Feedback on the Spectra development environment:

- Strengths:
  - Defines:
    We found the use of defines very helpful throughout development, as some of our guarantees were long and thus less readable. So the use of defines helped us code with easy maintenance and helped in readability.
  - Enums:
    Enums were a great tool to define meaningful types, making the spectra file more readable and consistent with the GUI enums.
  - Pre-defined Patterns:
    The library of the Dwyer Patterns was useful, as we included some of those that were explained in full at the lectures in our specification with ease.
- Weaknesses:
  - Use of next/prev in monitors:
    During developing our monitors system to keep track of waiting packages, we encountered difficulties using the *next* and *prev* keywords.
    Intuitively we expected *next* to implement the expected behavior for our monitor, where the system decides on the next state based on the current state, as such:

```
monitor boolean waitingPackageOutHouse1{
    !waitingPackageOutHouse1;
    G next(waitingPackageOutHouse1) = outHousePackages[0]   | (waitingPackageOutHouse1);
}
```

    Although as we found out – this behavior makes the monitor "late" by one state than expected, we later found out this issue can be solved with the use of the *prev* keyword, Although at a later stage of development we ditched the monitors system altogether in favor of a simpler assumption-based nature for monitoring requests.
  - Practical uses of triggers:
    We found it difficult to capture *complex* behaviors using triggers.
    This came up when trying to work on the 'Charge-when-needed' variant, we first tried using triggers, before converting to the current 'use-of-a-guarantee' version.
    We tried many versions, such as:

```
gar trig [true]*[atChargingStation] |=>
[!atChargingStation]*(([housePickupPackageOnly] | [warehousePickedup]){1,});
```

```
gar trig [true]*[atChargingStation] |=> [!atChargingStation]*([energy > 0]);
```

```
gar trig [true]*[atChargingStation][pickUpThatCounts]{0}[atChargingStation] |=>
false;
```

    And many more.
    Some were unrealizable and some just did not behave as expected.
    We found it useful to use triggers when our regular expressions regarded solely *on one aspect of the drone* (i.e. not mixing a few together).
    This became helpful especially in the 'Less Moves' variant.
  - DRY code:

Since our specification handles requests from multiple houses, we found it easy to implement some environment variables as arrays.
But those required usage of monitors and counters for each house and since Spectra does not have syntax support for monitor/counter arrays (that we found), or any other notation s.t we can “indice” those components we found repeating a lot of same-logic code.
Same goes for defines of locations (e.g. atHouse1, atHouse2...).
For example look at these guarantees:

```
// always eventually pick up a waiting package
gar pRespondsToS(waitingPackageOutHouse1, pickUpThisState = 1);
gar pRespondsToS(waitingPackageOutHouse2, pickUpThisState = 2);
gar pRespondsToS(waitingPackageOutHouse3, pickUpThisState = 3);
gar pRespondsToS(waitingPackageOutHouse4, pickUpThisState = 4);
```

Or those monitors:

```
monitor boolean waitingPackageOutHouse1{
    !waitingPackageOutHouse1;
    G waitingPackageOutHouse1 = outHousePackages[0]   | PREV(waitingPackageOutHouse1);
}
monitor boolean waitingPackageOutHouse2{
    !waitingPackageOutHouse2;
    G waitingPackageOutHouse2 = outHousePackages[1]   | PREV(waitingPackageOutHouse2);
}
monitor boolean waitingPackageOutHouse3{
    !waitingPackageOutHouse3;
    G waitingPackageOutHouse3 = outHousePackages[2] | PREV(waitingPackageOutHouse3);
}
monitor boolean waitingPackageOutHouse4{
    !waitingPackageOutHouse4;
    G waitingPackageOutHouse4 =outHousePackages[3] | PREV(waitingPackageOutHouse4);
}
```

In addition, during development, we found the *forall* keyword un-useful for some cases of using arrays since it has a *semantic meaning*.
A way to iterate through an array without the logical meaning of *forall* could also be useful.

- Counters:
  We were surprised to realize that the counter component has no enforcement over the system’s behavior by itself, i.e. it has to have a complementing guarantee.
  For example, that when *overflow* is set to *false* – it has no meaning on the system behavior without guaranteeing that the current counter value does not exceed it.
- Missing documentation of patterns:
  Outside of the select patterns shown in class, we could not understand the meaning of other patterns in the Dwyer Pattern library which maybe could have been helpful.