate) {
      let relativeAge = 0

      switch (animal.genus) {
        case 'carassius': {
          relativeAge = animal.age * 5
        } case 'canis': {
          relativeAge =
            Math.round(16 * Math.log(animal.age) + 31)
        } case 'felis': {
          switch (animal.age) {
            case 1: relativeAge = 15
            case 2: relativeAge = 25
            default: {
              relativeAge = (animal.age - 2) * 4 + 25
            }
          }
        }
      }

      // compare relative age to eldest so far
      if (relativeAge > eldest.relativeAge) {
        eldest = { animal, relativeAge }
      }
    }
  }

  return eldest
}
```

DOMAIN ABSTRACTION

1. **identify** (complexity)
2. **demarcate**
3. **extract**
4. **check**
5. **challenge**

BETTER IMPLEMENTATION

```.js
const relativeGoldfishAge = (fish) => {
  return fish.age * 5
}

const relativeDogAge = (dog) => {
  return Math.round(16 * Math.log(dog.age) + 31)
}

const relativeCatAge = (cat) => {
  switch (cat.age) {
    case 1: return 15
    case 2: return 25
    default: {
      return (cat.age - 2) * 4 + 25
    }
  }
}

const getRelativeAge = (animal) => {
  switch(animal.genus) {
    case 'carassius': return relativeGoldfishAge(animal)
    case 'canis': return relativeDogAge(animal)
    case 'felis': return relativeCatAge(animal)
    default: return 0
  }
}

const getRelativelyOldestVertebrate = (animals) => {
  let eldest = { animal: null, relativeAge: 0 }

  for (const animal of animals) {
    if (animal.isVertebrate) {
      const relativeAge = getRelativeAge(animal)
```

```
      // compare relative age to eldest so far
      if (relativeAge > eldest.relativeAge) {
        eldest = { animal, relativeAge }
      }
    }
  }

  return eldest
}
```

DOMAIN ABSTRACTION

1. **identify**
2. **demarcate**
3. **extract**
4. **check**
5. **challenge**

PORTAL ABSTRACTIONS

```
1 + 1
3 * 2
6 / 4
```
.js

```
1 + 1
3 * 2
6 / 4
```
.py

```.js
const multiply = (x, y) => {
  return x * y
}
```

```.py
def multiply(x, y):
  return x * y
```

```.js
const numbers = [1, 3, 5, 7, 9]
```

```.py
numbers = [1, 3, 5, 7, 9]
```

.js

```js
const numbers = [1, 3, 5, 7, 9]

const sum = (numbers) => {
  let accumulator = 0

  for (let n of numbers) {
    accumulator = accumulator + n
  }

  return accumulator
}

sum(numbers) //? 25
```

.py

```py
numbers = [1, 3, 5, 7, 9]

def sum(numbers):
    accumulator = 0

    for n in numbers:
        accumulator = accumulator + n


    return accumulator


sum(numbers) #? 25
```

```js
const recursiveSum = (numbers, accumulator = 0) => {
  if (numbers.length == 0)
    return accumulator

  const [currentNumber, ...remainingNumbers] = numbers

  return recursiveSum(remainingNumbers, accumulator + currentNumber)
}
```

```py
def recursive_sum(numbers, accumulator=0):
    if len(numbers) == 0:
        return accumulator

    current_number, *remaining_numbers = numbers

    return recursive_sum(remaining_numbers, accumulator + current_number)
```

```js
const product = (numbers, accumulator = 1) => {
  if (numbers.length == 0)
    return accumulator

  const [current, ...remaining] = numbers

  return product(remaining, accumulator * current)
}
```

```py
def product(numbers, accumulator=1):
    if len(numbers) == 0:
        return accumulator

    current, *remaining = numbers

    return product(remaining, accumulator * current)
```

.js

```js
const conditions = [true, true, false]

const all = (conditions, accumulator = true) => {
  if (conditions.length == 0)
    return accumulator

  const [current, ...remaining] = conditions

  return all(remaining, accumulator && current)
}

all(conditions) //? false
```

.py

```py
conditions = [True, True, False]

def all(conditions, accumulator=True):
    if len(conditions) == 0:
        return accumulator

    current, *remaining = conditions

    return all(remaining, accumulator and current)


all(conditions) #? False
```

```js
const conditions = [true, true, false]

const any = (conditions, accumulator = false) => {
  if (conditions.length == 0)
    return accumulator

  const [current, ...remaining] = conditions

  return any(remaining, accumulator || current)
}

any(conditions) //? true
```

```py
conditions = [True, True, False]

def any(conditions, accumulator=False):
    if len(conditions) == 0:
        return accumulator

    current, *remaining = conditions

    return any(remaining, accumulator or current)


any(conditions) #? True
```

.js

```js
const binaryOperation = (item1, item2) => {
  ???
}

const fn = (items, accumulator) => {
  if (items.length == 0)
    return accumulator

  const [current, ...remaining] = items

  return fn(remaining, binaryOperation(accumulator, current))
}
```

.py

```py
def binary_operation(item1, item2):
  return ???


def fn(items, accumulator):
  if len(items) == 0:
    return accumulator

  current, *remaining = items

  return fn(remaining, binary_operation(accumulator, current))
```


REDUCE it!

```js
const reduce = (fn, items, accumulator) => {
  if (items.length == 0)
    return accumulator

  const [current, ...remaining] = items

  return reduce(fn, remaining, fn(accumulator, current))
}
```

```py
def reduce(fn, items, accumulator):
    if len(items) == 0:
      return accumulator

    current, *remaining = items

    return reduce(fn, remaining, fn(accumulator, current))
```

.js

```js
const add = (x, y) => {
  return x + y
}

const numbers = [1, 2, 3, 4, 5]

numbers.reduce(add, 0) //?
```

.py

```py
from functools import reduce

def add(x, y):
    return x + y

numbers = [1, 2, 3, 4, 5]

reduce(add, numbers, 0)
```

```js
const numbers = [1, 2, 3, 4, 5]

const addOne = (x) => {
  return x + 1
}

numbers.map(addOne) //? [2, 3, 4, 5, 6]

const isEven = (x) => {
  return x % 2 == 0
}

numbers.filter(isEven) //? [2, 4]
```

```py
numbers = [1, 2, 3, 4, 5]

def add_one(x):
  return x + 1


list(map(add_one, numbers)) #? [2, 3, 4, 5, 6]

def is_even(x):
  return x % 2 == 0


list(filter(is_even, numbers)) #? [2, 4]
```

```js
const isVertebrate = (animal) => {
  return animal.isVertebrate
}

const getRelativeAge = (animal) => {
  switch(animal.genus) {
    case 'carassius': return relativeGoldfishAge(animal)
    case 'canis': return relativeDogAge(animal)
    case 'felis': return relativeCatAge(animal)
    default: return 0
  }
}

const getEldest = (age1, age2) => {
  return age1 > age2 ? age1 : age2
}

animals
  .filter(isVertebrate)
  .map(getRelativeAge)
  .reduce(getEldest, 0)
```

```py
def is_vertebrate(animal):
  return animal['isVertebrate']


def get_relative_age(animal):
  if animal['genus'] == 'carassius':
    return relative_goldfish_age(animal)
  if animal['genus'] == 'canis':
    return relative_dog_age(animal)
  if animal['genus'] == 'felis':
    return relative_cat_age(animal)
  return 0

def get_eldest(age1, age2):
  return age1 if age1 > age2 else age2


vertebrates = filter(is_vertebrate, animals)
ages = map(get_relative_age, vertebrates)
eldest = reduce(get_eldest, ages)

```

ZEN MASTERY

ABSTRACTION

- find the right semantic level

2 abstraction recipes to get:

- DRY code
- that reflects the problem domain
- and is open for extension

PORTAL ABSTRACTIONS

- are used to solve a *class* of problems
- allow higher-level reasoning
- require you to study your trade

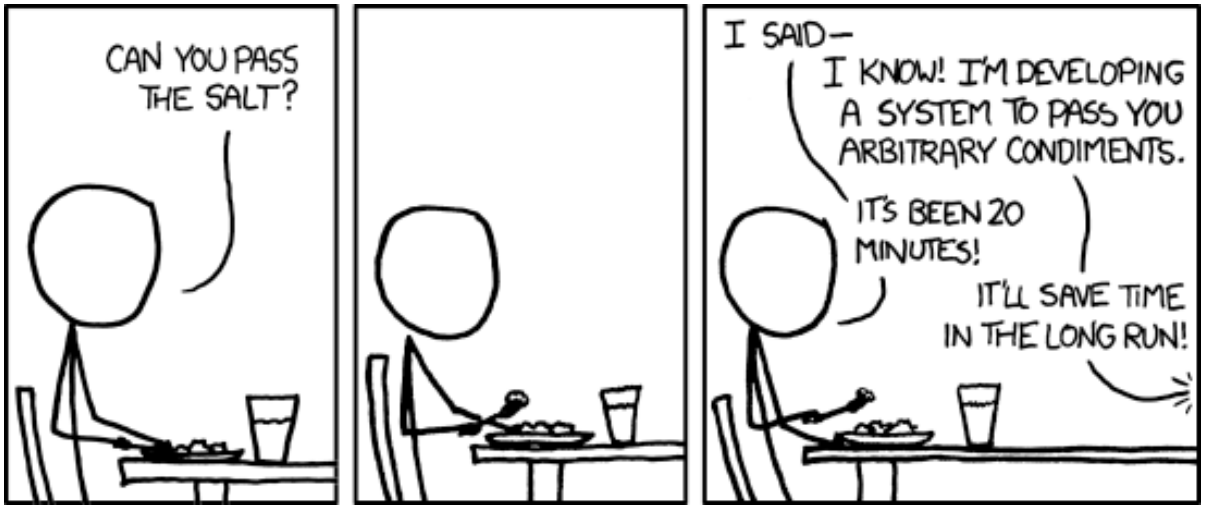
ABSTRACTION: DOWNSIDES

The trade-offs:

- indirection
- complexity
- time and effort
- polish

> *Simple is better than complex.*
> *Complex is better than complicated.*
>
> *-- Zen of Python*

OVER-ENGINEERING



WHEN (NOT) TO ABSTRACT

Bad reasons:

- reduce lines of code
- eliminate things that are similar
- eliminate things that are identical

The DRY pitfall:

> *Every piece of **knowledge** must have a single, unambiguous, authoritative representation within a system.*

"knowledge" not "code"

WHEN (NOT) TO ABSTRACT

Bad reasons:

- reduce lines of code
- eliminate things that are similar
- eliminate things that are identical

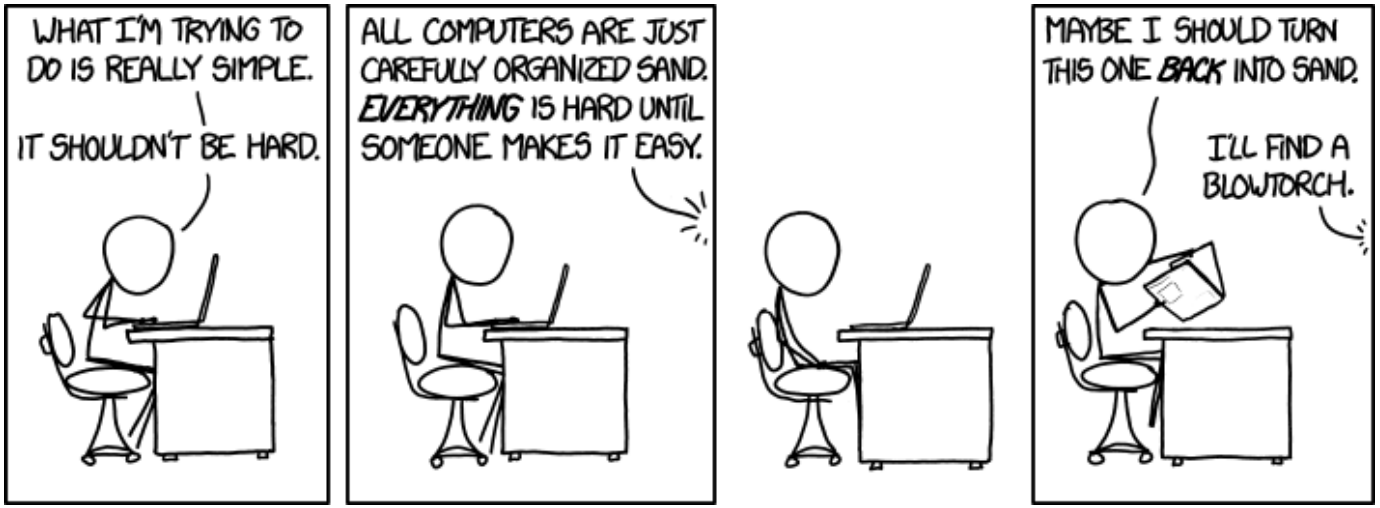
Good reasons:

- establish single point of control
- reflect the domain
- enable extension
- *simplify* through a portal abstraction

When:

- once you know the edge cases (rule of three)

YOU'VE MADE IT!



QUESTIONS?

THANKS!