B-OOP 2025: Semester assignment

Read **the whole** assignment several times carefully and then start solving it! We recommend creating a reasonable representation of your knowledge so that you don't forget anything (e.g. use a mind map).
We expect you to consult via one of the communication channels (emails, discord - if you do not have access, please email [redacted]).

1 Structure of the assignment

Download the file for the term assignment. You will find the `test` folder in it. The file has the following structure:

```
assignment.z
ip└── test
    └── RequiredTests.java
```

Place the test folder at the same level as the src folder of your implementation. You may not change the location of the `RequiredTests` class, its name, or the names of the tests.

2 Overview of the assignment

Your task is to implement a very simple version of an insurance system for small insurance companies. Your system will support three simplified types of insurance policies:

- `SingleVehicleContract` - compulsory insurance for a vehicle
- `MasterVehicleContract` - compulsory insurance for a fleet of vehicles, i.e. a collective insurance contract
- `TravelContract` - travel insurance

The characteristics and attributes of these contracts will be inspired by real insurance systems, but in several cases we will depart from them for simplicity. Contracts contain payment details, policyholder details, insurer details, insured objects, etc.

These contracts will be administered by an insurance company (`InsuranceCompany`). The InsuranceCompany creates and modifies the contracts. Pois-
The Commission regularly evaluates the maturity of contracts. On each contract where the maturity date has already passed, the insurer sets an arrears amount equal to the sum insured.

Payments on contracts are orchestrated by `PaymentHandler`. It has the task to correctly execute the payment of the contract (framework
or other) and store payment history.

The system will distinguish between two types of insured objects:

- `Person` - A person (both legal and natural, because we don't care in terms of insurance policies).
- `Vehicle` - vehicle

Last but not least, the insurance company supports the payment of insurance benefits in the event of a successfully resolved insurance claim.

Figure 1 shows a simplified overview of the classes you will implement and the relationships between them. This overview is informative.

3 Glossary of terms

In Table 1 you will find a brief overview of the terms that will be useful for implementing the assignment and their approximate translations. Some of the terms do not exist in foreign insurance companies, in these cases we use the translations created by Slovak insurance companies. We explain the terms mainly in the context of the assignment.

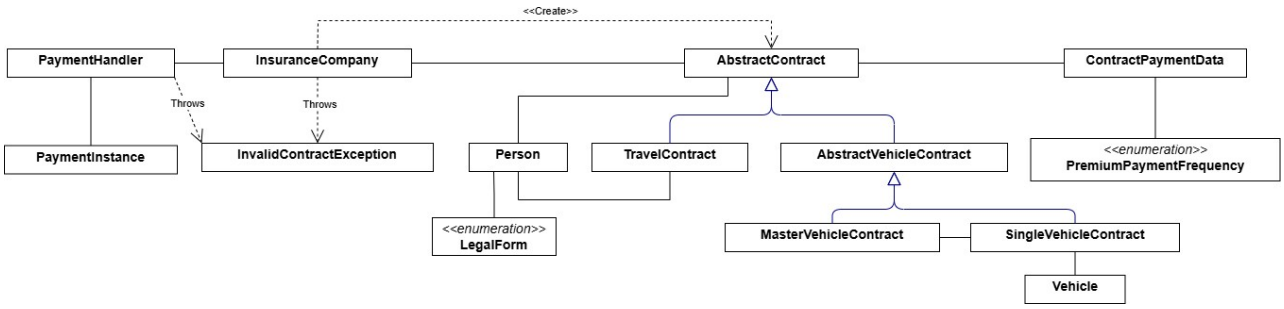

Fig. 1: Overview UML class diagram

Table 1: Table of terms from the insurance domain and their approximate translations

Slovak term	English term	Notes
Insurance contract	Insurance contract	Contract for the provision of insurance services. Insurance the contract in this assignment must contain the policyholder, the insurer, the insured objects and additional payment details.
Insurer	Insurer	An entity that provides insurance , i.e. po- ishroom.
The Insurer	Policy holder	An entity that has concluded an insurance contract with an insurance to the voters. In most cases, the policyholder is the person who pays the premium regulations on the policy. The fact that a person is the policyholder does not necessarily mean that he or she is also the insured. For example, a person who is a policyholder on an assignment policy is not necessarily the insured on that policy.
Authorised person	Beneficiary	A person to whom a premium is preferentially paid in full not in the event of an insurance claim. For example, if there is an accident involving a vehicle that has a contract of insurance with both a policyholder and a beneficiary, the insurance benefit is paid to the beneficiary. If no authorised person is named, the insurance benefit shall be paid to the policyholder. For example, the purchase of a leased vehicle. The policyholder is the person paying off the lease and the beneficiary is the leasing company. As the leasing is the owner of the vehicle, the damage to the property was incurred by it, not by the policyholder. Therefore, the insurance claim, if any, is payable to it.
Insurance benefits	Coverage amount	The amount paid by the insurer to the beneficiary shall be to the policyholder (policyholder, beneficiary, insured person...), in case of a positive evaluation of the reported insured event. For example, the policyholder has taken out an insurance policy for his/her vehicle. A crash occurs (an insured event). The policyholder reports the claim. The insurer evaluates all the elements of the report and decides whether the client is entitled to a 'payment of insurance'. If so, it will pay the client the amount of the claim.
Insurance premiums	Insurance premium	The amount to be paid for a given payable ob- period for the provision of insurance services to the insurer. For example, you pay an each year to the insurance company for providing the PZP. This amount is called the premium.
Insurance event	Claim	Any event that may lead to the payment of post- certain performance. For example, in the case of an accident insurance policy.

Table 1: Table of terms from the insurance domain and their approximate translations

Slovak term	English term	Notes
Arrears	Outstanding balance	An outstanding amount that is associated with any by contract. For example, if you have an insurance policy that you pay annually but did not pay last year, you will incur an underpayment on your policy in the amount of the premium. The arrearage is accrued after each billing period. In the context of this assignment, the underpayment on the contract is a positive number. If there is an overpayment (you have paid more than the premium), the underpayment is a negative number.
Insured object	Insured object	The object which is the subject of the insurance policy. Poiste- there may be several objects on one contract. In the case of a PZP, it is a car. In the case of travel insurance, it is the person or persons travelling.
Natural person	Natural person	A person identified by a birth number.
Legal entity	Legal person	A person identified by an ID number.

4 Data model

In this section you will find the UML class diagram (see Figure 2) that your implementation must conform to. This means:

- Your solution must contain all the classes, enumerations, and exceptions that are in this diagram. The names of these objects in your solution must match their names in the diagram. You must also respect the structural relationships between these objects (inheritance, associations, dependencies...) that the diagram prescribes. You must respect the placement of classes in packages and the names of packages. You must implement all the packages that are listed in the diagram in the src package (see 6).
- Your solution must contain all the attributes and methods that each object in this diagram has. The names of these attributes and methods in your solution must match their names in the diagram. You must satisfy the visibility modifiers (private, protected, public, default) that are prescribed by the diagram. You must comply with the attribute data types and method signatures that are prescribed by the diagram.
- In your solution, you can add additional methods and/or attributes to individual classes if your solution requires it. If you add methods and/or attributes that are not in the diagram, you must be able to justify why you have done so. Even if you do not add methods or attributes, you will need to be able to justify why you made that decision.
- You need to understand the diagram above. During the handover, the practitioners may also ask you questions about this diagram.

The above UML class diagram captures relatively much detail, but depending on your implementation, it may not contain all the necessary methods and attributes.

5 Specification

5.1 Common information

The following applies to all classes:

- If a class method throws an exception that accepts a message as a parameter, you can use an arbitrary message. In tests, the type of exception thrown is checked, not the message. Warning. This does not mean that messages can be meaningless. Remember that what is being evaluated is, among other things, the cleanliness of your code (see Section 7).
- Conversely, if the specification below does not explicitly state that the method throws an exception, then the method must not throw an exception.
- If a class method performs a calculation with sums of money, it is rounded **down** (as was the case for small assignments). That is:
  - If the sum is 4, half of the sum is 2.

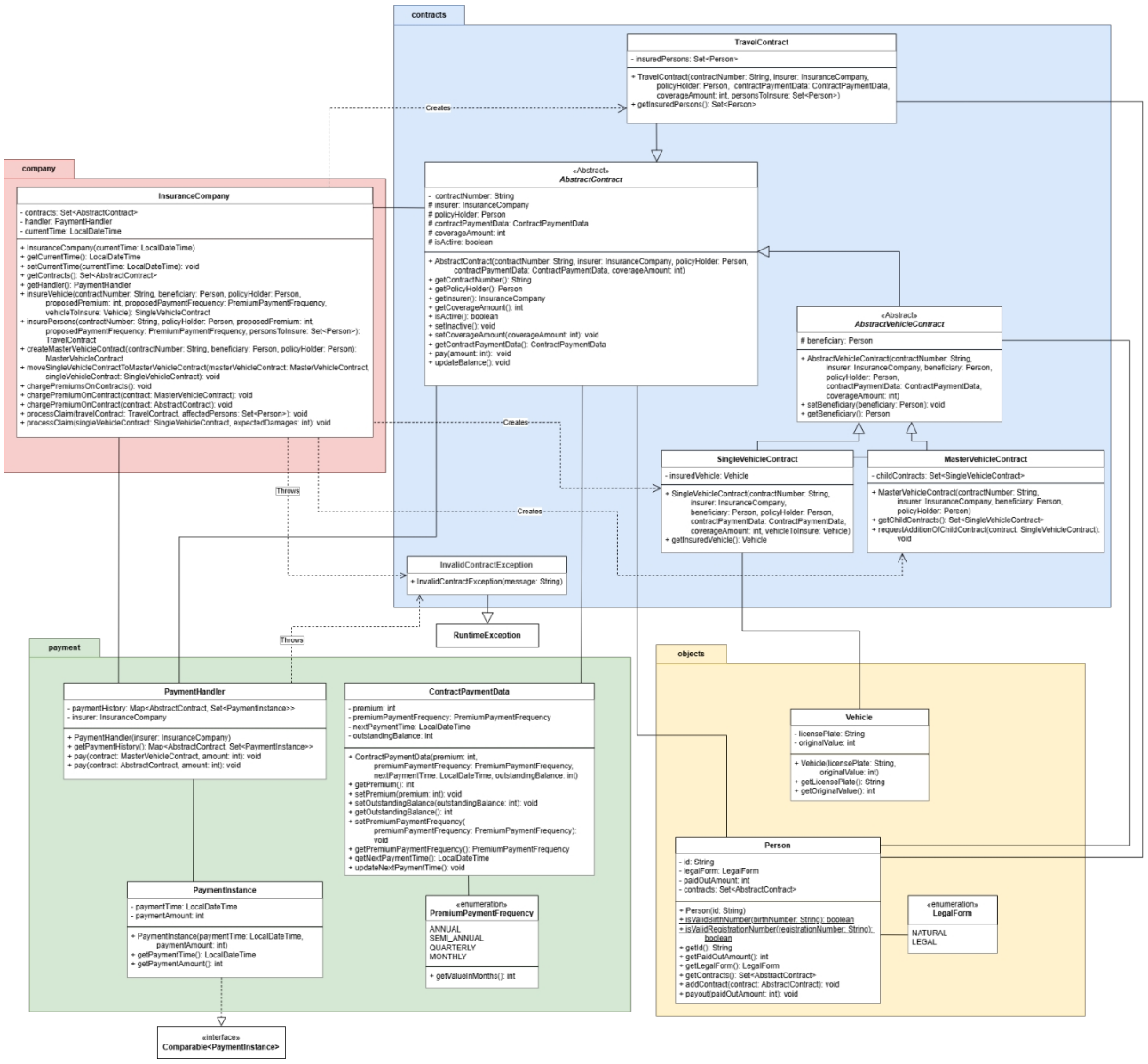

Fig. 2: UML class diagram of the semester assignment

  - If the sum is 5, half of the sum is 2.
  - If the amount is 5, 70% of the amount is 3.

- If the specification states that an attribute of a class will not change after it is constructed, it means that this attribute is to be `final`. However, this does not necessarily mean that its content will not change. For example, an attribute declared as `private final List<Integer> list;` means that `list` is a constant reference, but the contents of that list may change.
- Except for use in the `equals` method, you must not use the `instanceof` operator anywhere in your specification.

5.2 Contract

Our simple insurance system contains 4 classes of contracts forming a hierarchy. At the top of this hierarchy is the `AbstractContract` class.

5.2.1 AbstractContract

This class has the following attributes:

- contract number - `contractNumber`. This is a non-empty string different from `null`. This string serves as a contract identifier. The contract number must be unique within a single insurance company. Once set, the contract number must not be changed.
- insurer - `insurer`. This is a reference to the insurer (`InsuranceCompany` type) that has entered into the contract with the policyholder (`Person` type), i.e. it is the insurance company that has created the instance of the contract. The insurer must not be `null` and must not be changed once set.
- policyholder - `policyHolder`. This is a reference to the person who has entered into a contract with the insurer. The policyholder must not have a `null` value and must not be changed once set.
- contract payment details - `contractPaymentData`. This data contains the amount of the premium, the frequency of payment, the time of the next due date and the status of the payment (i.e. the existence of arrears or overpayments). The attribute must not be changed after setting.
- the amount of the insurance benefit - `coverageAmount`. The amount of the insurance benefit must be non-negative.
- the `isActive` attribute, which indicates that the policy is live and that it is available for payment and claims reporting. When the contract is created, this attribute is set to `true`.

Class thrown out:

- an `IllegalArgumentException` exception in the constructor if any validation criterion is not met.
- in the set method `setCoverageAmount` an `IllegalArgumentException` exception if the new amount is invalid.

5.2.2 TravelContract

Another type of contract is the `TravelContract`. It is a simple travel insurance policy. Its constructor ac- cepts the same parameters as the `AbstractContract` constructor, plus the `personsToInsure` parameter. It is a set that must not be `null` and must be non-empty. It is used to set the `insuredPersons` attribute, which must then not be changed. The insured may or may not be part of the insured persons. In the case of travel insurance, only natural persons can be insured, but the insured can also be a legal person (for example, an employer insures employees before they travel to a symposium abroad). If the above constraints of the `personsToInsure` set are not met or if `contractPaymentData` is `null`, the constructor throws an `IllegalArgumentException` exception.

5.2.3 AbstractVehicleContract

Car insurance is slightly more complicated. The `AbstractVehicleContract` class adds a `beneficiary` attribute, which can be `null`. If `beneficiary` is `null`, the contract does not have a beneficiary and any claim is paid to the policyholder. If the `beneficiary` is the same person as the `policyHolder`, the constructor throws an `IllegalArgumentException`. If the policy has a , any claim is paid to the beneficiary.

5.2.4 SingleVehicleContract

From `AbstractVehicleContract` inherits the `SingleVehicleContract`, which is used to insure a specific vehicle (`Vehicle`). This contract roughly represents a PZP. The Vehicle, which is the insured object on this contract, is a constructor parameter and must not be `null`. If this parameter is `null` or if `contractPaymentData` is `null`, the constructor throws an `IllegalArgumentException` exception. The vehicle set in the constructor shall not be changed.

5.2.5 MasterVehicleContract

The last contract is the `MasterVehicleContract`, which is a master contract used to insure a fleet of vehicles. It contains a set of `SingleVehicleContracts` for individual vehicles. As a real-world example, the Bratislava Transport Company has a master contract for PZP and this master contract contains PZP contracts for individual vehicles in the . The insurer on the `MasterVehicleContract` must be a legal entity. `The MasterVehicleContract` must set the payment data (`contractPaymentData`) to `null` and the `coverageAmount` to 0. If the above constraints do not apply, the constructor throws an `IllegalArgumentException`. The constructor also initializes the `childContracts` set. After initialization, this set must not be modified. Contracts are stored in this set in the order in which they were added to it (i.e., use the appropriate set implementation).

Vehicle contracts are added to the `MasterVehicleContract` by making a request to the insurer, i.e. calling the `InsuranceCompany::moveSingleVehicleContractToMasterVehicleContract` method.
`A MasterVehicleContract` is considered inactive if all of its child contracts are inactive. If it has no child contracts, its activity is evaluated according to the `isActive` attribute. The `setInactive` method must set all of its child contracts as inactive, including its `isActive` attribute.

5.2.6 Payment of contracts

Contracts can be paid. Payment is made by calling the `pay` method in the amount of the `amount`. `The amount` parameter can be any positive value. This means that the client does not have to pay the entire outstanding amount, or, on the contrary, can pay more than the `premium` prescribed in `contractPaymentData`. However, nothing is set directly in the contract. You can think of the `pay` method as an API that is exposed to the user. The `pay` method on the contract must internally call the corresponding `pay` method from the `PaymentHandler` class. In other , the sequence of operations when a user wants to pay some of their contract is as follows:

1. the user calls its `pay` method on the contract
2. the `pay` method of this contract calls the appropriate `pay` method from the `PaymentHandler` class
3. the `pay` method of the `PaymentHandler` class executes the payment logic

Your implementation must ensure that the appropriate variant of the `pay` method from the `PaymentHandler` class is called for the given contract, i.e. it should be called:

- method `pay(MasterVehicleContract, int)` for `MasterVehicleContract`.
- method `pay(AbstractContract, int)` for another contract.

Work out how this behaviour can be achieved. (Hint: You will need to rewrite `the pay` method appropriately.)

It is also possible to request an update of the arrears on the contract by calling the `updateBalance` method. This method is again just a request to perform an operation, the contract itself should not arbitrarily modify the arrears. The `updateBalance` method internally calls the appropriate `chargePremiumOnContract` method from the `InsuranceCompany` class, again you must ensure that the correct variant is called.

5.3 Insured objects - Insured objects

In our simple insurance system, only 2 types of objects can be insured: persons and vehicles.

5.3.1 Person

A person is represented by the `Person` class. It has the following attributes:

- `id` - a string other than `null` that is non-empty. Once the constructor sets the `id`, it must not be changed. `id` must be either a valid birth number or ID number.
- `legalForm` - type of person. If the `id` is a birth number, it is a natural person (`NATURAL`). If `id` is ID number, it is a legal person (`LEGAL`). This attribute must not be changed after initialization.

- `paidOutAmount` - this is the aggregate amount paid out from all processed . The constructor initializes it to 0.
- `contracts` - the set of contracts on which the given person is listed as `policyHolder`. This set stores the contracts in the order in which they were entered. Once initialized in the constructor, this set may not be changed.

We consider a valid birth number to be a string :

1. is not `null` and is 9 or 10 characters long and all its characters are digits (i.e. the birth number is without the slash symbol).
2. is in the format RRMMDDNNNN or RRMMDDNNNN.
3. The month (MM) must be in the range 1-12 (inclusive) or 51-62 (inclusive; this range indicates that the RIN belongs to a woman, and the month of her birth can be obtained by subtracting 50 from MM).
4. If the registration number has 9 characters, the year (YY) must be less than or equal to 53 (i.e. the registration number was issued up to and including 1953). If the date of birth added from the RR is an existing historical date, then the RR is valid. (You can verify the existence of the date by using the `LocalDate` class.)
5. If the registration number has 10 characters (i.e. the registration number was issued from and including 1954), then the control amount must apply.
   Let c_i denote the numerical value of the i-th digit of the RČ. The checksum holds if the relation $\sum_{i=0}^{(9)} (\quad -1)^i c_i \bmod 11 = 0$ holds.
   (Note: there are also valid RINs for which this checksum does not apply. However, your implementation does not have to take such accounts account.) If the applies, you still need to verify that the date of birth calculated from the RIN is an existing historical date. (You can verify the existence of the date by using the `LocalDate` class.) If so, it is a valid date.

We consider a valid ID to be a string that is not `null`, and consists of 6 or 8 characters that are digits.
If the `id` is not valid (it is `null` or not evaluated either as a valid ID or as a valid ID), constructor of the `Person` class throws an `IllegalArgumentException`. In addition, `the Person` class throws an `IllegalArgumentException` in the `payout` method if its argument is a non-positive amount, or in the `addContract` method its argument is `null`. In the `payout` method, the parameter of the `paidOutAmount` method is added to the total paid amount of the person `this.paidOutAmount`.

5.3.2 Vehicle

The second insured object is the vehicle (`Vehicle`). The Vehicle contains the license plate (`licensePlate`) and the price (`originalValue`). The `licensePlate` attribute is a `non-null` string must be 7 characters long. All its characters must be uppercase letters A-Z or digits. The `originalValue` attribute must be positive. If the above constraints do not apply, the constructor throws an `IllegalArgumentException`. Neither of these attributes may be changed once set.

5.4 Payments - Payments

5.4.1 ContractPaymentData, PremiumPaymentFrequency

The payment data on a contract is represented by the `ContractPaymentData` class. It contains attributes:

- `premium` - the amount to be paid for a given maturity period. It must be positive.
- `premiumPaymentFrequency` - frequency of payments (i.e. frequency of contract payments), must not be `null`. Possible values of this attribute are:
  - `ANNUAL` - annual payment (`getValueInMonths` returns value 12)
  - `SEMI_ANNUAL` - semi-annual payment (`getValueInMonths` returns value 6)
  - `QUARTERLY` - quarterly payment (`getValueInMonths` returns value 3)
  - `MONTHLY` - monthly payment (`getValueInMonths` returns value 1)
- `nextPaymentTime` - the time when the next contract payment is due. Must not be null.
- `outstandingBalance` - arrears.

If the above constraints are not met, the constructor throws an `IllegalArgumentException` exception. For a better understanding of the meaning of attributes, we will list the following:

- If `premium` is equal to 10 and `premiumPaymentFrequency` is set to `SEMI_ANNUAL`, means that 2 payments are to be made on this policy per year (the policy is paid every 6 months), so we expect the policyholder to pay 2*10=20 per year.
- `outstandingBalance` captures the payment status of the contract. If `outstandingBalance` is positive, the contract is in arrears. If it is zero, the contract is paid. If it is negative, there is an overpayment on the contract. For example, if `outstandingBalance` is -10, the client has paid 10 more than he should have. That's fine, the next time he pays, he may (or may not) pay less because of it.
- `nextPaymentTime` is the time when the insurance company adds the outstanding amount to the `premium` value. For example, if `the premium` is 10, `outstandingBalance` is 5, the current date according to the insurer is 01 February 2025 and the policy has a date in `nextPaymentTime` of 31 January 2025, this means that when the insurer calls `updateBalance` on the policy, the insurer should increase `outstandingBalance` to the value 5+10=15.

In addition, `the ContractPaymentData` class throws an exception in the set methods for the `premium` and `premiumPaymentFrequency` attributes if their arguments are invalid. The `updateNextPaymentTime` method sets the `nextPaymentTime` to the new due date by adding the number of months indicated by the payment frequency. For example, if the payment frequency is semi-annual, the new payment date is `nextPaymentTime` plus 6 months.

5.4.2 PaymentHandler, PaymentInstance

`PaymentHandler` takes care of the execution and processing of payments. `PaymentHandler` stores the instance of the insurance company to which it belongs. If it does not, the constructor throws an `IllegalArgumentException`. Once set, the `insurer` attribute cannot be changed. The constructor also initializes the `paymentHistory`. After initialization, this attribute cannot be changed either.

The `PaymentHandler::pay(AbstractContract, int)` method is implemented follows:

- If the contract is `null` or the `amount` is non-positive, it throws an `IllegalArgumentException`.
- If the contract is inactive or is not the contract of the insurer that runs this `PaymentHandler`, it throws an `InvalidContractException` exception.
- Otherwise, it will reduce the `outstandingBalance of` the contract by the value of `the amount`.
- It then creates a record of the payment execution (a `PaymentInstance` instance) with the current time per policy and the amount paid and stores it in the `paymentHistory` for the paid contract.

The `PaymentHandler::pay(MasterVehicleContract, int)` method is implemented follows:

- If the contract is `null` or the `amount` is non-positive, it throws an `IllegalArgumentException`.
- If the contract is inactive or is not the contract of the insurer that runs this `PaymentHandler`, or does not contain any child contracts, it throws an `InvalidContractException` exception.
- Otherwise, it iterates over all child contracts (`childContracts`) and tries to zero out their non-payments (always subtracting the consumed funds from the `amount`). If any funds remain, it iterates through the contracts again and creates overpayments of `premium` (or `amount` if there are no funds left). If there is still finance left, it iterates again. It iterates until it has consumed the entire amount.
- It then creates a payment execution record (a `PaymentInstance` instance) with the current time per policy and the amount paid, and saves it in the `paymentHistory` to the paid `MasterVehicleContract` contract (not to the individual child contracts). The payment history for a given contract is saved in order of when the payment was made, from oldest to youngest. This can be achieved by a suitable implementation of the `Comparable` extension.

`The MasterVehicleContract` payment process is captured in the following pseudocode:

```
pay(MasterVehicleContract contract, int amount): for
    each active child contract:
        If the contract is in arrears:
            If I have sufficient funds amount:
                amount -= contract.arrears
                contract.arrears = 0
            Otherwise:
                contract.arrears -= amount amount =
                0
```

```
As long as the amount is valid> 0:
    For each active child contract: If I
        have enough funding amount:
            contract.arrears -= contract.premium
            amount -= contract.premium
        Otherwise:
            contract.arrears -= amount amount =
            0
```

Let's take an example. Let's have a master contract that contains 4 child contracts:

- contract1 - has `premium` equal to 30 and is active.
- contract2 - has a `premium` of 50 and is active.
- contract3 - has a `premium` of 75 and is active.
- contract4 - has `premium` equal to 20 and is inactive.

	Before payment	**After payment of the non-surcharges**	**First cycle after reimbursement**	**Second cycle after reimbursement**
contract1	30	0	-30	-60
contract2	50	0	-50	-85
contract3	100	0	-75	-75
contract4	0	0	0	0
Amount	400	220	65	0

Table 2: Example of intermediate values for the payment of the framework contract.

`The PaymentInstance` class captures a single payment made - when it was made and at what value. Its `paymentTime` attribute must not be `null` and the `paymentAmount` must be positive. If these constraints do not apply, the con- structor throws an `IllegalArgumentException` exception. Once these attributes are set in the constructor, they cannot be changed.

5.5 Insurer - InsuranceCompany

The last component of the assignment is the `InsuranceCompany` class. This class creates and manages contracts, updates their payment data and last but not least processes insurance . `The InsuranceCompany` constructor takes the current time - `currentTime` - as a parameter (Note: Despite the name `currentTime`, this time will be set as in the tests, i.e. it is not guaranteed that `currentTime.isEqual(LocalDateTime.now())` is valid. If `currentTime` is `null`, the constructor throws an `IllegalArgumentException`. The constructor also initializes the set of contracts that the insurance company has entered into and creates a payment manager `handler`. Once initialized, these two attributes cannot be changed. The contracts are stored in the set of contracts in the order in which the insurance company made them.

The `currentTime` attribute be set by the corresponding set method. If the new `currentTime` is `null`, this method throws an `IllegalArgumentException`.

5.5.1 Creating new contracts

New contracts are concluded by the insurer using the following methods:

- `insureVehicle` - For the `contractNumber` parameter it must be true that there is no other contract with this number in the given insurance company. The total annual amount by the policyholder must be greater than or equal to 2% of the price of the vehicle. If these constraints are not met or if any of the arguments needed for the calculation are invalid, the method throws `an IllegalArgumentException`. If, on the other hand, they are satisfied, the method creates a new `SingleVehicleContract` that has `coverageAmount` set to half the value of the vehicle. In the payment data, the `premium` and `premiumPaymentFrequency` are set to the suggested values, the arrearage is set to 0, and the next payment date is set to the `currentTime` of the insurance company. Subsequently, the insurance company calls the `chargePremiumOnContract` method with this contract. Then the newly created contract is stored in the insurance company's contract set, the policyholder's contract set, and the method returns it.
- `insurePersons` - For the `contractNumber` parameter it must be true that there is no other contract with this number in the given insurance company. The total annual amount paid by the policyholder must be greater than or equal to five times the number of insured persons. If these limitations are not met or if any of the arguments necessary

on the calculation is invalid, the method throws an `IllegalArgumentException`. If, on the other hand, they are satisfied, the method creates a new `TravelContract` that has `coverageAmount` set to ten times the number of insured people. In the payment data, `premium` and `premiumPaymentFrequency` are set to suggested values, arrearage is set to 0, and the next payment date is set to the `currentTime` of the insurance company. Subsequently, the insurance company calls the `chargePremiumOnContract` method with this contract. Then the newly created contract is stored in the insurance company's contract set, the policyholder's contract set, and the method returns it.

- `createMasterVehicleContract` - For the `contractNumber` parameter it must be true that there is no other contract with this number in the given insurance company. If this constraint is not met, the method throws `an IllegalArgumentException`. If, on the other hand, it is satisfied, the method creates a new `MasterVehicleContract` that does not yet contain any child contracts. Then the newly created contract is stored in the set of insurance company contracts, the set of policyholder contracts, and the method returns it.

In order to add a subsidiary contract to a master contract, you must first create a separate subsidiary contract (i.e., `SingleVehicleContract`) and then ask the insurer to move it to the master contract. The move is performed by the `moveSingleVehicleContractToMasterVehicleContract` method. If any of its parameters are `null`, it throws an `IllegalArgumentException`. Both contracts that are its parameters must be active and must be entered into by the insurance company for which the move is requested. Additionally, both contracts must have the same policyholder. If these constraints do not apply, an `InvalidContractException` exception is thrown. If the constraints are satisfied, then `singleVehicleContract` is removed from the contract sets of both the insurer and the policyholder and added to the child contract set `of masterVehicleContract`. The payment history of the `singleVehicleContract` is left unchanged.

5.5.2 Update of outstanding contracts

The `chargePremiumsOnContracts` method iterates over all contracts that the insurance company has entered into (found in the set of contracts), and the `updateBalance` method is called on each contract that is active. The `updateBalance` method in turn calls some implementation of the `chargePremiumOnContract` method:

- `chargePremiumOnContract(AbstractContract)` - the method does not validate the argument. Validates whether the given contract has a due date before `currentTime` or is equal to `currentTime` (`isBefore`, `isEqual`). If so, it increments the contract's arrearage by the `premium` value from `contractPaymentData`. It also updates the due date on this contract (according to `premiumPaymentFrequency` in `contractPaymentData`). It repeats this process until the `currentTime` is earlier than the `nextPaymentTime` due date.
- `chargePremiumOnContract(MasterVehicleContract)` - the method does not validate the argument. Iterates through all child contracts of the given framework contract and calls `chargePremiumOnContract` for each of them.

5.5.3 Insurance claims processing

It is possible to report claims on insurance policies. However, the claims reporting process is extremely complicated and would take several dozen pages to describe. In your implementation, you will create two `processClaim` methods that assume that some claims handling process has already taken place, and are only responsible for paying the insurance claim. They work as follows:

- `processClaim(SingleVehicleContract, int)` - parameter `singleVehicleContract` must be different from `null`. The `expectedDamages` parameter must be positive. If these conditions are not met, the method throws an `IllegalArgumentException`. In addition, `singleVehicleContract` must be an active contract. If it not, the method throws `an InvalidContractException`. If there is an eligible person, the `coverageAmount` is paid (by calling its `payout` method). If there is none, this amount is paid to the policyholder. If the `expectedDamages` parameter is greater than or equal to 70% of the value of the vehicle, this is a total loss and the contract itself is changed to inactive.
- `processClaim(TravelContract, Set<Person>)` - parameter `travelContract` must be different from `null`. The `affectedPersons` parameter must be distinct from `null` and must be a non-empty set that is a subset of the insured persons in the `travelContract`. If these conditions are not met, the method throws `an IllegalArgumentException`. In addition, the `travelContract` must be an active contract. If it is not, the method throws `an InvalidContractException`. If the conditions are met, the coverage amount is calculated as `coverageAmount / affectedPersons.size()` and this coverage is paid to all persons in the set `affectedPersons`. The contract itself is then changed to inactive.

6 Retrieved from

The AIS is open for the submission of `PPE - semester assignment` until `2025-05-09T23:59:00+01:00`. You upload **a zip** archive (that is, not `rar`, not `tar`...) or any format other than `zip` to this upload location. Your archive will have the following structure:

```
assignment.z
ip└── src
    ├── company
    │   └── InsuranceCompany.java contracts
    │       AbstractContract.java
    ├──     AbstractVehicleContract.java
    │       InvalidContractException.java
    │       MasterVehicleContract.java
    │       SingleVehicleContract.java
    │       TravelContract.java
    │   objects
    │       LegalForm.java
    ├──     Person.java
    │       Vehicle.java
    │   payment
    │       ContractPaymentData.java
    │       PaymentHandler.java PaymentInstance.java
    ├──     PremiumPaymentFrequency.java
    │
    │
```

7 Rating

The evaluation will be carried out follows:

1. Automatic evaluation:
   (a) Verification of the structure of the archive submitted. If the structure is incorrect, the assignment will be scored 0 points.
   (b) Verification of UML class diagram conformance. Your solution must satisfy everything in the class diagram. If the class diagram is not met, the assignment will be scored 0 points. Your assignment may contain additional methods and attributes that are on the UML class diagram.
   (c) Evaluate the functional aspect of the submitted unit solution with the unit tests you have received (`RequiredTests`). If your solution fails any of these tests, it will be scored 0 points.
   (d) Evaluation of the functional aspect of the submitted solution by private unit tests. Based on the number (and significance) of the tests, a draft score for the assignment will be calculated.
   (e) Verification of your solution by an anti-plagiarism system. If your solution does not pass the check, it will be automatically scored 0 points. In this case, the further procedure will be in accordance with Article 13, para. 5 of the Study Regulations of FEI STU.
2. Oral interview:
   (a) If your solution did not receive a score of 0 in the automatic assessment process, you will participate in an oral debriefing of the assignment. The tutor will ask you a series of questions designed to test your detailed knowledge of your solution and the design principles you have used, as well as your knowledge of Java and basic OOP principles. The practitioner will evaluate your answers and use them to correct the proposed score from the automated testing. If the practitioner determines that you do not have a working knowledge of the code you submitted or have fundamental flaws in the material covered, your score will be 0 points. Cleanliness and readability of the code, adherence to Java conventions, etc. will also be assessed.