Tax-Calculator

Berechnung von Steuern. Hierzu werden Parameter, nämlich der zu besteuernde Betrag, das Land und die Art der Steuer angegeben (Mehrwertsteuer oder Kapitalertragssteuer auf Kursgewinne) und an den Server gesendet.
Die Steuersätze können fiktional sein und müssen nicht der Realität entsprechen. Auch wird es wahrscheinlich immer nur eine flat-tax-Besteuerung (linear) und keine progressive Besteuerung geben.

Frameworks und Implementierung

Der Service kann als RestAPI mit Java-Vert.x implementiert werden.
Die Response ist Text im JSON-Format. In ihm ist der Nettobetrag, der Steuersatz und der abgezogene Betrag enthalten.

Github-Repot: https://github.com/GeorgHs/taxcalculator

Commits-Verlauf: https://github.com/GeorgHs/taxcalculator/commits/main -> dann entsprechend auf Browse Files gehen.

Deployment: https://github.com/GeorgHs/taxcalculator/blob/main/Postman_requests.mp4

Zuerst MainVerticle starten.

Postman-Requests/Responses:

1. Post: http://localhost:1323/api/computations

Somit werden alle Länder-Entitäten erzeugt.

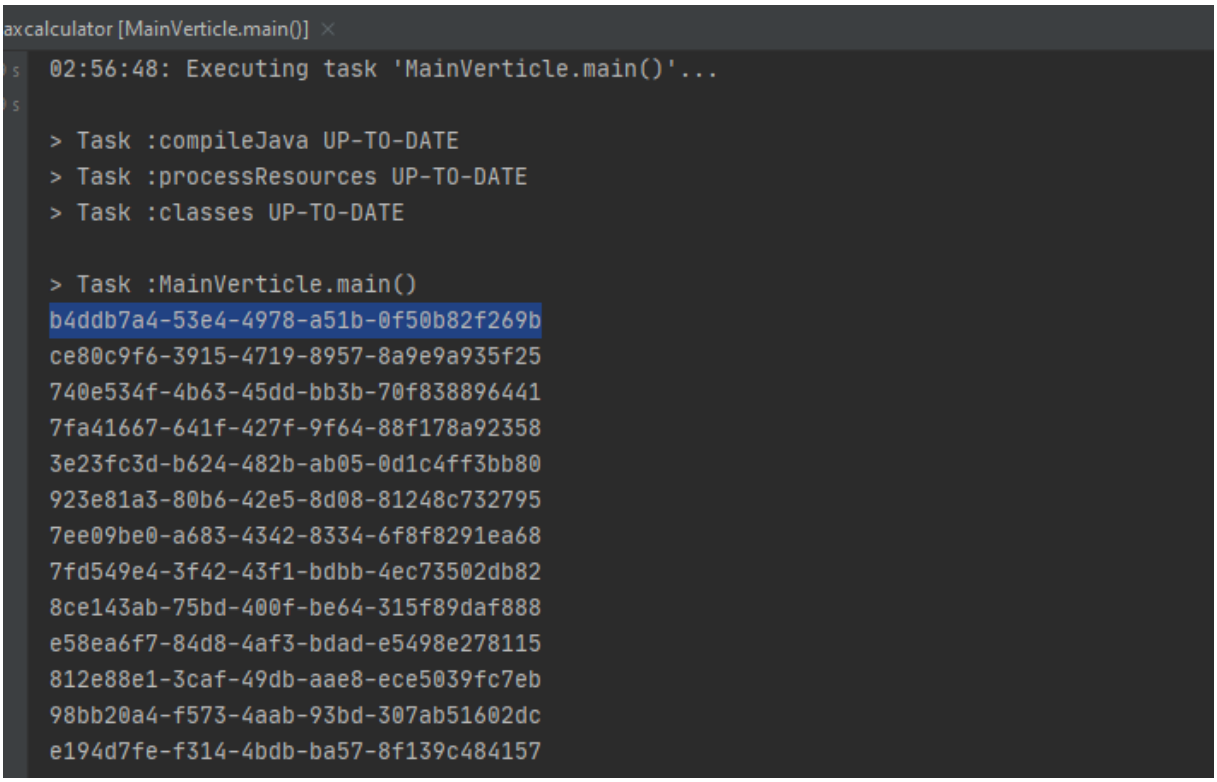
2. Get: http://localhost:1323/api/computations

Somit wird eine neue Berechnung als Entität erstellt. Die ID-Nummer ist als Eintrag zu finden: "id": "df910397-6e2e-4389-8021-b6dac2b76817".

3. Post: http://localhost:1323/api/computations/df910397-6e2e-4389-8021-b6dac2b76817/compute

In Body folgende Raw-Json einfügen. Die TaxRateID stammt von der erzeugten Country-Entität ab.

```
{
   "taxRateId":" b4ddb7a4-53e4-4978-a51b-0f50b82f269b",
   "grossAmount":10000.0
}
```



Ich habe leider keine IntelliJ-Ultimate Version, daher kann ich keine übersichtlichen Klassendiagramme erstellen. Mir stehen somit nur semi-professionelle UML-Diagram-Generator zur Verfügung bei denen ich vieles manuell anpassen musste.

Ubiquitous Language

Ein Domänenexperte ist der Steuerberater. Der Entwickler ist der Informatiker. Sie müssen beide auf einen gemeinsamen Nenner bei Begrifflichkeiten stoßen. Die Ubiquitous Language ist sowohl für den Domänenexperten als auch Entwickler verständlich. Es werden „Mehrdeutigkeiten und Unklarheiten beseitigt“.

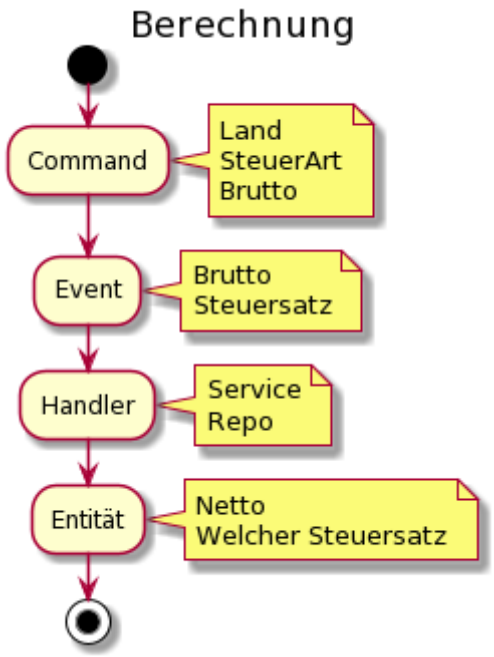
Schlüsselwörter wie sie in der Ubiquitous Language zu verwenden wären, sind:

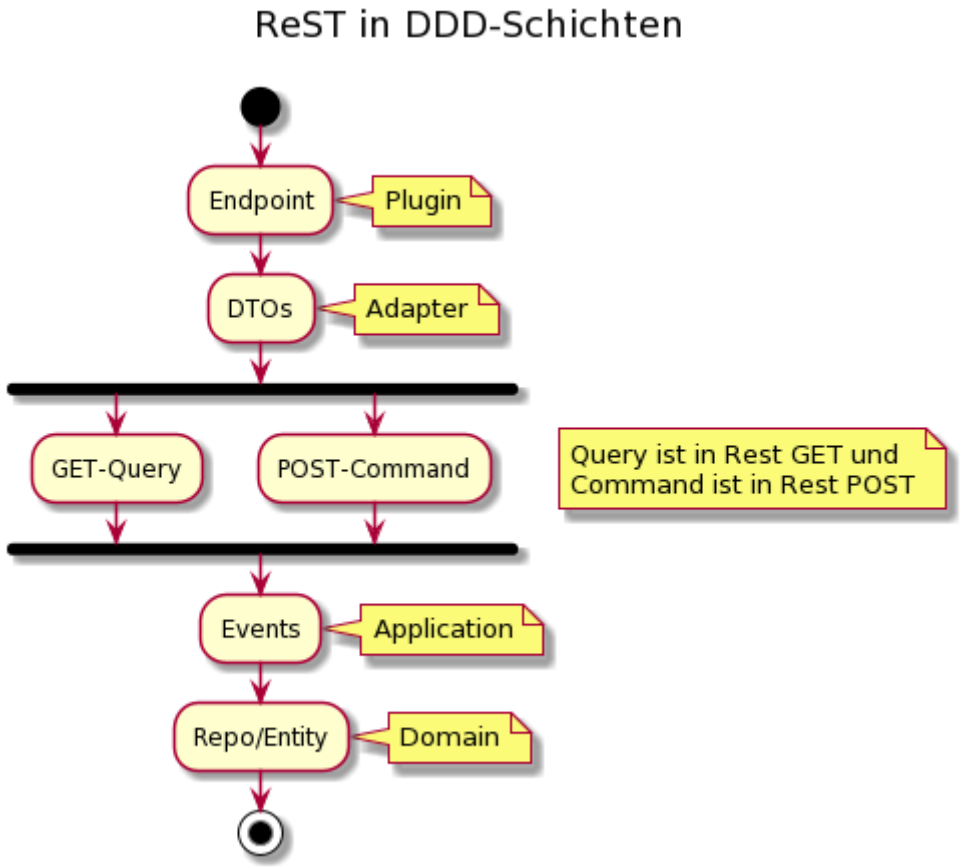
Schlüsselwort	Beschreibung
Country	Je nach Land unterscheidet sich die zu berechnende Steuer
Gross-Amount	Der Bruttobetrag ist der noch zu berechnende Betrag, der zu Beginn gewählt wird.
Net-Amount	Der Nettobetrag ist der am Ende berechnete Betrag nach Lohnkostenabzügen und Berechnung der Steuer.
Tax-Type	Die Steuerart – das kann Mehrwertsteuer oder Kapitalertragssteuer sein. Die Steuerart beeinflusst die Tax-Rate.
Tax-Rate	Die Tax-Rate wird aus Country und Tax-Type geschlussfolgert.

Um die Steuer berechnen zu können, muss der Bruttobetrag und die Tax-Rate bekannt sein. Die Tax-Rate ergibt sich aus Tax-Typ und Country. Beispielsweise bei Bulgarien und Mehrwertsteuer ergeben sich 20%. Bei Griechenland und Kapitalertragsteuer ergeben sich 15%.

Am Ende steht der Nettobetrag – sprich Net-Amount.

Clean Architecture und Modellbildung





Hieraus ergibt sich auch die Clean Architecture.

Tech-Stack: Vert.x.

Dependency Management: Gradle.

Daraus resultierendes DDD

Der Einsatzbereich und die Problemlösung dieser Software ist das Berechnen von Steuern. Ein Bruttobetrag, ein Land und die Art der Steuer sind ausschlaggebend für den Nettobetrag am Ende.

Wichtig für diesen konkreten Anwendungsfall ist, dass die Rest-Architektur und die Package-Bezeichnungen mit denen des Domain Driven Designs in Zusammenhang gebracht werden müssen. In den Plugins liegen beispielsweise die InMemory-Datenbank und die Endpoints. In der Adapter-Schicht die DTOs mit ihren Mappern.

Value Objects (VO)

Value Objects	**Beschreibung**
Country	Listet als Enum alle Länder auf
GrossAmount	Der Bruttobetrag kann nach Eingabe nicht mehr verändert werden.
NetAmount	Der Nettobetrag ergibt sich aus berechnetem Bruttobetrag
TaxType	In Form eines Enums werden zwei unterschiedliche Steuertypen aufgelistet.

Die Länder und der Tax-Typ sind Enums und damit vollständig unveränderlich – das ist ein klassischer Anwendungsfall für ein Value Object. Der GrossAmount und NetAmount speichern float-Werte, die nach Instanziierung nicht mehr verändert werden können. Es sind keine setter-Methoden auffindbar. Country, GrossAmount, NetAmount und TaxType liefern nur unveränderliche Rückgabewerte und defensive Kopien.

```
1 package com.ghertzsch.taxcalculator.domain.valueobjects;
2
3 public class GrossAmount {
4
5   private final float value;
6
7   public GrossAmount(float value) {
8     this.value = value;
9   }
10
11  public float getValue() {
12    return value;
13  }
14
15 }
```

```
1 package com.ghertzsch.taxcalculator.domain.valueobjects;
2
```

```java
public enum Country {
  BELGIUM,
  BULGARIA,
  DENMARK,
  GERMANY,
  ESTONIA,
  FINLAND,
  FRANCE,
  GREECE,
  IRELAND,
  ITALY,
  CROATIA,
  LATVIA,
  LITHUANIA,
  LUXEMBURG,
  MALTA,
  NETHERLANDS,
  NORTHERN_IRELAND,
  AUSTRIA,
  POLAND,
  PORTUGAL,
  ROMANIA,
  SWEDEN,
  SLOVAKIA,
  SLOVENIA,
  SPAIN,
  CZECH_REPUBLIC,
  HUNGARY,
  CYPRUS,
}
```

```java
package com.ghertzsch.taxcalculator.domain.valueobjects;

public class NetAmount {

  private final float value;

  public NetAmount(float value) {
    this.value = value;
  }

  public float getValue() {
    return value;
  }

}
```

```java
package com.ghertzsch.taxcalculator.domain.valueobjects;

public enum TaxType {
  VALUE_ADDED_TAX,
  CAPITAL_GAINS_TAX,
}
```

Entities & Aggregates

Entities	Beschreibung
NetAmountComputation	In NetAmountComputation ist am Ende festgelegt, wie die gesamte Rechnung durchgeführt werden soll.
TaxRate	In TaxRate ist festgelegt, wie es unter Beachtung der Faktoren Country und TaxType zu dem jeweiligen Steuersatz kommt.

NetAmountComputation und TaxRate haben so viel wie möglich an die Value Objects ausgelagert. Bei NetAmountComputation ist das NetAmount und GrossAmount. Bei Tax-Rate sind die genutzten Value Objects Country und Tax-Type. Eine Entität ist modifizierbar und verändert sich während der Lebenszeit. Beide Entitäten sind außerdem mit einem Surrogatschlüssel, der UUID versehen. Eine UUID ist sehr aussagekräftig und es besteht keine Gefahr darin, dass die Klasse doppelt verwendet werden könnte. Die ID wird aus den Eigenschaften der Entität zusammengesetzt. In diesem Fall wurde sich für die UUID als String entschieden und gegen die UUID als ValueObject oder Converter/CustomUserType.

NetAmountComputation:

```
 1 package com.ghertzsch.taxcalculator.domain.entities;
 2
 3 import com.ghertzsch.taxcalculator.domain.exceptions.DomainException;
 4 import com.ghertzsch.taxcalculator.domain.valueobjects.GrossAmount;
 5 import com.ghertzsch.taxcalculator.domain.valueobjects.NetAmount;
 6
 7 import java.util.UUID;
 8
 9 public class NetAmountComputation {
10
11   private final UUID id;
12
13   private NetAmount netAmount = new NetAmount(0);
14
15   private GrossAmount grossAmount = new GrossAmount(0);
16
17   private TaxRate taxRate;
18
19   public NetAmountComputation(UUID id) {
20     this.id = id;
21   }
22
23   public UUID getId() {
24     return id;
25   }
26
27   public NetAmount getNetAmount() {
28     return netAmount;
29   }
30
31   public void setNetAmount(NetAmount netAmount) throws DomainException {
32     if (netAmount.getValue() < 0) {
33       throw new DomainException("Value of netAmount can not be
34 negative");
35     }
```

```
36      this.netAmount = netAmount;
37    }
38
39    public GrossAmount getGrossAmount() {
40      return grossAmount;
41    }
42
43    public void setGrossAmount(GrossAmount grossAmount) throws
44 DomainException {
45      if (grossAmount.getValue() < 0) {
46        throw new DomainException("Value of grossAmount can not be
47 negative");
48      }
49      this.grossAmount = grossAmount;
50    }
51
52    public TaxRate getTaxRate() {
53      return taxRate;
54    }
55
56    public void setTaxRate(TaxRate taxRate) {
        this.taxRate = taxRate;
      }
    }
```

TaxRate:

```
 1 package com.ghertzsch.taxcalculator.domain.entities;
 2
 3 import com.ghertzsch.taxcalculator.domain.valueobjects.Country;
 4 import com.ghertzsch.taxcalculator.domain.valueobjects.TaxType;
 5
 6 import java.util.UUID;
 7
 8 public class TaxRate {
 9
10   private final UUID id = UUID.randomUUID();
11
12   private final Country country;
13
14   private final TaxType taxType;
15
16   private final float value;
17
18   public TaxRate(Country country, TaxType taxType, float value) {
19     this.country = country;
20     this.taxType = taxType;
21     this.value = value;
22   }
23
24   public UUID getId() {
25     return id;
26   }
27
28   public Country getCountry() {
29     return country;
30   }
31
```

```
32   public TaxType getTaxType() {
33     return taxType;
34   }
35
36   public float getValue() {
37     return value;
38   }
39
40 }
```

Ein Aggregate gruppiert die Entitäten und VOs zu gemeinsam verwalteten Einheiten. Jede Entität gehört zu einem Aggregat, selbst wenn das Aggregat nur aus dieser Entität besteht. Die NetAmountComputation ist eine Aggregate Root Entity. Die NetAmountComputation nutzt die Entität und Aggregate Tax-Rate und die beiden Value Objects Net Amount und Gross Amount. Die TaxRate nutzt nur die Value Objects Country und Tax-Type.

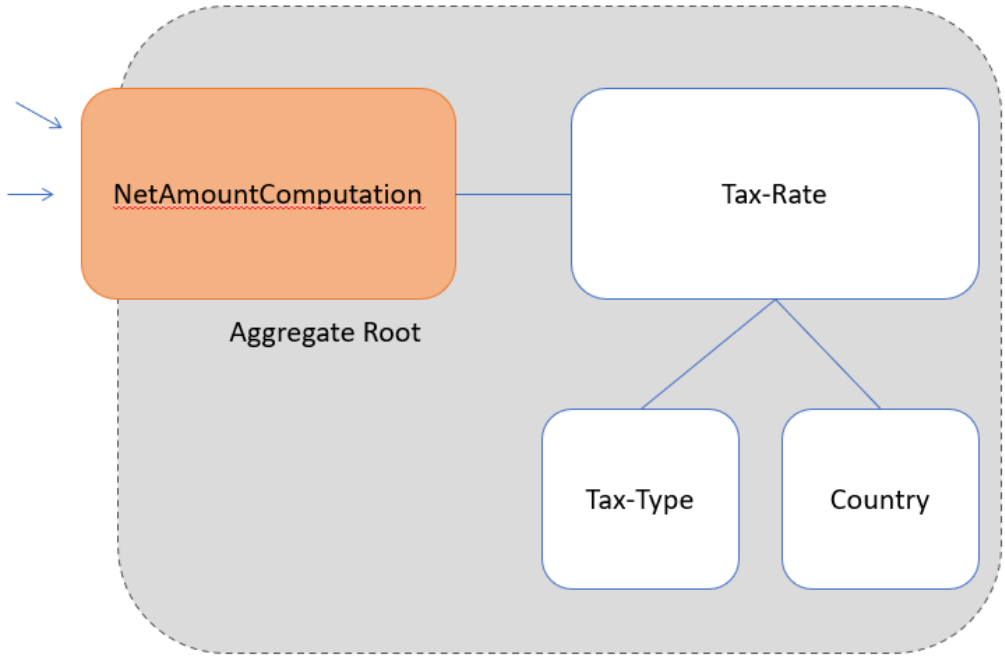


Abbildung 1 Zusammenspiel AR NetAmountComputation und Tax-Rate

Repositories

Respositories bieten dem Domain Code Zugriff auf persistenten Speicher. Ein Repository vermittelt zwischen Domäne und Datenmodell. Der konkrete technische Zugriff wird vom Repository verborgen. Die Anti-Corruption-Layer wird zur Persistenzschicht. NetAmountComputationRepository ist ein Interface, dass im Datenbank-Plugin die InMemory-Datenbank InMemoryNetAmountComputation implementiert. Je Aggregate existiert typischerweise ein Repository. NetAmountComputation ist das Aggregate, welches zum NetAmountComputationRepository gehört.

TaxRate:

```
1 package com.ghertzsch.taxcalculator.domain.repositories;
2
```

```
3 import com.ghertzsch.taxcalculator.domain.entities.TaxRate;
4
5 import java.util.List;
6 import java.util.UUID;
7
8 public interface TaxRateRepository {
9
10   void storeTaxRate(TaxRate taxRate);
11
12   TaxRate findTaxRateById(UUID id);
13
14   List<TaxRate> findAllTaxRates();
15
16 }
```

NetAmountComputationRepository:

```
1 package com.ghertzsch.taxcalculator.domain.repositories;
2
3 import com.ghertzsch.taxcalculator.domain.entities.NetAmountComputation;
4
5 import java.util.List;
6 import java.util.UUID;
7
8 public interface NetAmountComputationRepository {
9
10   void storeComputation(NetAmountComputation netAmountComputation);
11
12   List<NetAmountComputation> findAllComputations();
13
14   NetAmountComputation findComputationById(UUID id);
15
16 }
```

Das NetAmountComputationRepository wird vererbt von der InMemoryDatenbank InMemoryNetAmountComputation und genutzt. Die InMemory-Datenbank speichert in der Plugin-Schicht die einzelnen Computations listenartig ab.

```
1 package com.ghertzsch.taxcalculator.plugins.repositories;
2
3 import com.ghertzsch.taxcalculator.domain.
    entities.NetAmountComputation;
4 import com.ghertzsch.taxcalculator.domain.
    repositories.NetAmountComputationRepository;
5
6 import java.util.ArrayList;
7 import java.util.List;
8 import java.util.UUID;
9
10 public class InMemoryNetAmountComputation implements
11   NetAmountComputationRepository {
12
13   private final ArrayList<NetAmountComputation> netAmountComputations
14     = new ArrayList<>();
15
16   @Override
17   public void storeComputation(NetAmountComputation
18 netAmountComputation) {
19     var netAmountComputationIndex =
```

```
20        netAmountComputations.indexOf(netAmountComputation);
21      if (netAmountComputationIndex != -1) {
22        netAmountComputations.set(netAmountComputationIndex,
23          netAmountComputation);
24        return;
25      }
26      netAmountComputations.add(netAmountComputation);
27    }
28
29    @Override
30    public List<NetAmountComputation> findAllComputations() {
31      return netAmountComputations;
32    }
33
34    @Override
35    public NetAmountComputation findComputationById(UUID id) {
36      return netAmountComputations.stream()
37        .filter(netAmountComputation ->
38 netAmountComputation.getId().equals(id))
39        .findFirst()
40        .orElse(null);
41    }
42 }
```

Domain Services & Events

Ein Domain Service ist ein „Helfer“ innerhalb des Domänenmodells. Wenn Verhalten nicht eindeutig einer Entität oder Prozess zugeordnet werden kann, ist der Domain Service notwendig.

Da in meinem konkreten Beispiel aber das Event einer konkreten Entität zugeordnet werden kann, halte ich es für sinnvoll, dafür ein Event zu schreiben als einen gesamten Service. Auch arbeite ich bei Vert.X mit einem Event-Driven Microservice Framework, bei denen Events deutlich besser ins Schema passen als ein Service (anders als bei Spring).

Zwar nicht implementiert aber rein hypothetisch würde ein Service in seinen Grundzügen so oder so in etwa aussehen:

15 lines (10 sloc) | 592 Bytes

```
1   package com.ghertzsch.taxcalculator.application.services;
2
3
4   import com.ghertzsch.taxcalculator.domain.entities.NetAmountComputation;
5   import com.ghertzsch.taxcalculator.domain.repositories.NetAmountComputationRepository;
6
7   public class NetAmountComputationService {
8       private final NetAmountComputationRepository netAmountComputationRepository;
9       private NetAmountComputation netAmountComputation;
10
11
12      public NetAmountComputationService(NetAmountComputationRepository netAmountComputationRepository) {
13        this.netAmountComputationRepository = netAmountComputationRepository;
14      }
15    }
```

Abbildung 2 beispielshaft ein implementierter Service in der Application

10 lines (7 sloc) | 322 Bytes

```java
package com.ghertzsch.taxcalculator.domain.services;

import com.ghertzsch.taxcalculator.domain.entities.NetAmountComputation;

import java.util.UUID;

public interface NetAmountComputationDomainService  {
  void initializeService(String netAmountComputationId);
  NetAmountComputation getCurrentNetAmountComputation();
}
```

Abbildung 3 beispielhaft ein Service-Interface in der Domain

Events in der Domain sind sinnvoller:

```java
package com.ghertzsch.taxcalculator.domain.events;

import java.util.UUID;

public class Event {

  private final UUID id = UUID.randomUUID();

  public UUID getId() {
    return id;
  }

}
```

Im Rahmen des CQRS-Pattern (https://microservices.io/patterns/data/cqrs.html) muss später für die Plugin-Schicht für den Endpoint eine Prepared-Klasse vorhanden sein.

```java
package com.ghertzsch.taxcalculator.domain.events;

import java.util.UUID;

public class NetAmountComputationPrepared extends Event {

  private final UUID netAmountComputationId = UUID.randomUUID();

  public UUID getNetAmountComputationId() {
    return netAmountComputationId;
  }

}
```

Das eigentliche Event ist aber NetAmountComputed, die direkt der Entität NetAmountComputation zugeordnet werden kann:

```java
package com.ghertzsch.taxcalculator.domain.events;

```

```java
import com.ghertzsch.taxcalculator.domain.entities.TaxRate;
import com.ghertzsch.taxcalculator.domain.valueobjects.GrossAmount;

import java.util.UUID;

public class NetAmountComputed extends Event {

  private final UUID netAmountComputationId;

  private final TaxRate taxRate;

  private final GrossAmount grossAmount;

  public NetAmountComputed(UUID netAmountComputationId,
                           TaxRate taxRate,
                           GrossAmount grossAmount) {
    this.netAmountComputationId = netAmountComputationId;
    this.taxRate = taxRate;
    this.grossAmount = grossAmount;
  }

  public UUID getNetAmountComputationId() {
    return netAmountComputationId;
  }

  public TaxRate getTaxRate() {
    return taxRate;
  }

  public GrossAmount getGrossAmount() {
    return grossAmount;
  }

}
```

Programming Principles

Für die Veranschaulichung der Principles eignen sich besonders Code- und UML-Beispiele. Durch das Studieren dieser Code- und UML-Beispiele lassen sich Muster schnell erkennen. Die theoretischen Erklärungen dienen zur Ergänzung.

SOLID

Single Responsibility

“Eine Klasse sollte nur eine Ursache oder einen Grund haben sich zu ändern”. Das Single Responsibility Pattern zielt auf niedrige Komplexität und Kopplung ab. Und auch der Grundsatz, „jede Klasse sollte nur eine Zuständigkeit haben“ geht gegen Komplexität.

Ohne Single Responsibility Pattern hätte GrossAmount und NetAmount in einer Klasse zusammengefasst werden können. Weil wir uns aber an das Single Responsibility Pattern halten, werden GrossAmount und NetAmount getrennt aufgefasst.

```java
package com.ghertzsch.taxcalculator.domain.valueobjects;

public class GrossAmount {

  private final float value;
```

```
6
7   public GrossAmount(float value) {
8     this.value = value;
9   }
10
11  public float getValue() {
12    return value;
13  }
14
15 }
```

```
1 package com.ghertzsch.taxcalculator.domain.valueobjects;
2
3 public class NetAmount {
4
5   private final float value;
6
7   public NetAmount(float value) {
8     this.value = value;
9   }
10
11  public float getValue() {
12    return value;
13  }
14
15 }
```

Open Closed Principle

"Offen für Erweiterungen und geschlossen für Änderungen sein." Bestehender Code soll nicht verändert werden. Wenn man Code ändern will, dann durch Ableitungen, nicht durch Veränderung der eigentlichen Klasse. So können Bugs verhindert werden.

```
1 package com.ghertzsch.taxcalculator.domain.events;
2
3 import java.util.UUID;
4
5 public class Event {
6
7   private final UUID id = UUID.randomUUID();
8
9   public UUID getId() {
10    return id;
11  }
12
13 }
```

Das Event kann am Ende wie folgt genutzt werden:

```
1 package com.ghertzsch.taxcalculator.domain.events;
2
3 import java.util.UUID;
4
5 public class NetAmountComputationPrepared extends Event {
6
7   private final UUID netAmountComputationId = UUID.randomUUID();
8
```

```
 9   public UUID getNetAmountComputationId() {
10     return netAmountComputationId;
11   }
12
13 }
```

Liskov Substitution Principle

Es darf nicht von unerwartetem Verhalten überrascht werden. Die abgeleitete Klasse wirft keine andere oder neue Exception gegenüber der alten Superklasse. Verarbeitung innerhalb der Methode muss sich genauso verhalten. Es kann ein anderes Ergebnis entstehen, nicht aber eine andere Exception. Ein Beispiel hierfür ist die Domain Exception die dann in diversen Methoden Anwendung findet.

```
1 package com.ghertzsch.taxcalculator.domain.exceptions;
2
3 public class DomainException extends Exception {
4
5   public DomainException(String reason) {
6     super(reason);
7   }
8
9 }
```

```
 1 package com.ghertzsch.taxcalculator.domain.entities;
 2
 3 import com.ghertzsch.taxcalculator.domain.exceptions.DomainException;
 4 import com.ghertzsch.taxcalculator.domain.valueobjects.GrossAmount;
 5 import com.ghertzsch.taxcalculator.domain.valueobjects.NetAmount;
 6
 7 import java.util.UUID;
 8
 9 public class NetAmountComputation {
10
11   private final UUID id;
12
13   private NetAmount netAmount = new NetAmount(0);
14
15   private GrossAmount grossAmount = new GrossAmount(0);
16
17   private TaxRate taxRate;
18
19   public NetAmountComputation(UUID id) {
20     this.id = id;
21   }
22
23   public UUID getId() {
24     return id;
25   }
26
27   public NetAmount getNetAmount() {
28     return netAmount;
29   }
30
31   public void setNetAmount(NetAmount netAmount) throws DomainException {
32     if (netAmount.getValue() < 0) {
```

```
33        throw new DomainException("Value of netAmount can not be
34 negative");
35      }
36      this.netAmount = netAmount;
37    }
38
39    public GrossAmount getGrossAmount() {
40      return grossAmount;
41    }
42
43    public void setGrossAmount(GrossAmount grossAmount) throws
44 DomainException {
45      if (grossAmount.getValue() < 0) {
46        throw new DomainException("Value of grossAmount can not be
47 negative");
48      }
49      this.grossAmount = grossAmount;
50    }
51
52    public TaxRate getTaxRate() {
53      return taxRate;
54    }
55
56    public void setTaxRate(TaxRate taxRate) {
        this.taxRate = taxRate;
      }
    }
```

Interface Segregation Principle

Fat Interfaces sollen vermieden werden. Die Interfaces müssen passend zu den Anwendern gestaltet werden.

```
1  package com.ghertzsch.taxcalculator.domain.repositories;
 2
 3 import com.ghertzsch.taxcalculator.domain.entities.TaxRate;
 4
 5 import java.util.List;
 6 import java.util.UUID;
 7
 8 public interface TaxRateRepository {
 9
10   void storeTaxRate(TaxRate taxRate);
11
12   TaxRate findTaxRateById(UUID id);
13
14   List<TaxRate> findAllTaxRates();
15
16 }
```

```
1  package com.ghertzsch.taxcalculator.domain.repositories;
 2
 3 import com.ghertzsch.taxcalculator.domain.entities.NetAmountComputation;
 4
 5 import java.util.List;
 6 import java.util.UUID;
```

```
 7
 8 public interface NetAmountComputationRepository {
 9
10   void storeComputation(NetAmountComputation netAmountComputation);
11
12   List<NetAmountComputation> findAllComputations();
13
14   NetAmountComputation findComputationById(UUID id);
15
16 }
```

Dependency Inversion Principle

"Instead of high-level modules depending on low-level modules, both will depend on abstractions". High-Level Module sollten nicht von Low-Level Modulen abhängig sein. Beide sollten von Abstraktionen abhängen.

High Level Module	**Low-Level Module**
Geben Regeln vor	Implementierungen
Framework	
Sind Abstraktionen	Sind abhängig von Abstraktionen

Die TaxRate wird hierbei injected über die Methode storeTaxRate()

```
package com.ghertzsch.taxcalculator.domain.entities;

import com.ghertzsch.taxcalculator.domain.valueobjects.Country;
import com.ghertzsch.taxcalculator.domain.valueobjects.TaxType;

import java.util.UUID;

public class TaxRate {

  private final UUID id = UUID.randomUUID();

  private final Country country;

  private final TaxType taxType;

  private final float value;

  public TaxRate(Country country, TaxType taxType, float value) {
    this.country = country;
    this.taxType = taxType;
    this.value = value;
  }

  public UUID getId() {
    return id;
  }

  public Country getCountry() {
    return country;
  }

  public TaxType getTaxType() {
    return taxType;
  }
```

```
  public float getValue() {
    return value;
  }

}
```

```
package com.ghertzsch.taxcalculator.plugins.repositories;

import com.ghertzsch.taxcalculator.domain.entities.TaxRate;
import com.ghertzsch.taxcalculator.domain.repositories.TaxRateRepository;

import java.util.ArrayList;
import java.util.List;
import java.util.UUID;

public class InMemoryTaxRate implements TaxRateRepository {

  private final ArrayList<TaxRate> taxRates = new ArrayList<>();

  @Override
  public void storeTaxRate(TaxRate taxRate) {
    taxRates.add(taxRate);
  }

  @Override
  public TaxRate findTaxRateById(UUID id) {
    return taxRates.stream()
      .filter(taxRate -> taxRate.getId().equals(id))
      .findFirst()
      .orElse(null);
  }

  @Override
  public List<TaxRate> findAllTaxRates() {
    return taxRates;
  }

}
```

GRASP

Low Coupling

+ nähere Beschreibung

Kopplung ist ein Maß für die Abhängigkeit zwischen den Objekten. Vorteile sind verständlicherer Code, bessere Wiederverwendbarkeit und schnelleres Testen. In der Vorlesung Programming Principles wurde von verschiedenen Formen des Couplings gesprochen:

X implementiert Interface Y
X ist abgeleitet von Klasse Y (auch indirekt)
X hat ein Attribut vom Typ Y
X hat eine Methode mit Referenz zu Klasse Y
X verwendet eine statische Methode von Klasse Y
X verwendet eine polymorphe Methode von Klasse oder Interface Y

Neben Objekten kann auch eine Verbindung zu und zwischen abstrakten Datentypen, Threads und Resourcen (Dateien, Speicher und CPU) bestehen. Ein ADT ist in Java z.B.:

```
1 public interface IStack<E> {
2
3     public boolean isStackEmpty();
4     public IStack<E> push(E elem);
5     public IStack<E> pop();
6     public E top();
7 }
```

Durch geringe Kopplung kommt es zu einer geringeren Abhängigkeit in anderen Teilen.

High Cohesion

In einer Klasse müssen die Elemente inhaltlich zusammenpassen. Die hohe Kohäsion ergänzt sich zur losen Kopplung. Somit werden die Komponenten wiederverwendbarer und das Design verständlicher.

Hängt mit Single Responsibility Pattern zusammen – pro Klasse immer nur eine Funktionalität.

```
package com.ghertzsch.taxcalculator.domain.valueobjects;

public class GrossAmount {

  private final float value;

  public GrossAmount(float value) {
    this.value = value;
  }

  public float getValue() {
    return value;
  }

}
```

```
package com.ghertzsch.taxcalculator.domain.valueobjects;

public class NetAmount {

  private final float value;

  public NetAmount(float value) {
    this.value = value;
  }

  public float getValue() {
    return value;
  }

}
```

Oder auch bei der Generierung der Länder (die ich lieber im JSON-Format erzeugt hätte).

```
package com.ghertzsch.taxcalculator.plugins.Resources.CountryWithTax;

import com.ghertzsch.taxcalculator.domain.factories.TaxRateFactory;
import com.ghertzsch.taxcalculator.domain.repositories.TaxRateRepository;
import com.ghertzsch.taxcalculator.domain.valueobjects.Country;
import com.ghertzsch.taxcalculator.domain.valueobjects.TaxType;
```

```
public class Austria {
  public static void generate(TaxRateRepository taxRateRepository) {
    var vat = new TaxRateFactory()
      .OfType(TaxType.VALUE_ADDED_TAX)
      .WithCountry(Country.AUSTRIA)
      .WithValue(0.2f)
      .build();

    var capGains = new TaxRateFactory()
      .OfType(TaxType.CAPITAL_GAINS_TAX)
      .WithCountry(Country.AUSTRIA)
      .WithValue(0.275f)
      .build();

    taxRateRepository.storeTaxRate(vat);
    taxRateRepository.storeTaxRate(capGains);
  };
}
```

Polymorphismus

Polymorphie bedeutet, dass zwei Methoden denselben Methodennamen verwenden, aber die Implementierung der Methoden sich unterscheidet. Das findet besonders bei Vererbung Anwendung.

+ nähere Beschreibung

Information Expert

Der Information Expert wird verwendet, um zu entscheiden, wo Zuständigkeiten wie Methoden, Computed Fields, … hin delegiert werden sollen. Hierzu kommen nun zwei Möglichkeiten in Frage. Einem Objekt, dass die Informationen besitzt, die Verantwortung zu delegieren. Auf der anderen Seite im Designmodell eine passende Klasse zu suchen und diese zu verwenden. Andernfalls wird im Domänenmodell eine passende Klasse erstellt.

Creator

Dieses ist hier konkret Prepared. Hiermit wird die entsprechende Computation vorbereitet.

```
package com.ghertzsch.taxcalculator.plugins.endpoints;

import
com.ghertzsch.taxcalculator.application.commands.PrepareNetAmountComputatio
nHandler;
import
com.ghertzsch.taxcalculator.domain.commands.PrepareNetAmountComputation;
import
com.ghertzsch.taxcalculator.domain.repositories.NetAmountComputationReposit
ory;
import com.ghertzsch.taxcalculator.domain.repositories.TaxRateRepository;
import io.netty.handler.codec.http.HttpResponseStatus;
import io.vertx.core.Vertx;
import io.vertx.ext.web.Router;
import io.vertx.ext.web.RoutingContext;
import io.vertx.ext.web.handler.BodyHandler;

public class PrepareNetAmountComputationEndpoint {

  private final NetAmountComputationRepository
netAmountComputationRepository;
```

```
  public PrepareNetAmountComputationEndpoint(NetAmountComputationRepository
netAmountComputationRepository) {
    this.netAmountComputationRepository = netAmountComputationRepository;
  }

  public Router getRouter(Vertx vertx) {
    var router = Router.router(vertx);
    router.route("/computations*").handler(BodyHandler.create());
    router.post("/computations").handler(this::endpoint);

    return router;
  }

  private void endpoint(RoutingContext context) {
    var prepareNetAmountComputation = new PrepareNetAmountComputation();
    var prepareNetAmountComputationHandler = new
PrepareNetAmountComputationHandler(netAmountComputationRepository);

    prepareNetAmountComputationHandler.handle(prepareNetAmountComputation);


context.response().setStatusCode(HttpResponseStatus.ACCEPTED.code()).end();
  }

}
```

Indirection

Mit der Indirection werden Systeme voneinander entkoppelt. Die Indirektion bietet mehr Freiheitsgrade als Vererbung und Polymorphismus.

Controller

Der Controller verarbeitet die Benutzereingaben. In der Webentwicklung beispielsweise eine REST-API. Man unterscheidet zwischen System Controller und Use Case Controller. Der System Controller ist nur bei kleineren Anwendungen praktikabel. Hier gibt es insgesamt nur einen Controller für alle Aktionen. Beim Use Case Controller bestehen viele kleine Controller.

Sind in diesem konkreten Fall die Endpoints:

```
package com.ghertzsch.taxcalculator.plugins.endpoints;

import com.ghertzsch.taxcalculator.adapters.ComputeNetAmountDTO;
import
com.ghertzsch.taxcalculator.application.commands.ComputeNetAmountHandler;
import
com.ghertzsch.taxcalculator.application.commands.PrepareNetAmountComputatio
nHandler;
import com.ghertzsch.taxcalculator.domain.commands.ComputeNetAmount;
import
com.ghertzsch.taxcalculator.domain.commands.PrepareNetAmountComputation;
import
com.ghertzsch.taxcalculator.domain.repositories.NetAmountComputationReposit
ory;
import com.ghertzsch.taxcalculator.domain.repositories.TaxRateRepository;
import com.google.gson.Gson;
import io.netty.handler.codec.http.HttpResponseStatus;
import io.vertx.core.Vertx;
import io.vertx.ext.web.Router;
import io.vertx.ext.web.RoutingContext;
```

```
import io.vertx.ext.web.handler.BodyHandler;

public class ComputeNetAmountEndpoint {

  private final TaxRateRepository taxRateRepository;

  private final NetAmountComputationRepository
netAmountComputationRepository;

  public ComputeNetAmountEndpoint(TaxRateRepository taxRateRepository,
NetAmountComputationRepository netAmountComputationRepository) {
    this.taxRateRepository = taxRateRepository;
    this.netAmountComputationRepository = netAmountComputationRepository;
  }

  public Router getRouter(Vertx vertx) {
    var router = Router.router(vertx);
    router.route("/computations*").handler(BodyHandler.create());
    router.post("/computations/:id/compute").handler(this::endpoint);

    return router;
  }

  private void endpoint(RoutingContext context) {
    var request = context.request();
    var body = context.getBodyAsString();

    var dto = new Gson().fromJson(body, ComputeNetAmountDTO.class);
    dto.setNetAmountComputationId(request.getParam("id"));

    var computeNetAmount = new
ComputeNetAmount(dto.getNetAmountComputationId(), dto.getTaxRateId(),
dto.getGrossAmount());
    var computeNetAmountHandler = new ComputeNetAmountHandler(
      taxRateRepository,
      netAmountComputationRepository
    );

    computeNetAmountHandler.handle(computeNetAmount);


context.response().setStatusCode(HttpResponseStatus.ACCEPTED.code()).end();
  }

}
```

```
package com.ghertzsch.taxcalculator.plugins.endpoints;

import com.ghertzsch.taxcalculator.adapters.ListNetAmountComputationsDTO;
import com.ghertzsch.taxcalculator.adapters.NetAmountComputationMapper;
import
com.ghertzsch.taxcalculator.application.queries.ListNetAmountComputationsHa
ndler;
import
com.ghertzsch.taxcalculator.domain.queries.ListNetAmountComputations;
import
com.ghertzsch.taxcalculator.domain.repositories.NetAmountComputationReposit
ory;
import com.google.gson.Gson;
import com.google.gson.GsonBuilder;
import io.netty.handler.codec.http.HttpResponseStatus;
import io.vertx.core.Vertx;
```

```java
import io.vertx.ext.web.Router;
import io.vertx.ext.web.RoutingContext;
import io.vertx.ext.web.handler.BodyHandler;

import java.util.stream.Collectors;

public class ListNetAmountComputationsEndpoint {

  private final NetAmountComputationRepository repository;

  public ListNetAmountComputationsEndpoint(NetAmountComputationRepository
netAmountComputationRepository) {
    this.repository = netAmountComputationRepository;
  }

  public Router getRouter(Vertx vertx) {
    var router = Router.router(vertx);
    router.route("/computations*").handler(BodyHandler.create());
    router.get("/computations").handler(this::endpoint);

    return router;
  }

  private void endpoint(RoutingContext context) {
    var request = context.request();
    var queryParameters = request.params();

    var skipParam = queryParameters.get("skip");
    if (skipParam == null) {
      skipParam = "0";
    }

    var limitParam = queryParameters.get("limit");
    if (limitParam == null) {
      limitParam = "10";
    }

    var dto = new ListNetAmountComputationsDTO(
      Integer.parseInt(skipParam),
      Integer.parseInt(limitParam)
    );

    var query = new ListNetAmountComputations(
      dto.getSkip(),
      dto.getLimit()
    );

    var queryHandler = new ListNetAmountComputationsHandler(
      repository
    );

    var queryResult = queryHandler.handle(query);

    var mapper = new NetAmountComputationMapper();
    var result = queryResult.stream()
      .map(mapper)
      .collect(Collectors.toList());

    context.response().setStatusCode(HttpResponseStatus.OK.code())
      .putHeader("content-type", "application/json")
      .end(new GsonBuilder().serializeNulls().create().toJson(result));
  }
```

```
}
```

```
package com.ghertzsch.taxcalculator.plugins.endpoints;

import com.ghertzsch.taxcalculator.adapters.ListTaxRatesDTO;
import com.ghertzsch.taxcalculator.adapters.TaxRateMapper;
import
com.ghertzsch.taxcalculator.application.queries.ListNetAmountComputationsHa
ndler;
import com.ghertzsch.taxcalculator.application.queries.ListTaxRatesHandler;
import com.ghertzsch.taxcalculator.domain.queries.ListTaxRates;
import com.ghertzsch.taxcalculator.domain.repositories.TaxRateRepository;
import com.google.gson.Gson;
import com.google.gson.GsonBuilder;
import io.netty.handler.codec.http.HttpResponseStatus;
import io.vertx.core.Vertx;
import io.vertx.ext.web.Router;
import io.vertx.ext.web.RoutingContext;
import io.vertx.ext.web.handler.BodyHandler;

import java.util.stream.Collectors;

public class ListTaxRatesEndpoint {

  private TaxRateRepository repository;

  public ListTaxRatesEndpoint(TaxRateRepository repository) {
    this.repository = repository;
  }

  public Router getRouter(Vertx vertx) {
    var router = Router.router(vertx);
    router.route("/taxrates*").handler(BodyHandler.create());
    router.get("/taxrates").handler(this::endpoint);

    return router;
  }

  private void endpoint(RoutingContext context) {
    var request = context.request();
    var queryParameters = request.params();

    var skipParam = queryParameters.get("skip");
    if (skipParam == null) {
      skipParam = "0";
    }

    var limitParam = queryParameters.get("limit");
    if (limitParam == null) {
      limitParam = "10";
    }

    var dto = new ListTaxRatesDTO(
      Integer.parseInt(skipParam),
      Integer.parseInt(limitParam)
    );

    var query = new ListTaxRates(
      dto.getSkip(),
      dto.getLimit()
    );
```

```
    var queryHandler = new ListTaxRatesHandler(
      repository
    );

    var queryResult = queryHandler.handle(query);

    var mapper = new TaxRateMapper();
    var result = queryResult.stream()
      .map(mapper)
      .collect(Collectors.toList());

    context.response().setStatusCode(HttpResponseStatus.OK.code())
      .putHeader("content-type", "application/json")
      .end(new GsonBuilder().serializeNulls().create().toJson(result));
  }
}
```

Pure Fabrication

Hierbei geht es um die Trennung zwischen Technologie- und Problemdomäne. Die Algorithmen werden also gekapselt.

```
 1 public void loadPopulation(Connection connection) {
 2 try {
 3 Statement statement = connection.createStatement();
 4 ResultSet results = statement.executeQuery(population());
 5 while (results.next()) {
 6 handleElementOf(results)
 7 }
 8 } catch (SQLException exception) {
 9 process(exception)
10 }
11 }
12 private void handleElementOf(ResultSet results) {
13 String name = results.getString("Name");
14 String location = results.getString("Location");
15 Person person = new Person(name, location);
16 simulate(person);
17 }
```

Protected Variations

Es werden verschiedene Implementierungen hinter einer einheitlichen Schnittstelle gekapselt. Ursprünglich ist dieses System als Information Hiding bekannt gewesen. Polymorphie und Delegation sind gute Schutzmöglichkeiten. Ein Beispiel von Protected Variations ist ein Stylesheet im Webumfeld, Spezifikationen von Schnittstellen, Betriebssysteme und Virtuelle Maschinen und auch die Begrenzung von SQL.

Don’t Repeat Yourself! (DRY)

Es ist wichtig, dass es zu keinen Wiederholungen kommt.

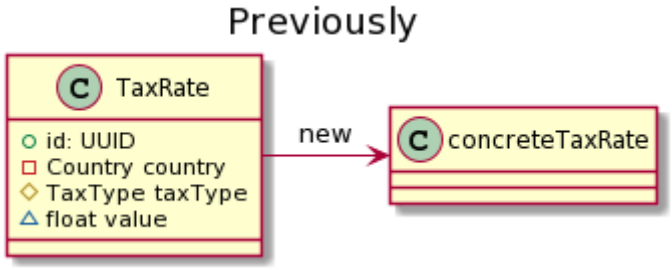
Factory-Design-Pattern

Statt durch direkten Aufruf mithilfe eines Konstruktors wird das Objekt mit einer Methode erzeugt. Der entscheidende Vorteil ist die lose Kopplung und modulare Erweiterbarkeit der Applikation. Falls

sich während der Lebenszeit der Applikation die zu instanziierende Klasse ändert kann einfach eine neue Klasse der TaxRateFactory hinzugefügt werden. Außerdem ist das Testen auf dem Factory-Pattern einfacher als jede einzelne Klasse zu testen. Wenn hinter TaxRateFactory später drei verschiedene Klassen stehen sollten, braucht man nur die TaxRateFactory zu testen.

Für das Speichern der entsprechenden TaxRate-Instanz im taxRateRepository eignet sich die Erzeugung mithilfe der TaxRateFactory. Ohne es nun zu implementieren, können der Factory modifizierte Tax-Rate-Klassen hinzugefügt werden. Es könnte beispielsweise die Klasse TaxRateProgressive hinzugefügt werden. Dann wäre eine Berechnung statt mit einer Flat Tax mit einer Progressive Tax möglich. Hierzu würde statt dem „withValue“ eine Liste mit den Steuer-Stufen eingefügt werden (z.B. 0-30.000 Eur: 14%, 30.000 Eur-45.000 Eur 32%, …).

Davor:



In MainVerticle oder Plugin:

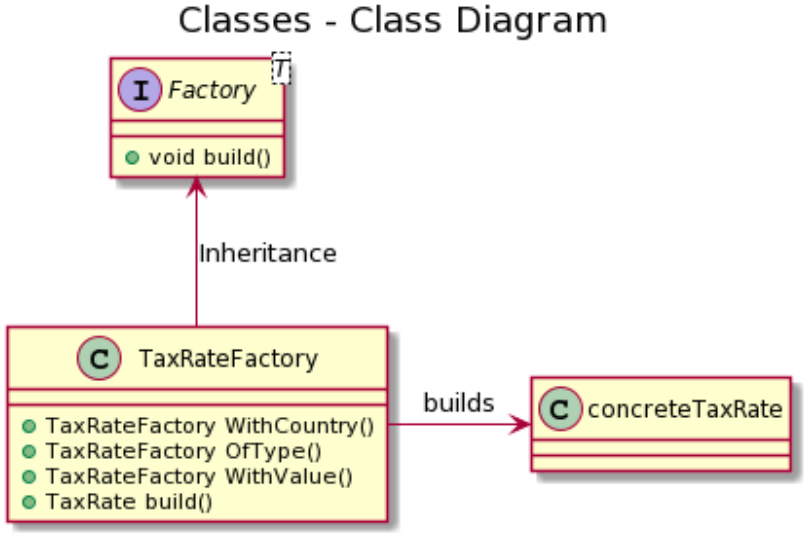
```
taxRateRepository.storeTaxRate(new TaxRate(
      Country.DENMARK,
      TaxType.VALUE_ADDED_TAX,
      25.0f
    ));
    taxRateRepository.findAllTaxRates().stream().map(taxRate ->
taxRate.getId()).forEach(System.out::println);
```

Im Factories-Ordner in der Klasse TaxRateFactory:

```
package com.ghertzsch.taxcalculator.domain.entities;

import com.ghertzsch.taxcalculator.domain.valueobjects.Country;
import com.ghertzsch.taxcalculator.domain.valueobjects.TaxType;

import java.util.UUID;

public class TaxRate {

  private final UUID id = UUID.randomUUID();

  private final Country country;

  private final TaxType taxType;

  private final float value;

  public TaxRate(Country country, TaxType taxType, float value) {
    this.country = country;
    this.taxType = taxType;
```

```
21      this.value = value;
22    }
23
24    public UUID getId() {
25      return id;
26    }
27
28    public Country getCountry() {
29      return country;
30    }
31
32    public TaxType getTaxType() {
33      return taxType;
34    }
35
36    public float getValue() {
37      return value;
38    }
39
40  }
```

Danach:



In MainVerticle oder Plugin:

```
1  var denmarkVat = new TaxRateFactory()
2        .OfType(TaxType.VALUE_ADDED_TAX)
3        .WithCountry(Country.DENMARK)
4        .WithValue(0.25f)
5        .build();
6
7      taxRateRepository.storeTaxRate(denmarkVat);
8      taxRateRepository.findAllTaxRates().stream().map(taxRate ->
  taxRate.getId()).forEach(System.out::println);
```

In der Domain-Schicht im „Factories“-Ordner – Factory-Interface:

```
1 package com.ghertzsch.taxcalculator.domain.factories;
2
3 public interface Factory<T> {
4   T build();
```

```
5 }
```

Im Factories-Ordner in der Klasse TaxRateFactory:

```
1 Package com.ghertzsch.taxcalculator.domain.factories;
2
3 import com.ghertzsch.taxcalculator.domain.entities.TaxRate;
4 import com.ghertzsch.taxcalculator.domain.valueobjects.Country;
5 import com.ghertzsch.taxcalculator.domain.valueobjects.TaxType;
6
7 public class TaxRateFactory implements Factory<TaxRate> {
8
9   private Country country;
10
11   private TaxType taxType;
12
13   private float value;
14
15   public TaxRateFactory() { }
16
17   public TaxRateFactory WithCountry(Country country) {
18     this.country = country;
19     return this;
20   }
21
22   public TaxRateFactory OfType(TaxType taxType) {
23     this.taxType = taxType;
24     return this;
25   }
26
27   public TaxRateFactory WithValue(float value) {
28     this.value = value;
29     return this;
30   }
31
32   @Override
33   public TaxRate build() {
34     return new TaxRate(country, taxType, value);
35   }
36 }
```

Refactoring

Was refactored werden müsste:

- Resources CountryWithTax-Klassen müssen In Json-config-Datei verlegt werden
  - Da hardcoded Werte reingeschrieben wie withValue…
  - Da Resources bei Vert.x in anderen Ordner müssen (resources/conf und resources/webroot)
-