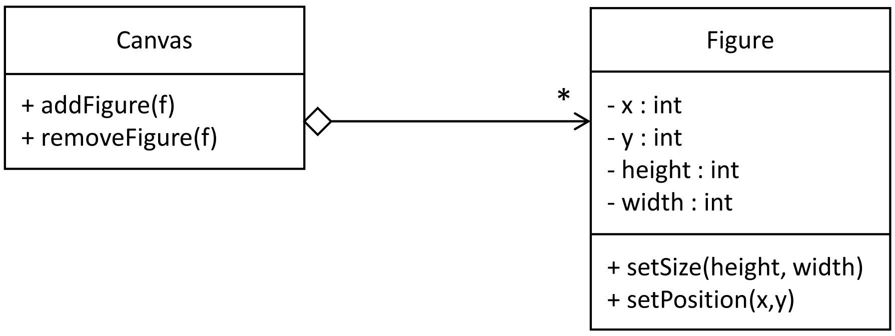
Design Patterns and Frameworks
Dipl.-Medieninf. Christian Piechnick
INF 2080
`christian.piechnick@tu-dresden.de`

Exercise Sheet No. 7
Software Technology Group
Institute for SMT
Department of Computer Science
Technische Universität Dresden
01062 Dresden

GoF Roundup

Task 1: Memorable Graphics

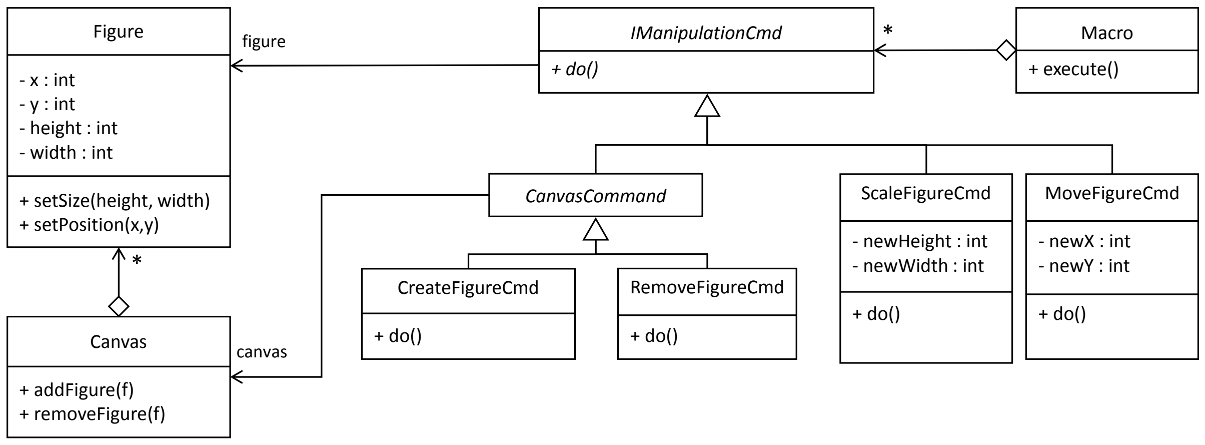
In this task we are going to design a graphical editing application. The application drawing area of our application consists of a *Canvas*, which contains *Figures.* Currently, it is possible to add and remove figures, as well as to change their position and size. The picture shows an small part of the class diagram of our application.



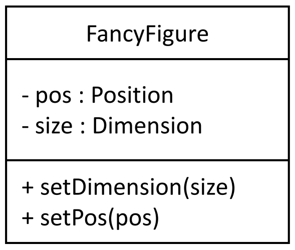
1a) **Task:** We want to be able to support a macro functionality. This means, that we want to record all possible manipulation operations (i.e., create figure, remove figure, move figure, scale figure) and want to re-execute them in a script. Therefore, we need to be able handle manipulation operations as first-class-citizen of the application.

What design pattern can be used to objectify the manipulation operations? Draw a class diagram for the operations *create, remove, scale and move*, that support the macro functionality.

Solution: The intent of the *Command* design pattern is to objectify operations of a class to handle execution of an operation directly. By applying the command pattern, commands can put in a queue, logged or undone.

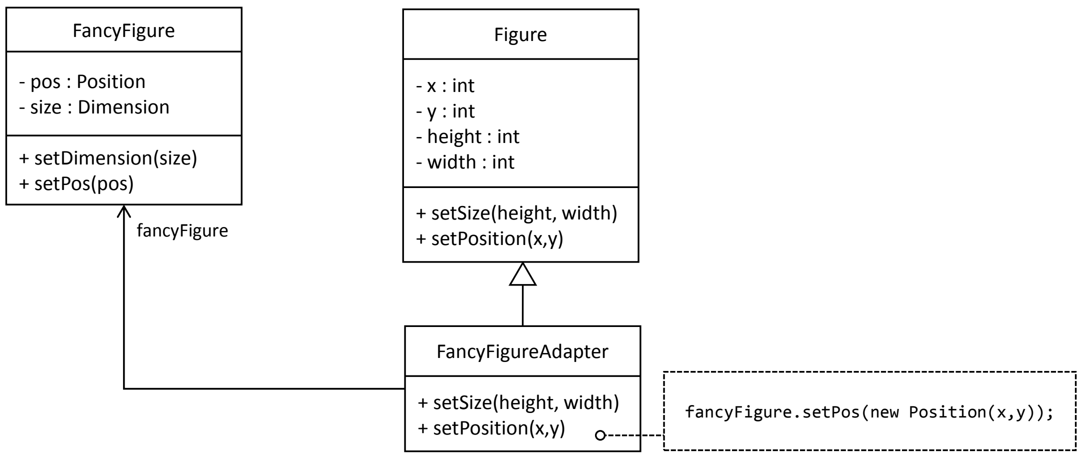


1b) **Task:** Let's consider we want to integrate a third-party figure library which offers *FancyFigures*. Unfortunately, the class FancyFigure has a different interface, then our figures.



What design pattern can we apply, to integrate the FancyFigure class in our application, without changing the Figures interface? Draw a class diagram!

Solution: We can use the Class Adapter pattern. We create a subclass of figure (FancyFigureAdapter) which maintains a reference to an object of the FancyFigure type. All methods (getters and setters) will be overwritten and forwarded to the FancyFigure object.

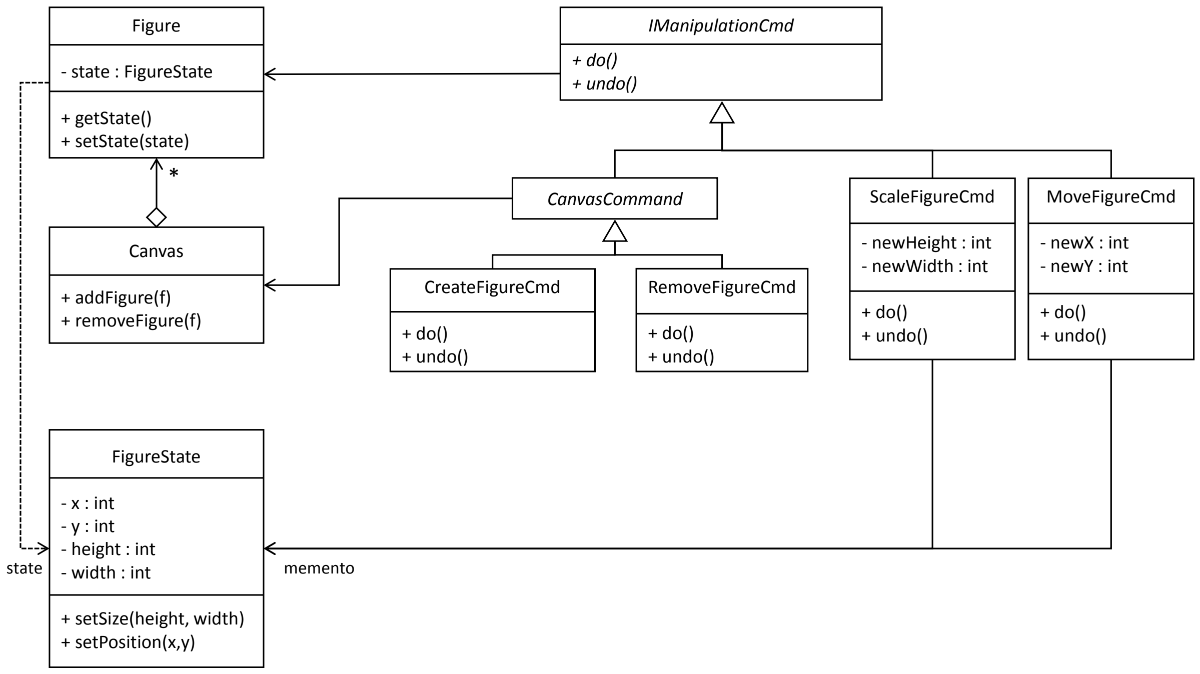


```
fancyFigure.setPos(new Position(x,y));
```

1c) **Task:** Our interactive application requires an undo mechanism so that tentative commands can be reverted and a redo mechanism so that such reversions can be undone again. This requires that some part of the application's state be stored and kept available for undoing modifications.

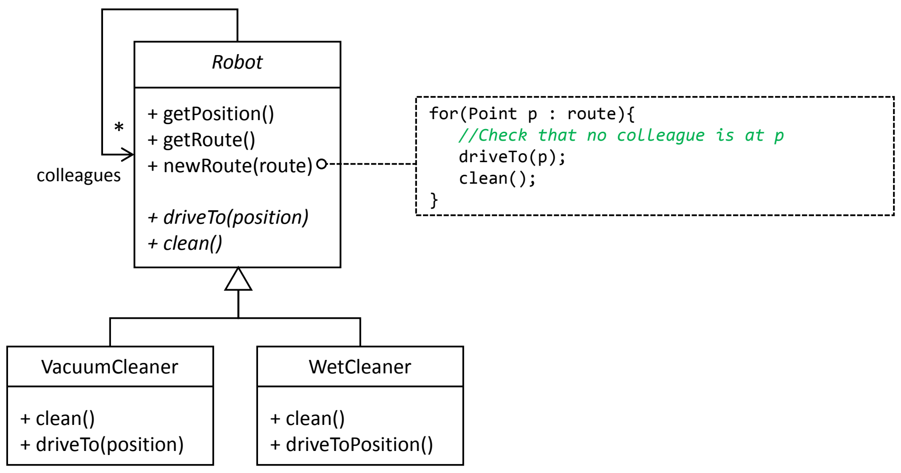
To implement undo, we need to store the state of the currently selected figure before performing a change. Then, we can use this information to perform an undo. However, explicitly accessing a figure's state breaks encapsulation. What would be needed is something that allows us to hand out state information without breaking the class's state. What design pattern can we use to solve this problem and how would we do this? Draw a class diagram.

Solution: The intent of the *Memento* design pattern is to externalize the internal state of an object to be able to restore the state, after changes were made. We can apply the memento pattern, therefore we can objectify the internal state of a figure (i.e., position and size) in a separate object. Thus, we can save the previous state of a figure.



Task 2: Cleaning Robots

Let's consider a modern production company that uses cleaning robots to keep the production halls nice and clean. Currently, two different kinds of robots are supported: Vacuum-Cleaning and Wet-Cleaning robots. As depicted in the figure, both robots share a same common functionality: When a new route is provided, they will drive to every point in this route (when there is no other robot currently) and clean that position. The driving and cleaning functionality however, depends on the concrete robot.



```
for(Point p : route){
   //Check that no colleague is at p
   driveTo(p);
   clean();
}
```

2a) **Task:** What design pattern was used, to model both shared and robot-specific behavior in one class hierarchy? What are the pros and what are the cons?

Solution: The Template Method design pattern was applied. The method `newRoute(route)` is the template method and the methods `driveTo(position)` and `clean()` are the hook methods.

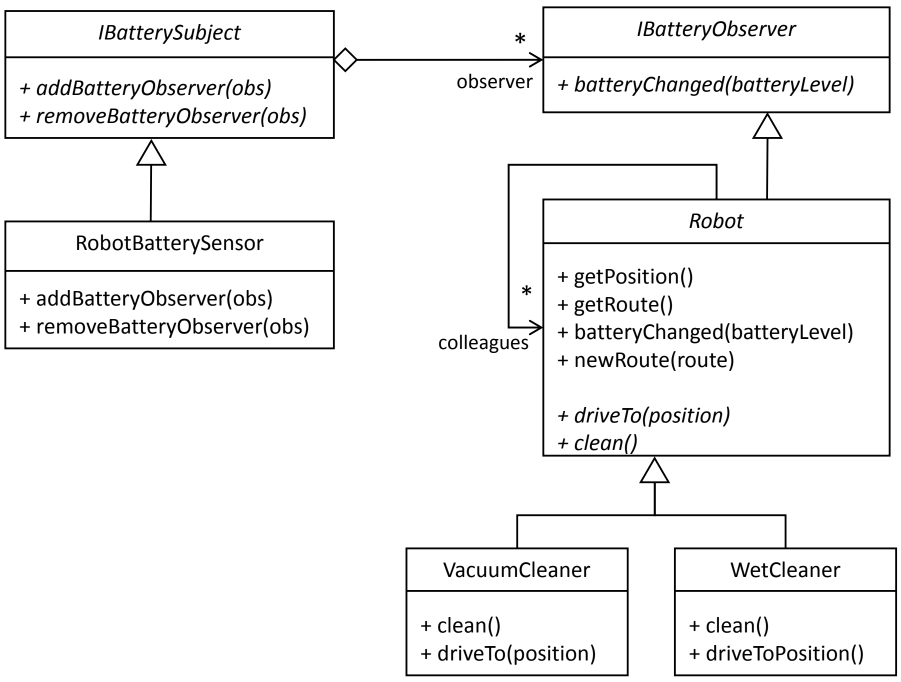
Pros: Template method enhances reuse (of the shared behaviour), and thus, reduces redundancy and increases maintainability.

Cons: If the concrete cleaning and driving behaviour has to be varied at runtime (e.g., because of plug-and-play hardware components), the object must be destroyed and recreated. In that case, Template Class might be a better solution.

2b) **Task:** One of the main problems of mobile robot is the battery that runs empty very quickly. The robot manufacturer provides you the source code of a battery sensor component. This component receives the remaining power with a frequency of 10Hz via a serial interface. The robot class should be notified, when the battery level changes. In future, other software components (e.g., a graphical monitoring interface) should be notified as well.

What design pattern can be applied to decouple the battery sensor from the robot class, potentially supporting more listeners of the battery level? Change the class diagram accordingly.

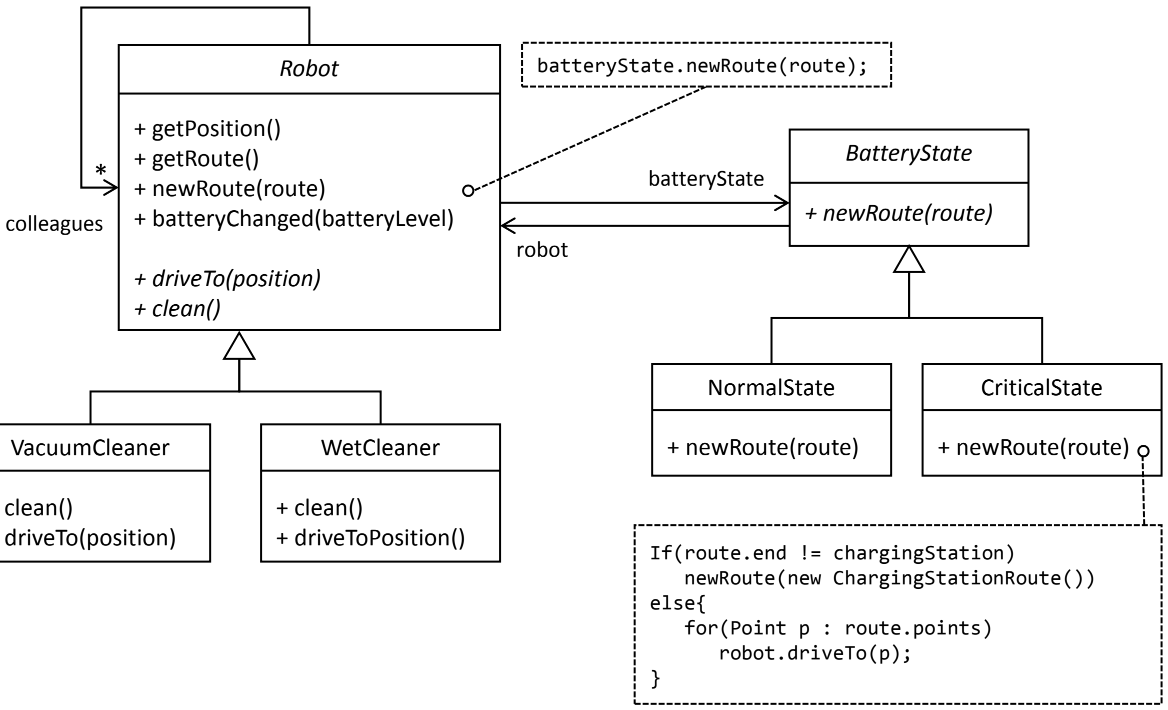
Solution: The observer pattern can be applied. It decouples the sender and receivers of events. Potential receivers can register themselves at the battery sensor. When the battery level changes, the battery sensor will notify every observer.



$\boxed{\text{2c)}}$ **Task:** Now, the robot is notified, whenever the battery level changes. When the battery level is below a certain threshold, it should stop its normal cleaning behavior and should drive back to it's charging station.

What design pattern can be used to model the different states (normal and critical) of the battery? How can the normal behaviour of the `newRoute(route)` method be changed, according to the state? Change the class diagram accordingly.

Solution: The *State Pattern* can be applied. The state pattern objectifies the state of an object in a separate class hierarchy. The robot maintains the corresponding state, depending on the battery level. When the robot is notified, that the battery level drops under a certain threshold, it will replace the normal with a critical state object. Furthermore, every call to the `newRoute(route)` method, is forwarded to the state. Depending on the concrete state, the robot will have different behavior (cleaning or driving back to the charging station).

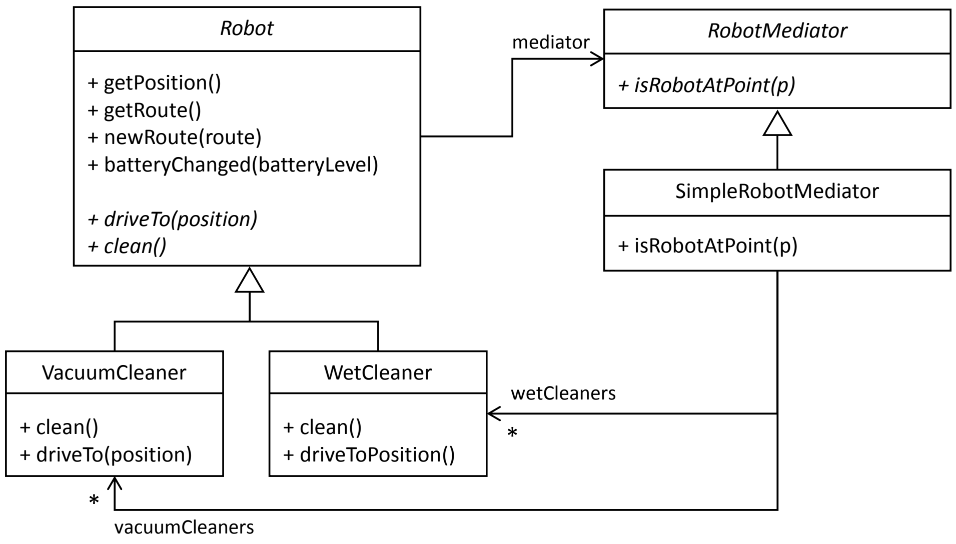


```
If(route.end != chargingStation)
   newRoute(new ChargingStationRoute())
else{
   for(Point p : route.points)
      robot.driveTo(p);
}
```

2d) **Task:** The company owns more than 100 robots, which manage a peer-to-peer connection to every other robot, to check their position and prevent collisions. Thus, more than 100000 network connections are active constantly. The local communication infrastructure cannot handle this amount of network traffic. The company asks you to change the current design of the application in a way, that not every robot has to manage a direct connection to all other robots.

What design pattern can be applied? Draw a class diagram.

Solution: The *Mediator Pattern* can be applied. Thus, every robot only maintains one connection to the mediator, and the mediator one connection to every robot. This results in only 200 instead of 100000 connections. Furthermore the position information can be stored in a cache and reused for multiple queries, instead of retrieving this information for every robot constantly.



2e) **Task:** What drawback does this design introduce? Sketch possible solutions.

Solution: The main drawback is, that the mediator introduces a single point of failure. When the mediator crashes, the robots cannot communicate. Furthermore, possibly expensive computation is concentrated on the mediator. Depending on the number of queries, the computing power of the mediator might not be enough to serve all requests in time.

One possibility could be to introduce redundancy, be hosting multiple mediators simultaneously. In order to hide this complexity, a gateway-proxy can be used, that manages to load balancing.