Objects + Tests = Magic

(Let's try the KataPotter)

Sebastien Mosser

"On demand" lecture, 12.12.16

pictures: sxc.hu

Michalis Famelis @MFamelis · 12 oct.

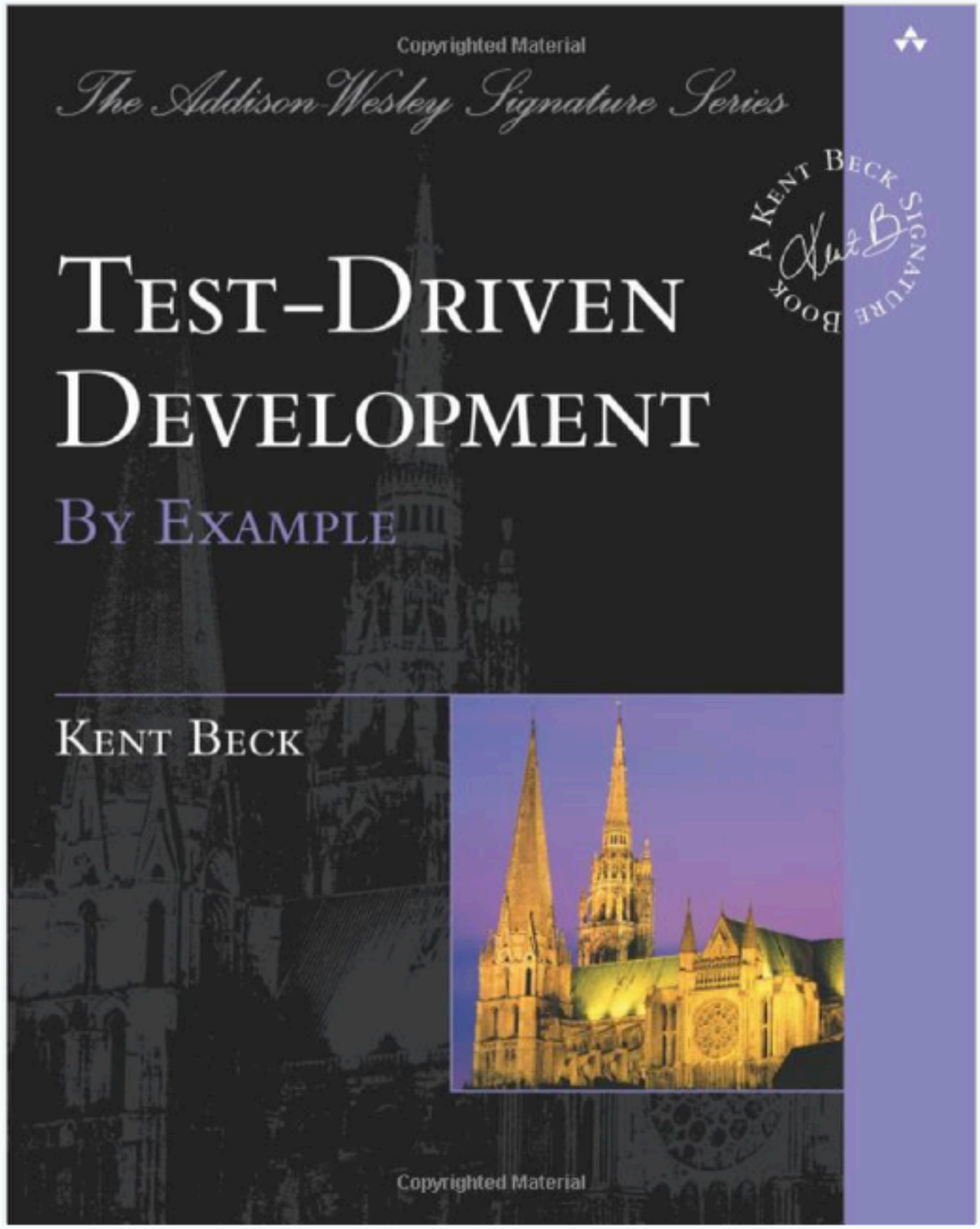
Most obvious proof that Devs are better than Wizards: Devs are often asked to perform magic. Wizards are never asked to write software.

Objectives

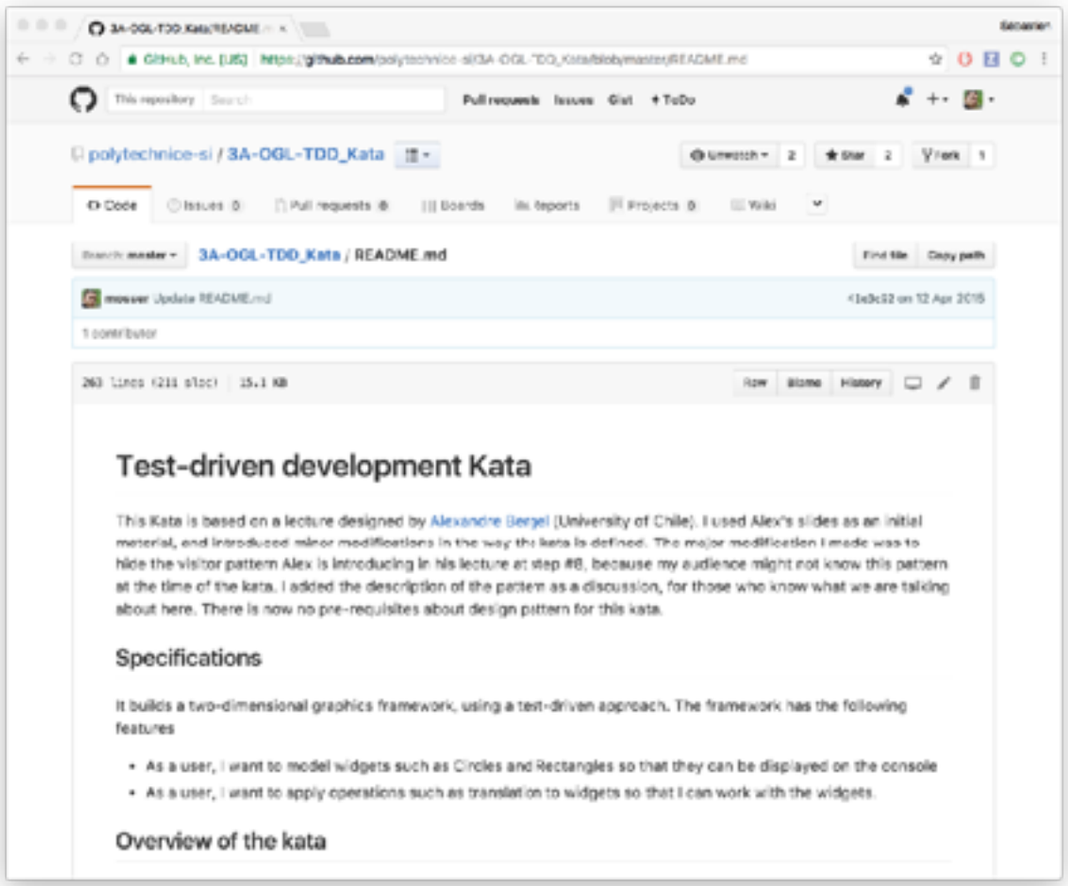
Test, Test, Test

Object-orientation

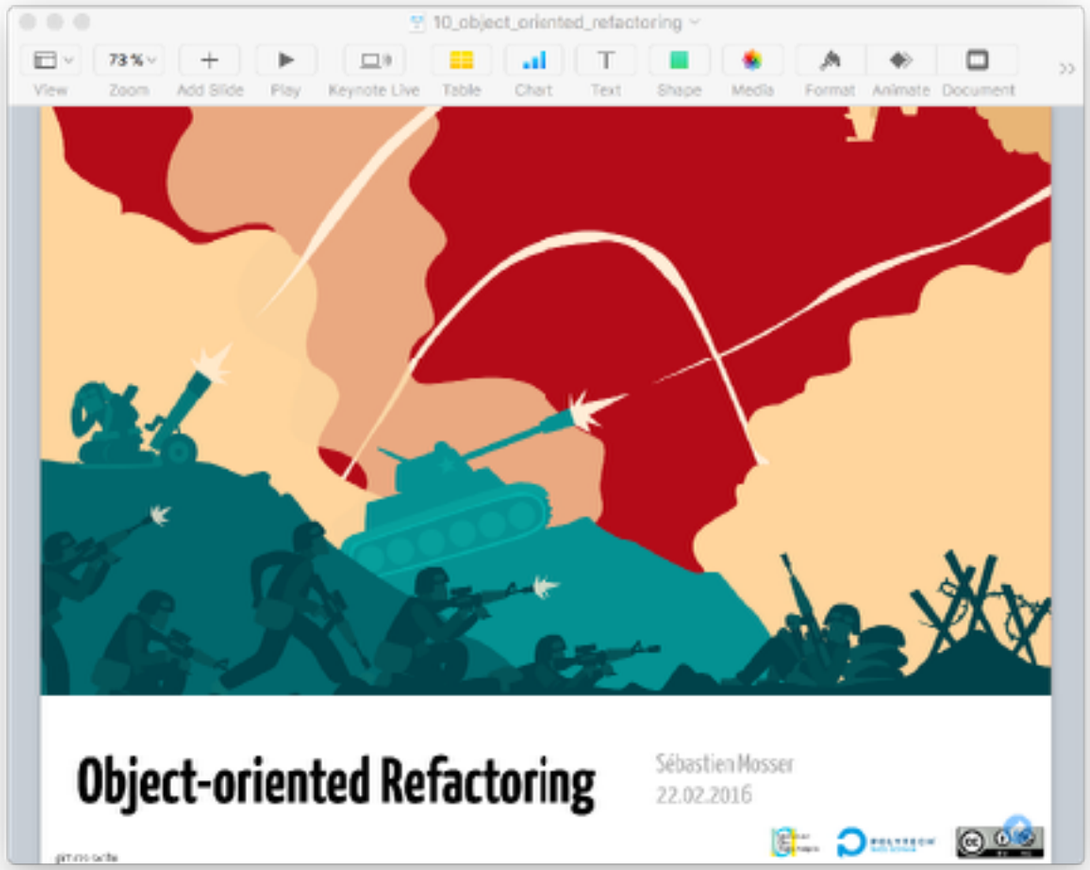
Reference


Copyrighted Material
The Addison Wesley Signature Series
A Kent Beck Book Signature
Test-Driven Development
By Example
Kent Beck
Copyrighted Material

Link with others QGL sessions (next sem.)



Test-driven development (kata)



OO refactoring (lecture)

The problem:

Selling Harry Potter books

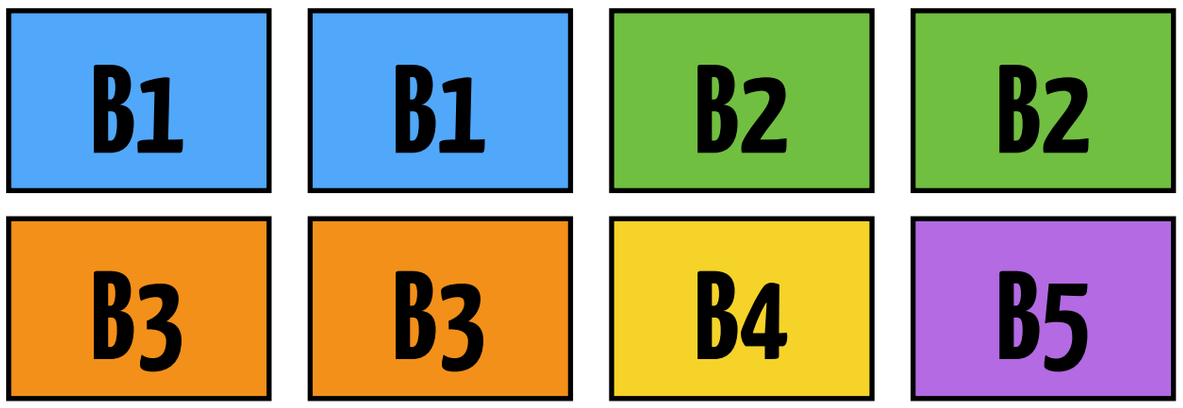
Harder than expected!

Specifications

- We consider the five firsts books. Each book costs 8 €
- Discount policies based on the number of different books to purchase:
    - 2 different books: 5%
    - 3 different books: 10%
    - 4 different books: 20%
    - The whole series: 25%
- The discount is eventually applied only on the relevant books

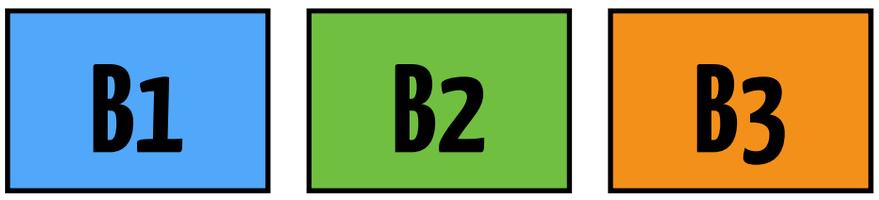
http://www.codingdojo.org/cgi-bin/index.pl?action=browse&id=KataPotter&revision=41

Pricing Policy Example



= ?

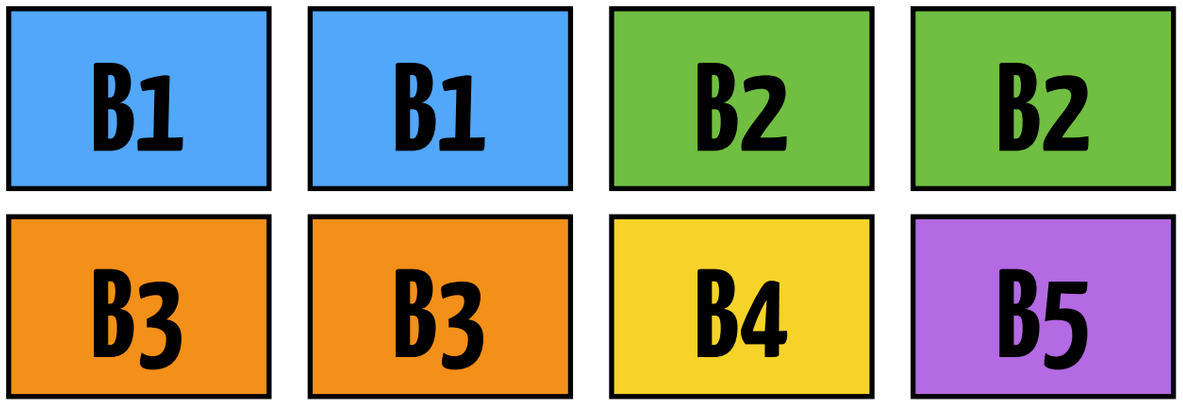
5x 8 x 0,75 = 30



3x 8 x 0,90 = 21,6

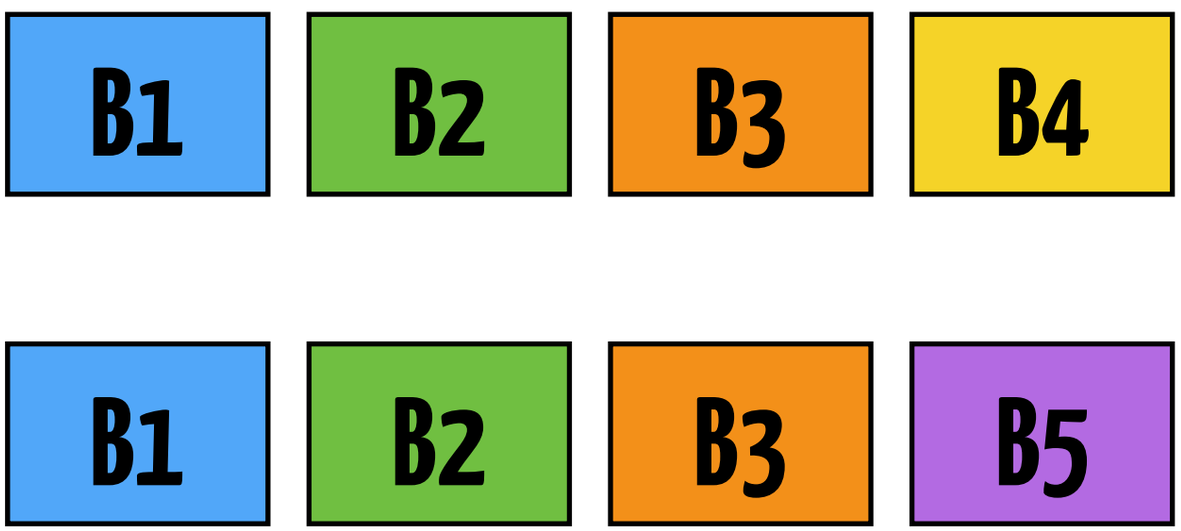
} 51,60

It's a trap !



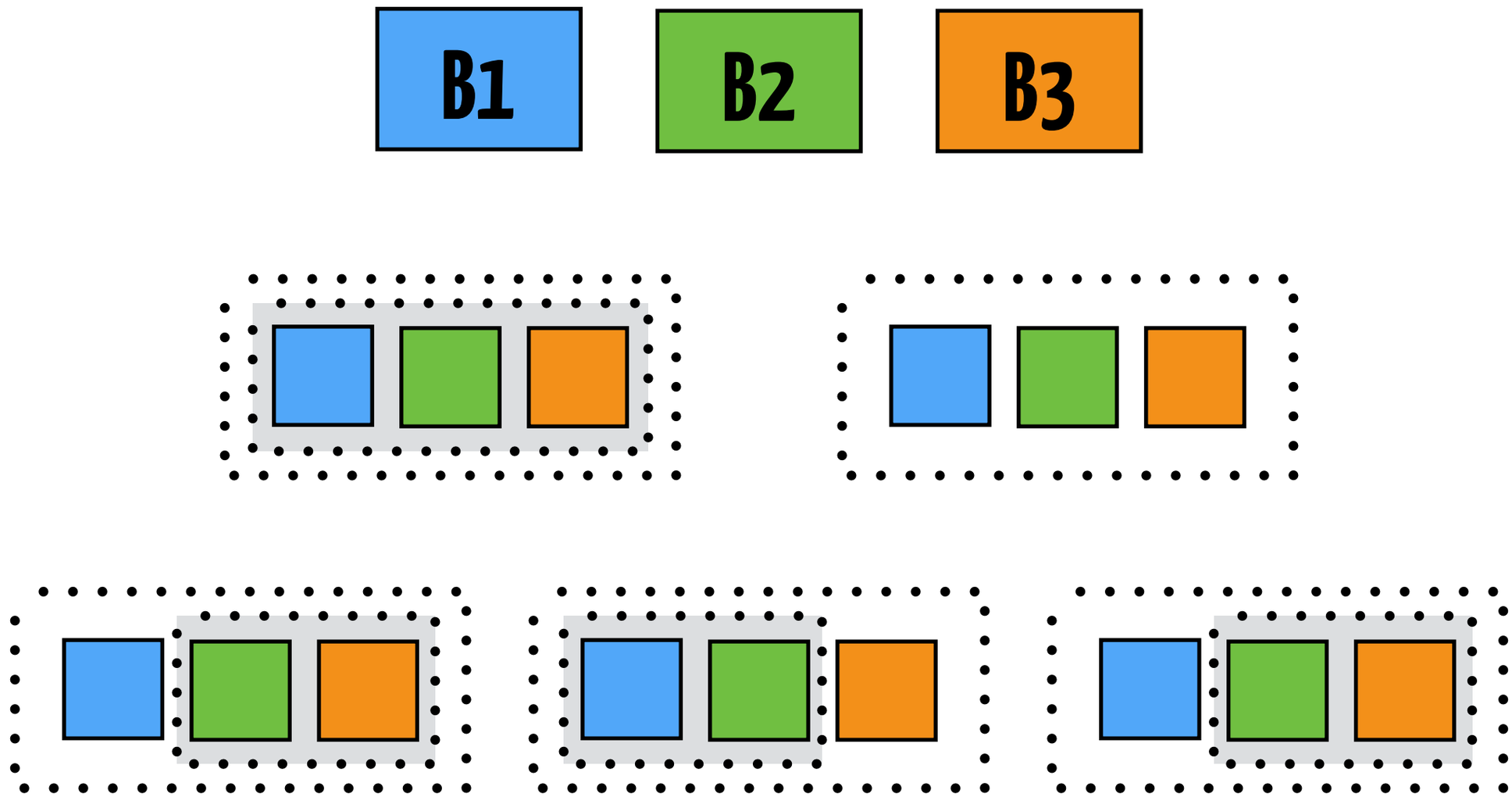
= ~~51,60~~

4 x 8 x 0,8 = 25,6



} = 2 x 25,6 = **51,20**

Combinatorial situation!



Order of magnitude: Bell number

Bell(7) = ??

877

```java
public class PotterDiscountCalculator {

    private double[] discountRates;
    private int[] discounts;

    private void init() {
        discountRates = new double[]{ 1, 0.95, 0.90, 0.80, 0.75 };
        discounts = new int[5];
    }

    public double calculatePrice(List<Integer> order) {
        init(); // resetting the calculator before computing a price
        if (order == null || order.isEmpty())
            return 0.0;
        int[] booksInOrder = calculateBooksInOrder(order);
        calculateDiscounts(booksInOrder);
        optimizeDiscounts();
        double price = 0.0;
        for (int i = 0; i < discounts.length; i++)
            price += (8 * (i + 1) * discounts[i]) * discountRates[i];
        return price;
    }

    protected void optimizeDiscounts() {
        while (discounts[2] > 0 && discounts[4] > 0) {
            discounts[2]--;
            discounts[4]--;
            discounts[3] += 2;
        }
    }

    protected void calculateDiscounts(int[] booksInOrder) {
        if (booksInOrder == null)
            return;
        int differentFromZero = 0;
        for (int i = 0; i < booksInOrder.length; i++) {
            if (booksInOrder[i] > 0) {
                differentFromZero++;
                booksInOrder[i]--;
            }
        }
        if (differentFromZero > 0) {
            discounts[differentFromZero - 1] += 1;
            calculateDiscounts(booksInOrder);
        }
    }

    private int[] calculateBooksInOrder(List<Integer> order) {
        int[] result = new int[5];
        for (int book : order)
            result[book - 1]++;
        return result;
    }
}
```

Yes

we

can !

```java
private double[] discountRates;
private int[] discounts;

private void init() {
    discountRates = new double[]{ 1, 0.95, 0.90, 0.80, 0.75 };
    discounts = new int[5];
}

public double calculatePrice(List<Integer> order) {
    init(); // resetting the calculator before computing a price
    if (order == null || order.isEmpty())
        return 0.0;
    int[] booksInOrder = calculateBooksInOrder(order);
    calculateDiscounts(booksInOrder);
    optimizeDiscounts();
    double price = 0.0;
    for (int i = 0; i < discounts.length; i++)
        price += (8 * (i + 1) * discounts[i]) * discountRates[i];
    return price;
}
```

http://bfindeiss.blogspot.fr/2013/07/the-katapotter-in-java-solved-in-31.html

```java
protected void calculateDiscounts(int[] booksInOrder) {
    if (booksInOrder == null)
        return;
    int differentFromZero = 0;
    for (int i = 0; i < booksInOrder.length; i++) {
        if (booksInOrder[i] > 0) {
            differentFromZero++;
            booksInOrder[i]--;
        }
    }
    if (differentFromZero > 0) {
        discounts[differentFromZero - 1] += 1;
        calculateDiscounts(booksInOrder);
    }
}
```

http://bfindeiss.blogspot.fr/2013/07/the-katapotter-in-java-solved-in-31.html

```java
private int[] calculateBooksInOrder(List<Integer> order) {
    int[] result = new int[5];
    for (int book : order)
        result[book - 1]++;
    return result;
}
```

```java
protected void optimizeDiscounts() {
    while (discounts[2] > 0 && discounts[4] > 0) {
        discounts[2]--;
        discounts[4]--;
        discounts[3] += 2;
    }
}
```

http://bfindeiss.blogspot.fr/2013/07/the-katapotter-in-java-solved-in-31.html

```java
@Test
public void edgeCase() {
    assertEquals( 51.20,
            calculator.calculatePrice(build(1, 1, 2, 2, 3, 3, 4, 5)), 0.01);
    assertEquals(141.20, calculator.calculatePrice(
            build(  1, 1, 1, 1, 1,
                    2, 2, 2, 2, 2,
                    3, 3, 3, 3,
                    4, 4, 4, 4, 4,
                    5, 5, 5, 5      )), 0.01);
}
```

```java
@Test
public void simpleDiscounts() {
    assertEquals(15.20, calculator.calculatePrice(build(1, 2)),           0.01);
    assertEquals(21.60, calculator.calculatePrice(build(1, 3, 5)),        0.01);
    assertEquals(25.60, calculator.calculatePrice(build(1, 2, 3, 5)),     0.01);
    assertEquals(30.00, calculator.calculatePrice(build(1, 2, 3, 4, 5)), 0.01);
}
```

```java
public class PotterDiscountCalculator {

    private double[] discountRates;
    private int[] discounts;

    private void init() {
        discountRates = new double[]{ 1, 0.95, 0.90, 0.80, 0.75 };
        discounts = new int[5];
    }

    public double calculatePrice(List<Integer> order) {
        init(); // resetting the calculator before computing a price
        if (order == null || order.isEmpty())
            return 0.0;
        int[] booksInOrder = calculateBooksInOrder(order);
        calculateDiscounts(booksInOrder);
        optimizeDiscounts();
        double price = 0.0;
        for (int i = 0; i < discounts.length; i++)
            price += (8 * (i + 1) * discounts[i]) * discountRates[i];
        return price;
    }

    protected void optimizeDiscounts() {
        while (discounts[2] > 0 && discounts[4] > 0) {
            discounts[2]--;
            discounts[4]--;
            discounts[3] += 2;
        }
    }

    protected void calculateDiscounts(int[] booksInOrde
        if (booksInOrder == null)
            return;
        int differentFromZero =
        for (int i = 0; i < bo  nOrder.len  i
            if (booksInOrder[i  0) {
                differentFromZe
                booksInOrder[i]-
            }
        }
        if (differentFromZero > 0) {
            discounts[differentFromZero - 1]  1;
            calculateDiscounts(booksInO
        }
    }

    private int[] calculateBooksInOrder(List<Integer> order) {
        int[] result = new int[5];
        for (int book : order)
            result[book - 1]++;
        return result;
    }
}
```

Seriously?

Yes we can!

Problems?

“

Bon, j'ai réfléchi un peu à ce problème et algorithmiquement, **il y a trois options** pour le résoudre : une **naïve** (ton exemple de code java) qui est exponentielle en le nombre de bouquin à acheter, une **dynamique** qui est linéaire en le nombre de bouquins à acheter (quadratique si le nombre des offres de prix est infini) et enfin une qui est **logarithmique** (amorti) en le nombre de bouquins (mais seulement avec un nombre d'offres de prix finis) et **qui sera en pratique en temps constant** (à moins d'avoir un nombre de bouquin à acheter qui ne tient pas sur un int32, mais il y a peu de chances quand même).

- Christophe Papazian

```
from itertools import product

Bundles = list(product((0,1),repeat=5))[1:]

tags = [0,800,2*8*95,3*8*90,4*8*80,5*8*75]

def bundles(t) :
    for k in Bundles :
        if all(t[i]>=k[i] for i in range(5)) :
            yield tags[sum(k)], tuple(t[i]-k[i] for i in range(5))

def best_price(t,cache={(0,0,0,0,0):0}) :
    t=tuple(sorted(t))
    try : return cache[t]
    except KeyError :
        cache[t]=res=min(a[0]+best_price(a[1]) for a in bundles(t))
        return res

def price(l) : return best_price(tuple(l.count(i) for i in range(5)))/100
```

“Mais **je trouve ça plutôt bof**, ça passe les tests, et c'est polynomial en le nombre de bouquins en entrée, mais c'est un polynome de degré 5, donc les performances sont mauvaises en pratique. **Faut que je refléchisse pour la solution logarithmique**... mais pas ce soir” - Christophe

Complexity survival guide

```
if (christophe.says("Je dois réfléchir"))
    return "Fly you fools!"
```

=> This is a non trivial problem

Better version

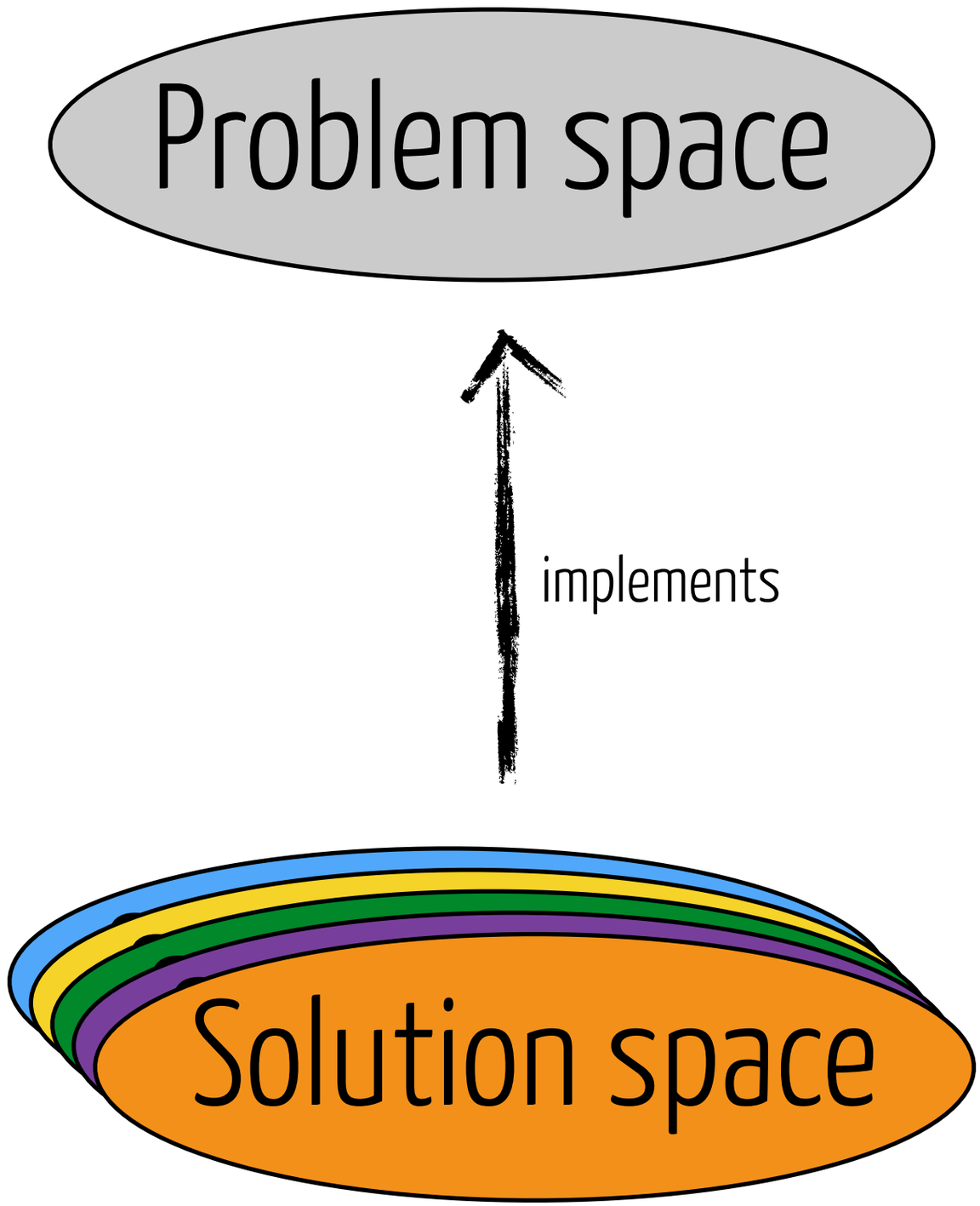

```
tags = [0,800,2*8*95,3*8*90,4*8*80,5*8*75]

def best_price(t) :
    t=tuple(sorted(t))
    m=min(t[0],t[2]-t[1])
    return tags[5]*(t[0]-m)+tags[4]*(2*m+t[1]-t[0])+
           tags[3]*(t[2]-t[1]-m)+tags[2]*(t[3]-t[2])+
           tags[1]*(t[4]-t[3])

def price(l) :
    return best_price(tuple(l.count(i) for i in range(5)))/100
```

“ok, **j'ai une version logarithmique** qui passe les tests ;P. Version python avec test, c'est linéaire par rapport au log du nombre de bouquins achetés.” - Christophe

How can one encapsulate the computation logic in an object graph to enhance maintenance?


Problem space
implements
Solution space

Software engineering rule of three

Readability

Readability
Readability

Maintenance: Open/Closed principle

Closed for **Modification**

Open for **Extension**

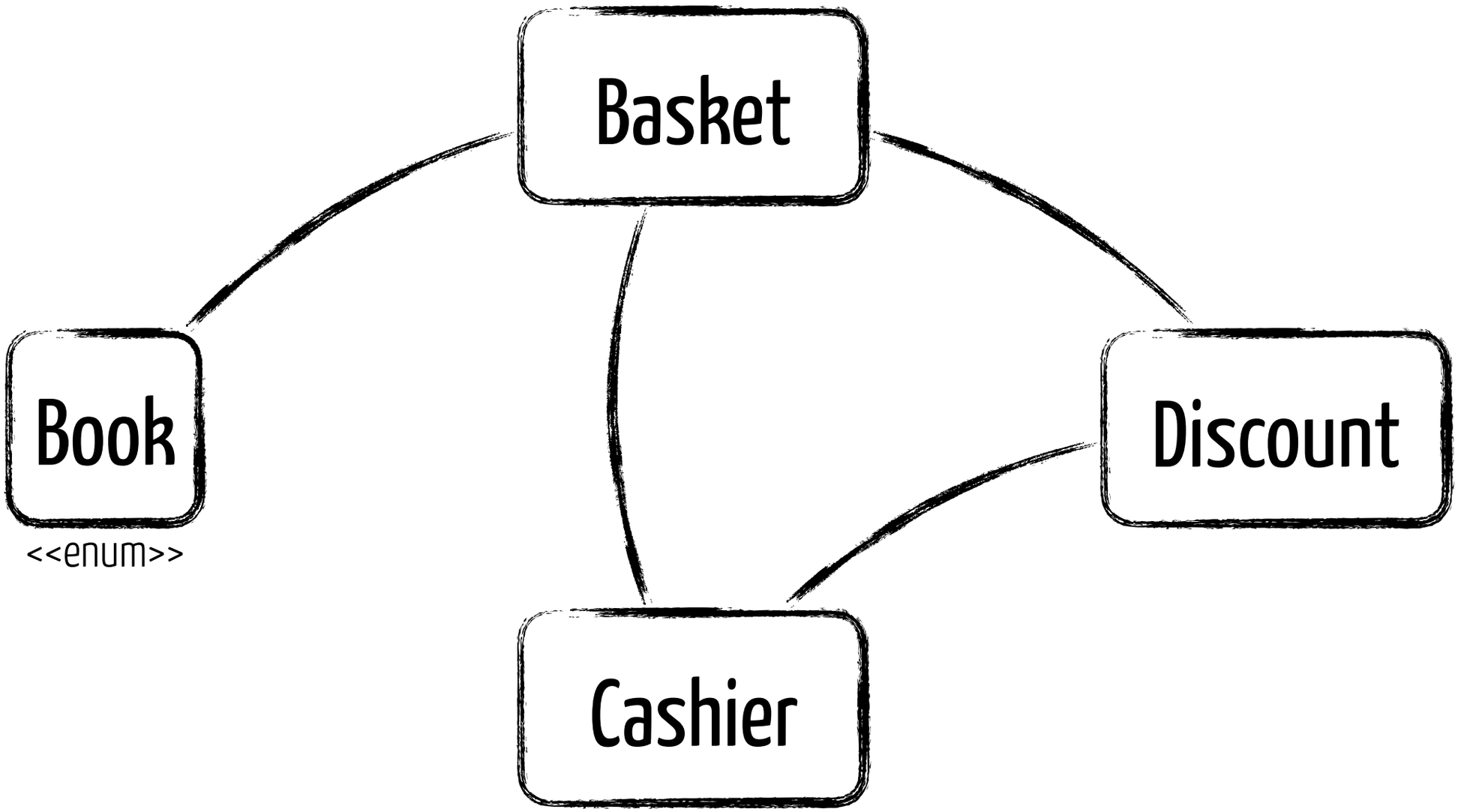
SOLIDity applied to OO code

- S : Single Responsibility
- O: Open / closed principle
- L: Liskov-compliant (substitution)
- I: Interface Segregation
- D: Dependency inversion

From **Computation** to **Objects**

Let's start the transformation

Modularisation



Computing Price for **b** ∈ Basket

---

b is empty => **0.0**

Let **discounts** = {**d** ∈ Discount | **d** can be applied to **b**}

discounts is empty => $|\textbf{books}|_b$ * **PRICE**

Let **candidates** = { **p** ∈ $\mathbb{R}$ | **d** ∈ **discounts**, **p** = compute(**b**,**d**) }

=> **min(candidates)**

NAIVE

Computing a price with a discount **d** for **b**

Let **local price** = **d** applied to **b**

Let **remaining** = clone of **b** without the books used in **d**

=> **local price** + compute(**remaining**)

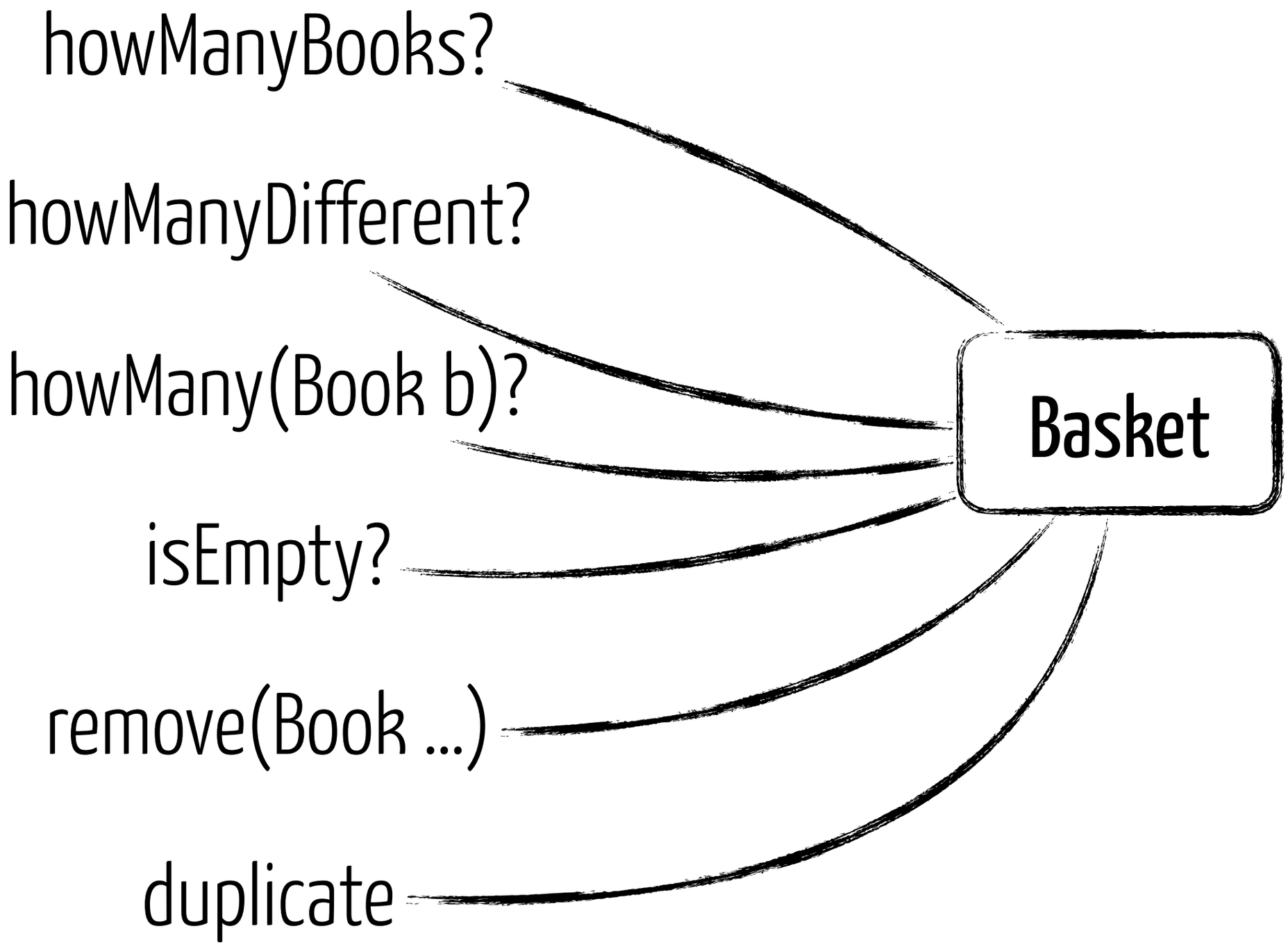
NAIVE

WAT? No getters? no setters?

Are we even coding in Java?

https://www.youtube.com/playlist?list=PLuNTRFkYD3u5ibkjD1nHP9QZXPjtvnPXL

Behavioural description



Internal implementation

```
public class Basket {

    private Map<Book, Integer> contents = new EnumMap<>(Book.class);

    public Basket() {
        for(Book b: Book.values())
            contents.put(b, 0);
    }

    public Basket(Book... chosen) {
        this();
        for(Book book: chosen) {
            contents.put(book, contents.get(book) + 1);
        }
    }
```

We don't care!

Tests rely on the public interface

```
private static final Basket empty = new Basket();
private static final Basket complete = new Basket(BOOK1, BOOK2, BOOK3, BOOK4, BOOK5);
private static final Basket azkaban = new Basket(BOOK3, BOOK3, BOOK3);

@Test
public void basketContents() {
    for(Book b: Book.values())
        assertEquals(0, empty.howMany(b));

    for(Book b: Book.values())
        assertEquals(1, complete.howMany(b));

    for(Book b: Book.values())
        if(b == BOOK3) {
            assertEquals(3, azkaban.howMany(b));
        } else {
            assertEquals(0, azkaban.howMany(b));
        }

}
```

Testing Critical Cases

```java
@Test
public void removeBooks() {
    Basket basket = new Basket(BOOK1, BOOK2, BOOK2);
    assertEquals(3, basket.howManyBooks());
    basket.remove(BOOK1);
    assertEquals(2, basket.howManyBooks());
    basket.remove(BOOK2);
    assertEquals(1, basket.howManyBooks());
}
```

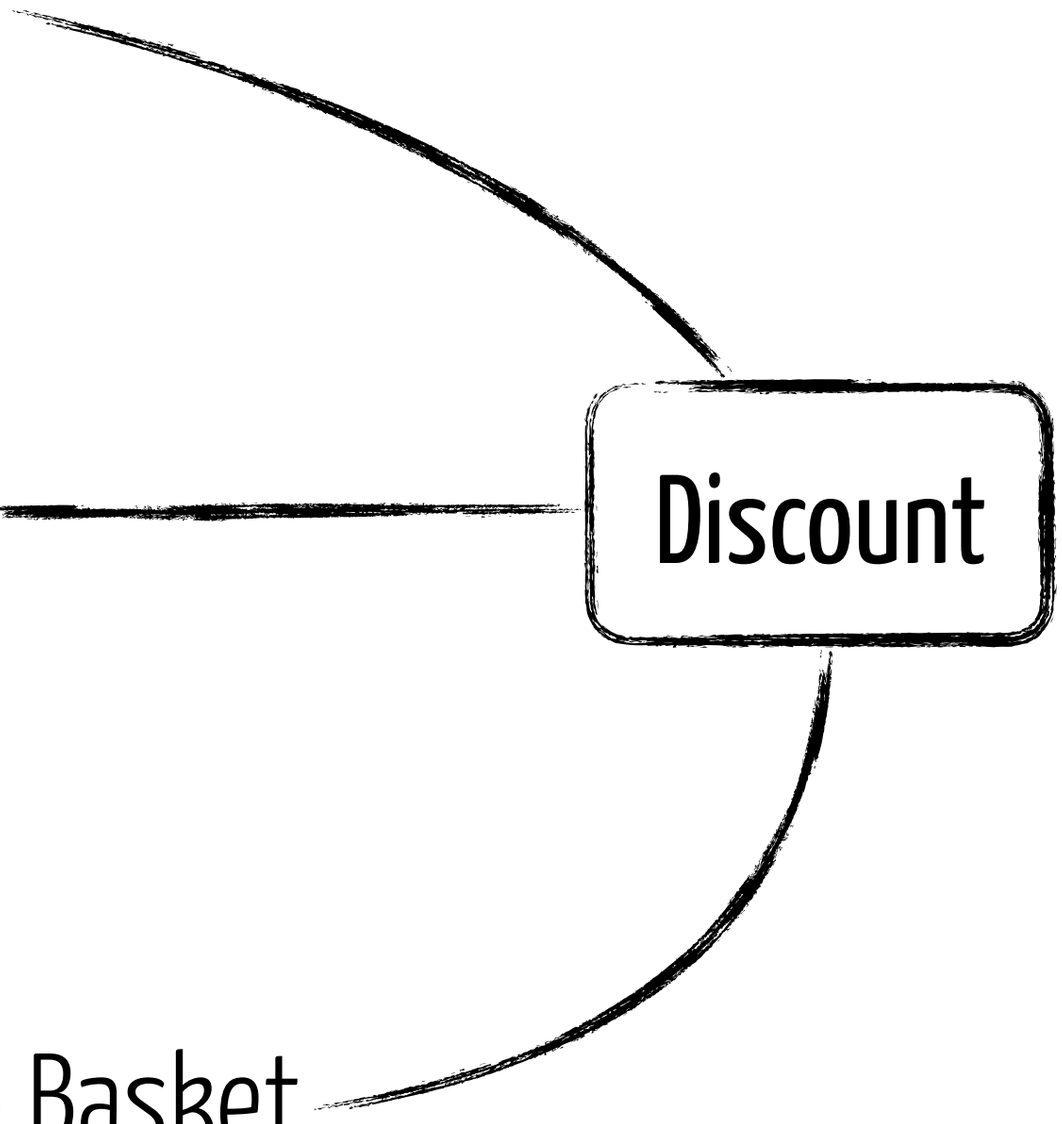
```java
@Test
public void cloneBooks() {
    Basket basket = new Basket(BOOK1, BOOK2, BOOK2);
    Basket clone = basket.duplicate();
    clone.remove(BOOK2);
    assertEquals(2, basket.howMany(BOOK2));
    assertEquals(1, clone.howMany(BOOK2));
}
```

The **Discount** case

apply(Basket b) -> Double

canBeApplied(Basket b)?

removePayed(Basket b) -> Basket



```java
public class  Discount {

    private int nbBooks;
    private double percentage;

    public Discount(int nbBooks, double percentage) {...}

    public boolean canBeApplied(Basket b) { return b.howManyDifferent() >= nbBooks; }

    public double apply(Basket b) { return Cashier.PRICE * nbBooks * percentage; }

    public Basket removePayedBooks(Basket b) {
        Map<Book, Integer> data = contents(b);
        Book[] consumed =
                data.entrySet().stream()
                        .sorted((e1, e2) -> Integer.compare(e2.getValue(), e1.getValue()))
                        .map(Map.Entry::getKey)
                        .collect(Collectors.toList()).subList(0, nbBooks).toArray(new Book[]{});
        Basket result = b.duplicate();
        result.remove(consumed);
        return result;
    }

    private Map<Book, Integer> contents(Basket b) {
        Map<Book, Integer> data = new HashMap<>();
        for(Book book: Book.values())
            data.put(book, b.howMany(book));
        return data;
    }
}
```

Testing the **Discount** concept

```
private Basket empty  = new Basket();
private Basket school = new Basket(BOOK1, BOOK1, BOOK1, BOOK1, BOOK1);
private Basket series = new Basket(BOOK1, BOOK2, BOOK3, BOOK4, BOOK5);
private Basket allin  = new Basket(BOOK1, BOOK1, BOOK2, BOOK2, BOOK3, BOOK3, BOOK4, BOOK5);

@Test
public void testDiscountMatch() {...}

@Test
public void testDiscountApplication() {...}

@Test
public void testEdgeCase() {
    Discount d = new Discount(4, 0.80);
    Basket remaining = d.removePayedBooks(allin);
    assertEquals(8, allin.howManyBooks());
    assertEquals(4, remaining.howManyBooks());
    assertEquals(4, remaining.howManyDifferent());
    assertEquals(1, remaining.howMany(BOOK1));
    assertEquals(1, remaining.howMany(BOOK2));
    assertEquals(1, remaining.howMany(BOOK3));
    assertTrue(remaining.howMany(BOOK4) == 1 || remaining.howMany(BOOK5) == 1);
}
```

```java
public static final double PRICE = 8.0;

private List<Discount> discounts =
        Arrays.asList(new Discount(5, 0.75), new Discount(4, 0.80),
                      new Discount(3, 0.90), new Discount(2, 0.95) );

public double price(Basket b) {
    if(b == null) { return 0.0; }
    return compute(b);
}

private double compute(Basket b) {
    if(b.isEmpty()) { return 0.0; }

    List<Discount> availables = findAvailableDiscount(b);
    if(availables.isEmpty()) {
        return PRICE * b.howManyBooks();
    } else {
        return availables.stream().map(d -> compute(b, d)).min(Double::compare).get();
    }
}

private double compute(Basket b, Discount d) {
    double local = d.apply(b);
    Basket remaining = d.removePayedBooks(b);
    return local + compute(remaining);
}

private List<Discount> findAvailableDiscount(Basket b) {
    return discounts.stream().filter(d -> d.canBeApplied(b)).collect(Collectors.toList());
}
```

Cashier

```java
private double compute(Basket b) {
    if(b.isEmpty()) { return 0.0; }

    List<Discount> availables = findAvailableDiscount(b);
    if(availables.isEmpty()) {
        return PRICE * b.howManyBooks();
    } else {
        return availables.stream().map(d -> compute(b, d)).min(Double::compare).get();
    }
}
```

b is empty => **0.0**

Let **discounts** = {**d** ∈ Discount | **d** can be applied to **b**}

discounts is empty => $|\textbf{books}|_b$ * **PRICE**

Let **candidates** = { **p** ∈ $\mathbb{R}$ | **d** ∈ **discounts**, **p** = compute(**b**,**d**) }

=> **min(candidates)**

Let **local price** = **d** applied to **b**

Let **remaining** = clone of **b** without the books used in **d**

=> **local price** + compute(**remaining**)

```java
private double compute(Basket b, Discount d) {
    double local = d.apply(b);
    Basket remaining = d.removePayedBooks(b);
    return local + compute(remaining);
}
```

Testing the cashier

```
@Test
public void edgeCase() {
    Basket b = new Basket(BOOK1, BOOK1, BOOK2, BOOK2, BOOK3, BOOK3, BOOK4, BOOK5);
    assertEquals(2 * (4 * PRICE * 0.8),
            cashier.price(b), 0.01);

    Basket allin = new Basket(
            BOOK1, BOOK1, BOOK1, BOOK1, BOOK1,
            BOOK2, BOOK2, BOOK2, BOOK2, BOOK2,
            BOOK3, BOOK3, BOOK3, BOOK3,
            BOOK4, BOOK4, BOOK4, BOOK4, BOOK4,
            BOOK5, BOOK5, BOOK5, BOOK5 );
    assertEquals(3 * (5 * PRICE * 0.75) + 2 * (4 * PRICE * 0.8), cashier.price(allin), 0.01);
}
```

Let the battle begin!

Benchmarking

Benchmark Configuration

```java
@BenchmarkMode(Mode.AverageTime)       // Production configuration (avg time in microseconds)
@OutputTimeUnit(TimeUnit.MILLISECONDS)
public class PotterBenchmark {

    @Benchmark
    public void ooBasics() {
        Cashier cashier = new Cashier();
        cashier.price(new Basket(BOOK1)); cashier.price(new Basket(BOOK2));
        cashier.price(new Basket(BOOK3)); cashier.price(new Basket(BOOK4));
        cashier.price(new Basket(BOOK5));
    }

    @Benchmark
    public void rawBasics() {
        PotterDiscountCalculator calculator = new PotterDiscountCalculator();
        calculator.calculatePrice(Arrays.asList(1));
    }
```

http://blog.soat.fr/2015/07/benchmark-java-introduction-a-jmh/

http://blog.soat.fr/2015/07/benchmark-java-jmh-fine-tuning/

Each bench is executed 10 times

20 executions to warm up the JVM

```
# Run progress: 41.67% complete, ETA 00:23:44
# Fork: 6 of 10
# Warmup Iteration   1: 43.680 ms/op
…
# Warmup Iteration  20: 27.823 ms/op
Iteration   1: 28.076 ms/op
…
Iteration  20: 27.848 ms/op
```

20 executions to collect measurements

Aggregated results

```
Benchmark                    Mode  Cnt   Score    Error  Units
PotterBenchmark.ooEdge       avgt  200  29.504 ±  0.487  ms/op
```

Response time Measurement

1. A response time of 100ms is perceived as instantaneous.
2. Response times of 1 second or less are fast enough for users to feel they are interacting freely with the information.
3. Response times greater than 10 seconds completely lose the user's attention.

Miller's laws [1968]

https://www.pubnub.com/blog/2015-02-09-how-fast-is-realtime-human-perception-and-technology/

```
# Run complete. Total time: 00:40:41

Benchmark                        Mode  Cnt    Score    Error  Units
PotterBenchmark.ooBasics         avgt  200    0.008 ±  0.001  ms/op
PotterBenchmark.ooDiscount       avgt  200    0.012 ±  0.001  ms/op
PotterBenchmark.ooEdge           avgt  200   29.504 ±  0.487  ms/op
PotterBenchmark.rawBasics        avgt  200  ≈ 10⁻⁴             ms/op
PotterBenchmark.rawDiscount      avgt  200  ≈ 10⁻⁴             ms/op
PotterBenchmark.rawEdge          avgt  200  ≈ 10⁻⁴             ms/op
```

So who's the winner ?