Software Engineering

WS 2020/21, Assignment 02

Prof. Dr. Sven Apel
Sebastian Böhm, Christian Hechtl

Handout: 15.12.2020
Handin: 11.01.2021 23:59 CET

Organizational Section:

- The assignment must be accomplished by yourself. You are not allowed to collaborate with anyone. Plagiarism leads to failing the assignment.
- The deadline for the submission is fixed. A late submission leads to a desk reject of the assignment.
- We provide a project skeleton that must be used for the assignment. The skeleton can be downloaded from the CMS.
- The submission must consist of a *ZIP* archive containing only the project folder (i.e., the folder included in the provided skeleton)
- Questions regarding the assignment can be asked in the forum or during tutorial sessions. Please don't share any parts that are specific to your solution, as we will have to count that as attempted plagiarism.
- If you encounter any technical issues, inform us as early as possible
- FeatureIDE[1] is an extension to eclipse–java. It can be downloaded via the eclipse marketplace.

Task 1

Your task is to implement a configurable traffic light simulator using FeatureHouse[2]. The program will simulate an intersection that is managed by traffic lights. The variants of the simulation differ in the number of roads that merge at the intersection, whether there are pedestrian crossings present, and how the traffic lights behave. We provide a skeleton project that you have to use for your implementation. The following paragraphs describe how the traffic light simulator should behave and what has to be done to pass this assignment.

Passing Criteria To pass this assignment, your submission must fulfill the following criteria in addition to the criteria from the organizational section:

- You must not edit any of the provided files unless specified otherwise.
- Your solution must be implemented using FeatureHouse.
- Your solution must pass, at least, 50% of our private test cases.
- The code of your solution must be unit tested; your solution must pass all of your tests and at least 50% of your code must be covered.

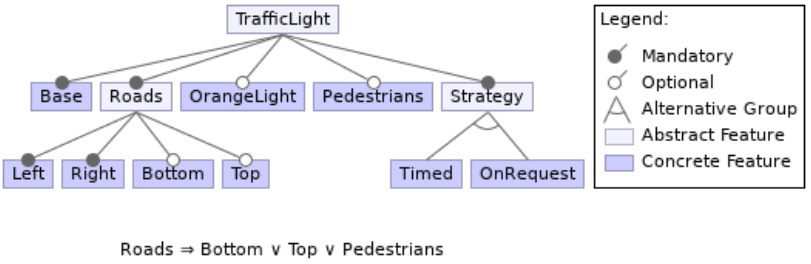
Implementation You must implement your solution based on the provided project skeleton. The skeleton has the following structure:

```
assignment01
├── model.xml
├── configs
│   ├── default.xml
│   └── ...
└── features
    ├── Base
    │   ├── intersection
    │   └── tests
    ├── Left
    │   ├── intersection
    │   └── tests
    └── ...
```

[1] http://www.featureide.com/
[2] http://www.fosd.de/featurehouse

The file `model.xml` contains the feature model for the project. This file must not be modified. The folder `configs` contains configurations for the project, that is, feature selections that define a variant of the simulation. You can activate one configuration at a time. FeatureIDE will automatically generate the sources for the currently active configuration. The folder `features` is where the actual source code for the project lives. It contains a subfolder for each feature. The implementation for each feature must be done in the respective subfolder. In the folders for the features *Base* and *Pedestrians*, you can find an abstract class `Intersection` that you have to extend with your functionality (see Listing Listing 1). *Make sure to read the documentation for this class in the actual source code in the provided skeleton!* This class is used by the class `Main`–the class that manages the simulation to instantiate your implementation of the traffic light simulator.

Feature Model You must implement all features that are modeled in the following feature diagram:



The following table explains each feature in more detail:

Table 1: Explanation for all features.

Base	Contains the base functionality that is included in all variants.
Roads	*Note that this is an abstract feature and has no code associated with it.* The subfeatures of this feature determine from which directions roads lead to the intersection. Each road automatically has a traffic light at the intersection. Traffic lights of opposing roads (e.g., left and right) always show the same light.
Left	Road from the left. This road is always present.
Right	Road from the right. This road is always present.
Bottom	Road from the bottom.
Top	Road from the top.
OrangeLight	Adds an orange light to traffic lights. Pedestrian traffic lights do **not** get an orange light. Traffic lights with an orange light take an additional 2 steps where the light is orange when switching from green to red or from red to green. When a road's traffic light is orange, the crossing lights must be red.
Pedestrians	Adds pedestrian traffic lights to each road that is present in the configuration. Pedestrian lights are green if the traffic light on the same road is red, and red otherwise. Also, pedestrians can now enqueue at the intersection. Pedestrians have an own queue per road. That is, per road and simulation step, one vehicle *and* one pedestrian can cross the intersection (if their respective traffic light is green).
Strategy	*Note that this is an abstract feature and has no code associated with it.* The subfeatures of *Strategy* determine how the traffic lights behave. Regardless of the strategy, at the start of the simulation, the traffic lights of the roads *Left* and *Right* always show green, and the traffic lights of *Bottom* and *Top* show red. Figure 1 visualizes the different strategies with and without orange lights.
Timed	The *Left*/*Right* roads start with a 10 steps long green phase. Then the traffic lights switch and the *Bottom*/*Top* roads get a 10 steps long green phase before the traffic lights switch again and the cycle repeats. Watch out how the addition of orange lights affect the duration of a complete cycle.
OnDemand	The *Left*/*Right* roads are green by default. When a vehicle queues at one of the *Bottom*/*Top* roads or a pedestrian queues at the *Left*/*Right* roads, the traffic lights switch to a 10 steps enduring green phase on the *Bottom*/*Top* roads, and then switches back. The *Left*/*Right* roads then need to have at least a 10 steps long green phase before the lights can switch again.

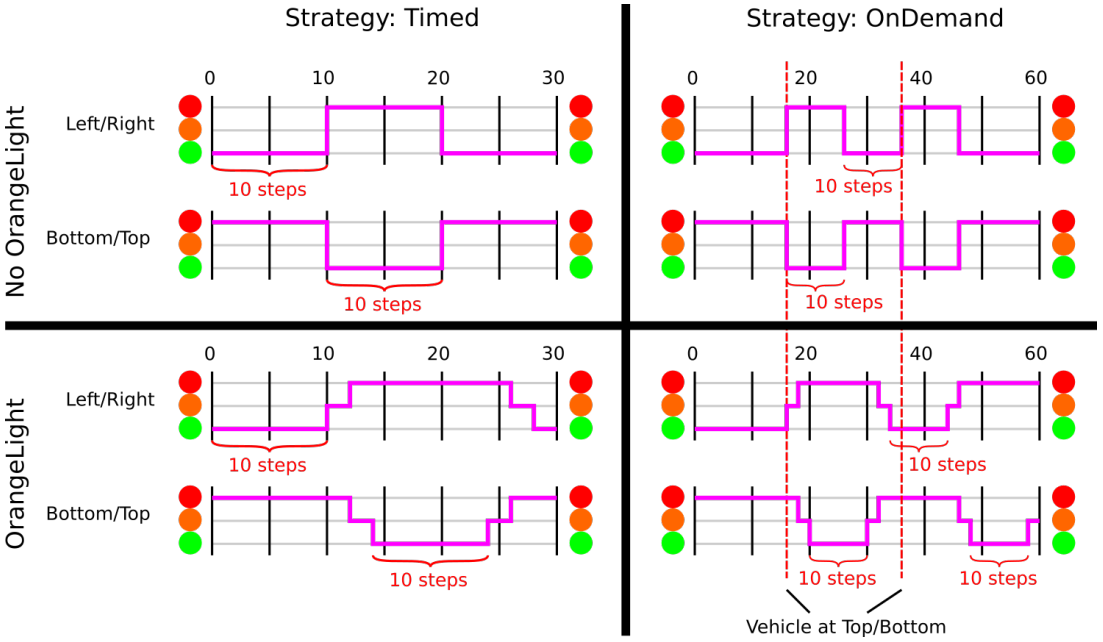


Figure 1: This figure shows the traffic light behavior for the different switching strategies when the orange light is enabled or not. Note that, in the lower right quarter of the figure, the vehicle coming from the bottom/top must wait until the 10-step green phase of the left/right roads is complete.

Simulation The simulation first creates an `Intersection` instance by calling `Intersection.createIntersection()`. You have to modify this function such that it returns an instance of your `Intersection` implementation. Then, the simulation is performed in steps with each step consisting of the following actions:

1. Vehicles (and pedestrians if configured) are enqueued via calls to `Intersection.enqueueVehicle()`.
2. The end of the step is signaled via a call to `Intersection.advanceTime()`.
3. The traffic lights are updated according to the configured strategy.
4. The traffic is simulated:
   - Only one vehicle (and one pedestrian) can pass per step and road.
   - A vehicle (or pedestrian) can only cross if the corresponding traffic light is green.

Listing 1: Interface for the intersection.

```java
public abstract class Intersection {

  /**
  * This method creates an instance of your intersection implementation.
  */
  public static Intersection createIntersection() {
    // TODO: return an instance of your implementation.
  }

  /**
  * Enqueue a vehicle on the given road.
  */
  public abstract void enqueueVehicle(Road road);

  /**
  * Enqueue a pedestrian on the given road.
  */
  public abstract void enqueuePedestrian(Road road);

  /**
  * Advances the time of the simulation by one step.
  */
  public abstract void advanceTime();

  /**
  * Get the state of the traffic lights as a string.
  */
  public abstract String getIntersectionState();
}
```

The simulation can ask for the current state of the intersection in the form of a string at any time by calling `Intersection.getIntersectionState()`. The string is built as follows depending on the active features:

- for each active road, there is a substring:
  - a single uppercase letter for the road (*Left* → `L`, *Right* → `R`, *Bottom* → `B`, *Top* → `T`)
  - a uppercase `V` (for vehicles)
  - a single lowercase letter for the traffic light color (*red* → `r`, *orange* → `o`, *green* → `g`)
  - the number of queued vehicles on this road
  - only if pedestrians are active:
    - a uppercase `P` (for pedestrians) followed by the number of queued pedestrians on this road
- the substrings for the roads must occur in the following order: *Left*, *Right*, *Bottom*, *Top*
- roads that are not included in the current configuration are left out
- the remaining substrings are separated by a single space

For example, given the configuration $\{TrafficLight, Base, Left, Right, Top, Pedestrians, Timed\}$ the initial state of the intersection is `"LVg0Pr0 RVg0Pr0 TVr0Pg0"`.

Running the Simulation The project skeleton includes a run configuration you can use to start the simulation. Once started, you can enter the following commands on the commandline:

`vehicle <road>` Enqueue a vehicle on `<road>`, where road can be one of `left`, `right`, `top`, `bottom`.
`pedestrian <road>` Enqueue a pedestrian on `<road>`, where road can be one of `left`, `right`, `top`, `bottom`.
`step` Advance the simulation by one step.
`state` Print the current state of the intersection.

Unit Tests To pass the assignment, you also have to provide unit tests for the assignment. The project skeleton is set up such that you can write unit tests with JUnit 5[3]. It also includes a run configuration you can use to run all unit tests for the current implementation. Unit tests also have to be implemented in a feature oriented way. Simply put the tests for each feature in a package called `tests` next to the `intersection` package in the implementation folder for that feature. The feature *Base* contains an example unit test that you can use as a template.

FeatureHouse FeatureHouse is similar to AHEAD in the sense that it also uses superimposition to compose features. With FeatureHouse, you can add new classes, add new members and functions to existing classes, override existing classes, members and functions and extend existing methods. There is no keyword `refines` in FeatureHouse, you can simply write the class declaration as usual. And instead of `Super.functionName()` you can use `original()` to refer to the original function definition.

WARNING: FeatureHouse only supports Java 5, so newer language features cannot be used. In case you accidentally use an unsupported language feature, eclipse most likely will give you a compile error.

We include a brief demo on how to use FeatureHouse in the introduction video to the assignment.

[3] `https://junit.org/junit5/`