Software-Praktikum

SS 2021, Blatt 2: Grundprinzipien

Ausgabe: 31.08.2021

Prof. Dr. Sven Apel

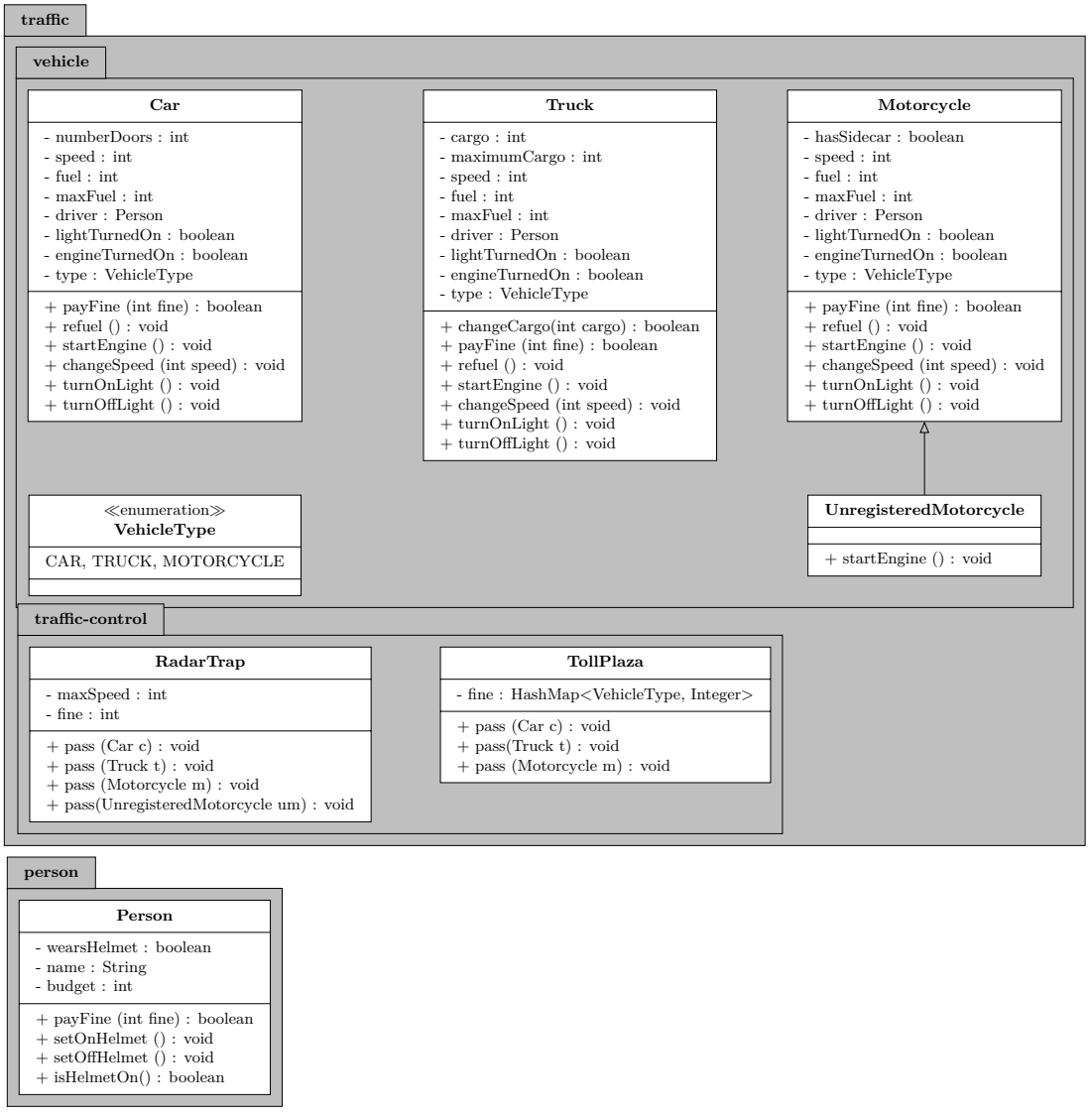
Aufgabe 1 Grundprinzipien der objektorientierten Modellierung

a) Gegeben ist das untenstehende Klassendiagramm zu einer Modellierung von verschiedenen Fahrzeugen (PKW – `Car`, LKW – `Truck`, Motorrad – `Motorcycle`, nicht-registriertes Motorrad – `UnregisteredMotorcycle`), dem Fahrer – `Person`, sowie Strukturen, die der Verkehrskontrolle dienen (Mautstelle – `TollPlaza`, Blitzer – `RadarTrap`). Dieser Klassenentwurf ist jedoch ein sehr früher Entwurf und weist keinerlei Hierarchie oder Abstraktion auf.

Verändern und ergänzen Sie das Klassendiagramm nach den Grundprinzipien der objektorientierten Modellierung.

Hinweise:

- Sie können davon ausgehen, dass gleichnamige Methoden im Package `vehicle` auch dieselbe Funktionalität bereitstellen. Eine Ausnahme bildet dabei die Methode `pass(UnregisteredMotorcycle um)` in der Klasse `RadarTrap`, die im Gegensatz zu den anderen Methoden nichts macht. Außerdem wird in der Methode `startEngine()` der Klasse `Motorcycle` erst der Motor gestartet, wenn sichergestellt ist, dass der Fahrer einen Helm trägt; bei `UnregisteredMotorcycle` hingegen wird sofort der Motor gestartet.
- Sie können vererbte (aber nicht überschriebene) Methoden weglassen.



b) Gehen Sie den Code der Klasse `DynamicBinding` durch und erklären Sie die jeweiligen Ausgaben, die gemacht werden sollten. Wieso sollte es in Zeile 21 keinen Unterschied mehr machen, ob der Helm aufgesetzt ist oder nicht?

Hinweis: Die Konzepte von statischer und dynamischer Bindung sind hierbei hilfreich.

Im Folgenden ist außerdem der Code für die drei Klassen `Person`, `Motorcycle`, `UnregisteredMotorcycle`, sowie `RadarTrap` gegeben.

```
7  class DynamicBinding {
8
9      void main(String[] args) {
10         // Create a driver
11         Person p = Person("James", 100)
12
13         // Create a motorcycle
14         UnregisteredMotorcycle um = UnregisteredMotorcycle(20, p, false)
15         print("Light turned on? " + um.isLightOn())
16         Motorcycle m = (Motorcycle) um
17         print("Engine on? " + m.isEngineTurnedOn())
18         m.turnEngineOn()
19         print("Engine on? " + m.isEngineTurnedOn())
20         p.setHelmetOn()
21         m.turnEngineOn()
22         print("Engine on? " + m.isEngineTurnedOn())
23         m.changeSpeed(100)
24
25         // The motorcycle will pass a radar trap twice in the following
26         RadarTrap rt = RadarTrap(80, 20)
27         rt.pass(um)
28         rt.pass(m)
29         print("Current budget: " + p.getBudget())
30     }
31
32 }
```

```
3  class Person {
4      String name
5      int budget
6      boolean helmetOn
7
8      Person(String name, int budget) {
9          this.name = name
10         this.budget = budget
11         this.helmetOn = false
12     }
13
14     String getName() {
15         return this.name
16     }
17
18     boolean isHelmetOn() {
19         return this.helmetOn
20     }
21
22     void setHelmetOn() {
23         this.helmetOn = true
24     }
25
26     void setHelmetOff() {
27         this.helmetOn = false
28     }
29
30     boolean payFine(int price) {
31         if (this.budget > price) {
32             this.budget -= price
```

```
            return true
        }
        return false
    }

    int getBudget() {
        return this.budget
    }
}
```

```
class Motorcycle {

    boolean hasSidecar
    int speed
    boolean lightTurnedOn
    boolean engineTurnedOn
    int fuel
    int maxFuel
    Person driver
    VehicleType type

    Motorcycle(int fuel, Person driver, boolean hasSidecar) {
        this.fuel = fuel
        this.driver = driver
        this.lightTurnedOn = true
        this.hasSidecar = hasSidecar
        this.type = VehicleType.MOTORCYCLE
    }

    VehicleType getType() {
        return this.type
    }

    boolean payFine(int price) {
        return this.driver.payFine(price)
    }

    boolean isLightOn() {
        return this.lightTurnedOn
    }

    void turnLight() {
        this.lightTurnedOn = !this.lightTurnedOn
    }

    void changeSpeed(int changedValue) {
        this.speed += changedValue
    }

    int getSpeed() {
        return this.speed
    }

    void fuel() {
        this.fuel = maxFuel
    }

    boolean isEngineTurnedOn() {
        return this.engineTurnedOn
    }

    void turnEngineOff() {
        this.engineTurnedOn = false
    }
```

```
59
60      void turnEngineOn() {
61          if (driver.isHelmetOn()) {
62              this.engineTurnedOn = true
63          }
64      }
65  }
```

```
5   class UnregisteredMotorcycle extends Motorcycle {
6
7
8       UnregisteredMotorcycle(int fuel, Person driver, boolean hasSidecar) {
9           super(fuel, driver, hasSidecar)
10      }
11
12      @Override
13      void turnEngineOn() {
14          super.engineTurnedOn = true
15      }
16  }
```

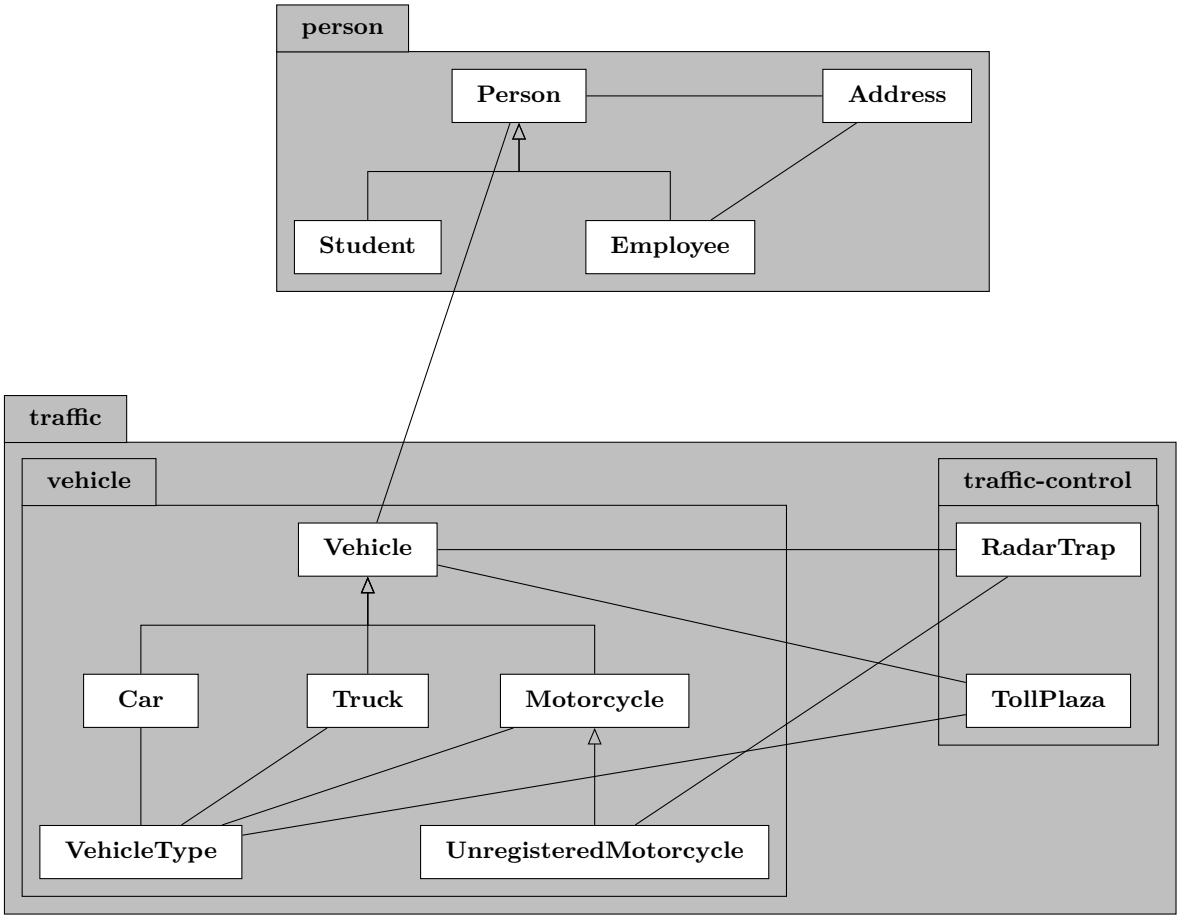
```
6   class RadarTrap {
7       int maxSpeed
8       int fine
9
10      RadarTrap(int maxSpeed, int fine) {
11          this.maxSpeed = maxSpeed
12          this.fine = fine
13      }
14
15      // Other non-relevant methods
16
17      void pass(Motorcycle m) {
18          print("Vehicle passed.")
19          if (m.getSpeed() > this.maxSpeed) {
20              if (m.payFine(fine)) {
21                  print("Vehicle had to pay!")
22              } else {
23                  print("Rolling a dice to decide whether the driver will be arrested or
                        ↪ not.")
24              }
25          }
26      }
27
28      void pass(UnregisteredMotorcycle um) {
29          print("Unregistered motorcycle passed; Could not register the license plate.")
30      }
31  }
```

c) Betrachten Sie nachfolgende Module aus einer Erweiterung des initial vorgestellten Modells und beantworten Sie für die folgenden Module, ob sie jeweils eine geringe oder hohe Kohäsion beziehungsweise eine starke oder schwache Kopplung aufweisen.

(i) vehicle, traffic-control

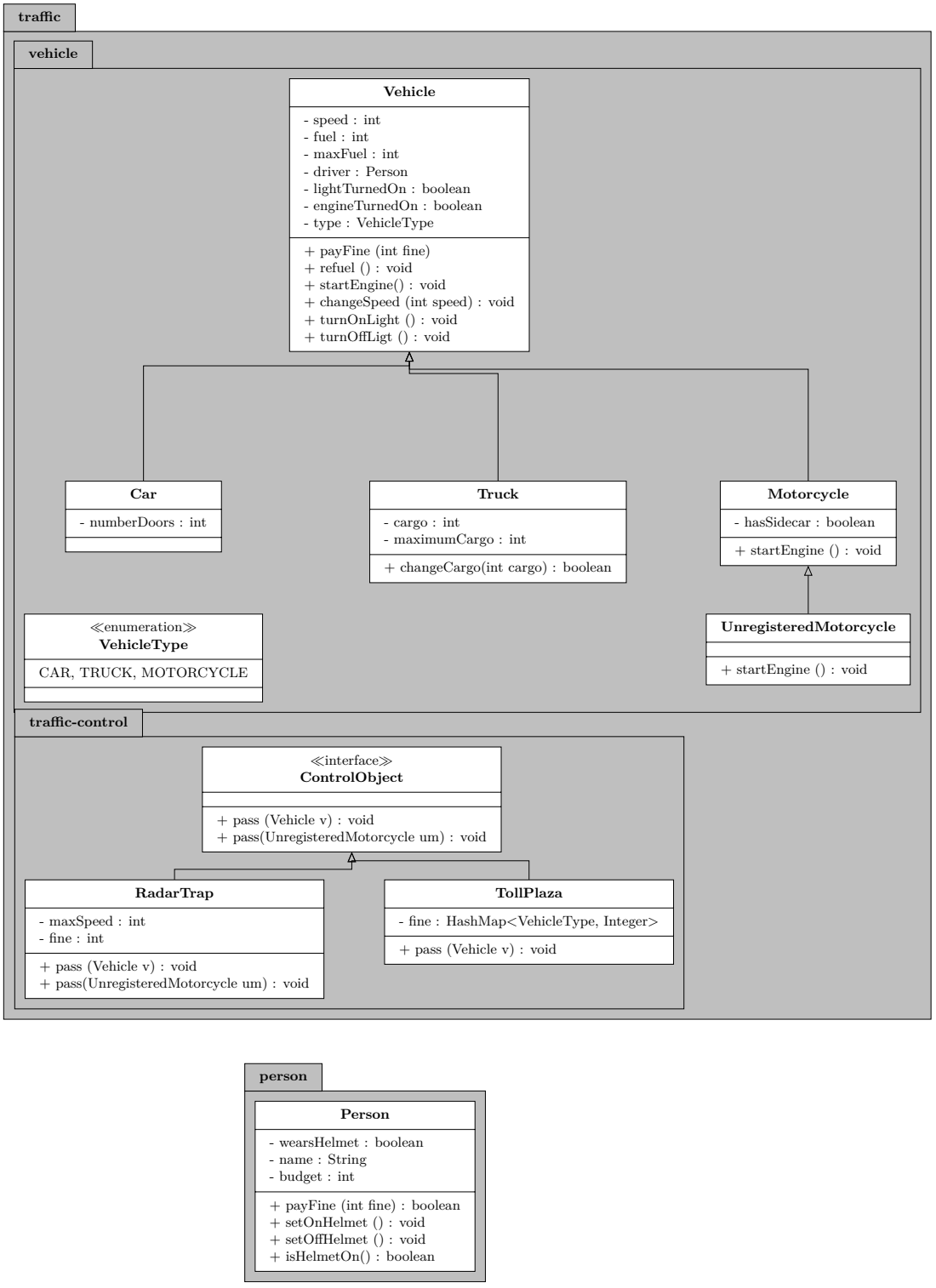
(ii) traffic, person

Nennen Sie die Vor- und Nachteile sowohl auf genereller Ebene als auch auf Modularitätsebene, wenn man die Klassen im Package `traffic` in weitere Packages (`vehicle` und `traffic-control`) unterteilt.


person
Person
Address
Student
Employee
traffic
vehicle
traffic-control
Vehicle
RadarTrap
Car
Truck
Motorcycle
TollPlaza
VehicleType
UnregisteredMotorcycle

Lösung

a)



b) Das Ergebnis lautet:

```
Light turned on? true
Engine on? false
Engine on? true
Engine on? true
Unregistered motorcycle passed; Could not register the license plate.
Vehicle passed.
Vehicle had to pay!
Current budget: 80
```

Statische Bindung wird beim Auswählen des jeweiligen Methodenaufrufs ausgeführt. Das heißt, dass einmal `pass(UnregisteredMotorcycle um)` als korrekte Methode für den Aufruf mittels `UnregisteredMotorcycle` aufgerufen wird, da der statische Typ des ersten Aufrufs `UnregisteredMotorcycle` ist. Beim zweiten Aufruf der Methode `pass` ist der statische Typ jedoch durch den Cast `Motorcycle` und daher wird `pass(Motorcycle m)` aufgerufen.

Dynamische Bindung kommt dann ins Spiel, wenn ausgewählt werden muss, aus welcher Klasse die jeweilige Methode ausgeführt wird. Der dynamische Typ ändert sich nicht mittels Casts. Deswegen wird immer die

Methode `startEngine` aus der Klasse `UnregisteredMotorcycle` ausgeführt.

c) (i) vehicle, traffic control: hohe Kohäsion (allerdings auf Seite von `traffic-control` nicht ganz eindeutig), hohe Kopplung

(ii) traffic, person: hohe Kohäsion, schwache Kopplung

Vorteile der Unterteilung:

- Bessere Übersichtlichkeit
- Bessere Abtrennung der Funktionalitäten

Nachteile der Unterteilung:

- Man hat eine hohe Kopplung zwischen den Packages; es leidet die Modularität
- Pakete können manchmal zu klein sein, um eine Aussage über die Kohäsion treffen zu können