Professorship of Embedded Systems and Internet of Things
Department of Electrical and Computer Engineering
Technical University of Munich

Homework 3

Software Architecture for Distributed Embedded Systems, WS 2020/2021

M. Sc. Regnath, Dr.-Ing. Hamad, and Prof. Steinhorst

Submission Instruction

Please submit your solution as one **zip file** `HW3_lastname_firstname.zip`. In the root of the zip file, you must place directly all your main python files for each homework task. Use the same names of the files as they were specified in each question. Besides manual inspection, we also want to do some automatic testing, so please stick to the exact naming pattern. **There must not be any extra subfolders besides the module folder "smartl" in the root of your zip file. Also, you should not add any new files to this sub-folder.** If you want to include libraries, please use only standard Python libraries: https://docs.python.org/3/library/

? Exercise 3.1: Refactor Badly Written Code

Check the file `hw311.py` which contains an implementation of a PacketHandler which process a packet based on specified policy (i.e., SSH, TLS, IPsec). However, the code was badly written. Your tasks are:

1. Refactor the PacketHandler class in `hw311.py` using the strategy pattern. Your refactored code should handle the existing policies in a unified way and it should be possible to add new policies easily. Please submit `hw311_strategy.py`
2. Draw the UML class diagram for the refactored code. Please submit `hw312.svg`.

   Instructions:

   1. *You have to stick with the design guide that we have provided.*
   2. *You need to test your code. No test code will be provided for testing your solution.*
   3. *Use multiplicity everywhere it is needed in the UML class diagram.*

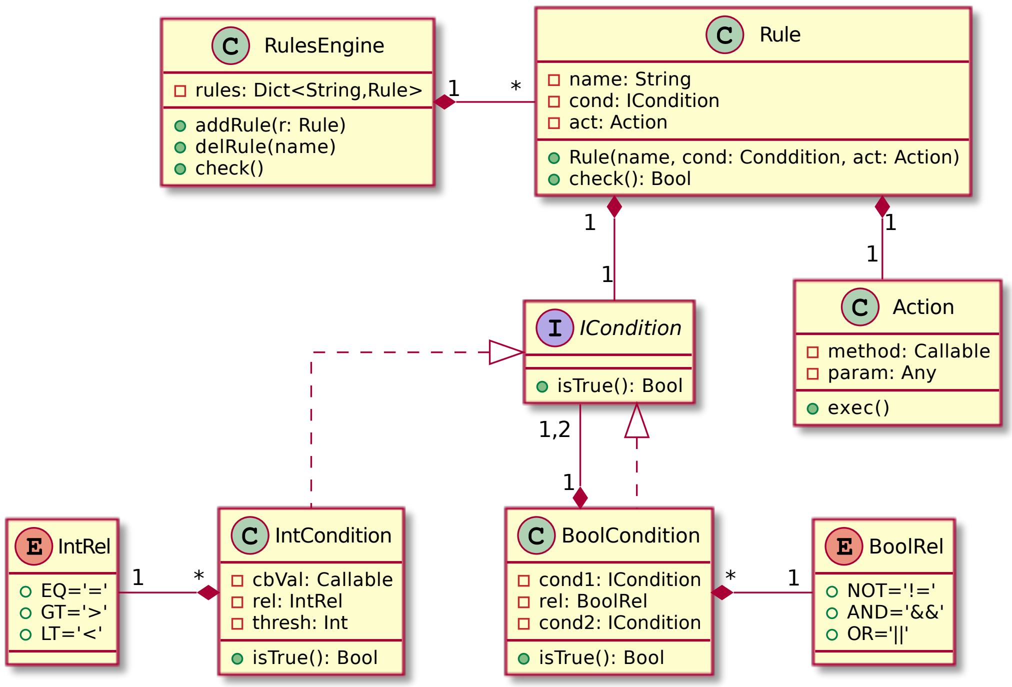
? Exercise 3.2: Refactor Smart Light

We now want to improve our smart light system to be more flexible and extensible. One of the problems is to account for future types of lights. The other problem is to give the user more freedom for defining conditions for triggering the lights. We will partly solve these problems by refactoring our code and introduce further patterns from the lecture. For the refactoring, try to implement all your code in the root folder files (e.g. `hw322_factory.py`). You can import specific classes from the "smartl" folder to make your code running.

1. Introduce the GoF factory pattern for the two different types of lights: *SwitchLight* and *DimmLight*. First, draw an UML with the relevant classes using the name *SwitchLightFactory* and a method `createLight(name: String)`. Please submit `hw321_factory.pu` and `hw321_factory.svg`.

2. Implement the factory pattern in Python for the given smart light code. The factory pattern should handle the creation of all lights and automatically register the lights in the *LightManager* via the method `addLight(l: Light)`. Please submit `hw322_factory.py`.

3. Refactor and extend the Trigger/IntCondition classes and implement the *IoT Rules-Engine* pattern. Each rule should be able to specify an *Action* via the Command Pattern and an arbitrary amount of *IntCondition*s that are combined with *and/or* logic operators. The following gives the UML diagram of the RuleEngine. Please submit `hw323_rules.py`.



4. Provide an example for a more complex rule that uses at least one Boolean operator. Give the Rule and the variables for the conditions (cond1, cond2) meaningful names such that it becomes clear what your rule should do. If in doubt, provide extra comments to clarify your intention. Import your rule engine from 3.2.3 and make it running. Provide at least two different input values (see code example in `hw324_rules.py`) and illustrate the working of the rule via print outputs. Submit `hw324_rules.py`.

5. Besides the covered patterns (Observer, Factory, Strategy, Rules Engine), which others patterns could be used to improve the smart light code? List two additional patterns, explain why each pattern is needed and which specific problem it solves. Give an example for the problem it solves. Submit `hw325-patterns.md`. Example solution: *"The observer pattern is needed to efficiently apply a config to several lights. It solves the problem of notifying only certain lights about user inputs that affect their state. For example, if a button is pressed, a certain but flexible amount of lights should change their state. Without the observer pattern every light would need to poll or process every event."*