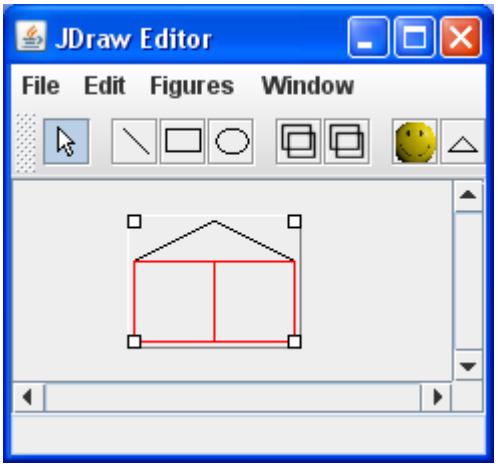
Worksheet: Decorator

In the lecture we learned when and how to apply the decorator pattern. We also compared this pattern with inheritance and we found many advantages of the decorator pattern over inheritance.

In this worksheet, we look at the dark side of the decorator pattern and examine three special aspects of this design pattern. If you want to reproduce the described problems, you can either use a decorator that you have already programmed for the assignment, or you can use the green decorator that we programmed last week in the lecture that is available on the AD.

Group Figures

Create a group figure and decorate it with a decorator. The following figure shows a group figure decorated with a border decorator.



If you now call the *Ungroup* operation on this selection, the group figure will not be split into its parts.

If you decorate a figure with the green decorator and then add a border decorator, do you expect to be able to remove the green decorator without removing the border?

Tasks:

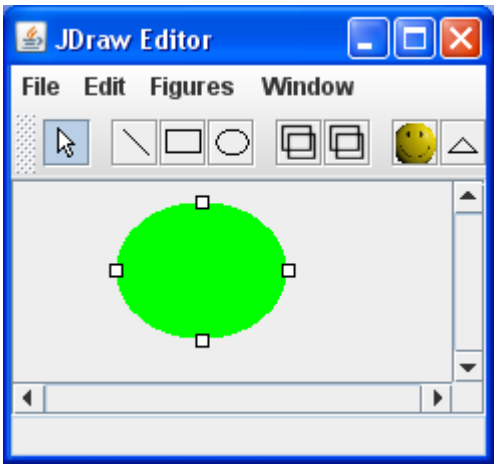
1. Problem analysis (alone or in pairs): 10 minutes
   - Think about why the group figure is not ungrouped.
   - Propose a solution that would allow you to unpack the decorated group figure or at least to access the group-specific functions (such as a `getNumberOfFigures`) or a solution that would allow to remove the green decorator without having to remove the border decorator.
   - Also, describe which result you expect on the screen after the decorated group has been ungrouped.

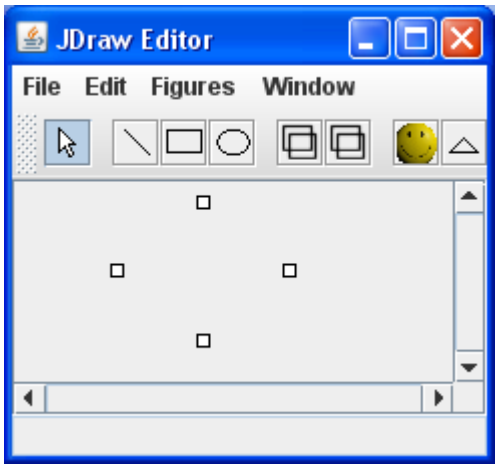
2. Expert round: 15 minutes
   Summarize your observations.
   Last week we compared the decorator pattern with inheritance. Compare these two approaches once again with respect to the aspect discussed in this worksheet.

3. Presentation:
   You have a time slot of 5 minutes to present your results. A short discussion with the class completes your presentation.
   Use the following structure for your presentation:
   a) Analysis of the problem (for the group figure and probably a generalization of the problem)
   b) Solution of the problem (for the group figure with a description of the behavior)
   c) Solution of the general problem
   d) Comparison between the decorator pattern and inheritance on the hand of this aspect.

Removing decorated figures

Decorate a figure and then delete it. You will see the following screen:





If a figure is deleted, it is removed from the model. All views are notified accordingly and remove the deleted figure with method `removeFromSelection` from their selection. For efficiency reasons, the displayed handles are kept in a separate list. The handles of a deleted figure must be removed from this list, and this is done in method `removeFromSelection` as well:

```
@Override
public void removeFromSelection(Figure f) {
	if (selection.remove(f)) {
		handles.removeIf(h -> h.getOwner() == f);
	}
}
```

Although the above code is executed upon deletion of a figure, the handles will not disappear.

Tasks:

1. Problem analysis (alone or in pairs): 10 minutes
   - Think about why the handles are not deleted in this situation.
   - Propose a solution how the decorator has to be fixed so that the code above works

2. Expert round: 15 minutes
   Summarize your observations and write down, how the decorator must be changed. Probably several solution are proposed in the expert round. Compare these variants.

3. Presentation:
   You have a time slot of 5 minutes to present your results. A short discussion with the class completes your presentation.
   Use the following structure for your presentation:
   a) Analysis of the problem (for the group figure and probably a generalization of the problem)
   b) Solution of the problem

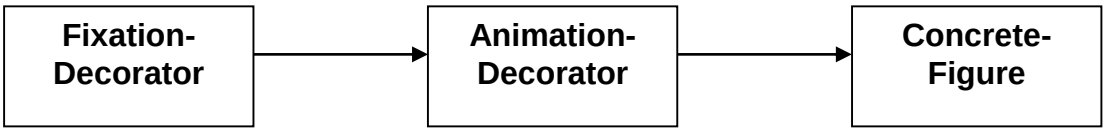
Animator

For this problem, we assume that we have programmed an `AnimationDecorator`. This decorator starts its own thread and regularly calls the `move` method of the decorated figure.

Let's assume that we also implemented a `FixationDecorator` that prevents the figure from being changed using methods `move` or `setBounds` or using the handles. Invocations of the `move` and `setBounds` methods are not forwarded to the inner figure in this decorator, i.e. they are implemented as follows:

```
@Override public void move(int dx, int dy) { }
@Override public void setBounds(Point origin, Point corner) { }
```

If you now decorate a figure with the `AnimationDecorator` and then apply the `FixationDecorator`, the figure continues to move around.



Tasks:

1. Problem analysis (alone or in pairs): 10 minutes
   - Explain why the `FixationDecorator` does not fulfill its task in this case.
   - What if the animator is defined as a subclass of the concrete figure and the fixator as a subclass of the animator?
   - Propose a solution that would allow the `FixationDecorator` to work as expected, even if it is added after the `AnimationDecorator`.

2. Expert round: 15 minutes
   Summarize your observations
   Last week we compared the decorator pattern with inheritance. Compare these two approaches once again with respect to the aspect discussed in this worksheet.

3. Presentation:
   You have a time slot of 5 minutes to present your results. A short discussion with the class completes your presentation.
   Use the following structure for your presentation:
   a) Analysis of the problem (and its generalization)
   b) Solution of the problem
   c) Comparison between the decorator pattern and inheritance on the hand of this aspect.