TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute of Software and Multimedia Technology, Software Technology Group

WS2019/20 – Design Patterns and Frameworks

Role-Based Modeling for Design Patterns (Part 2)

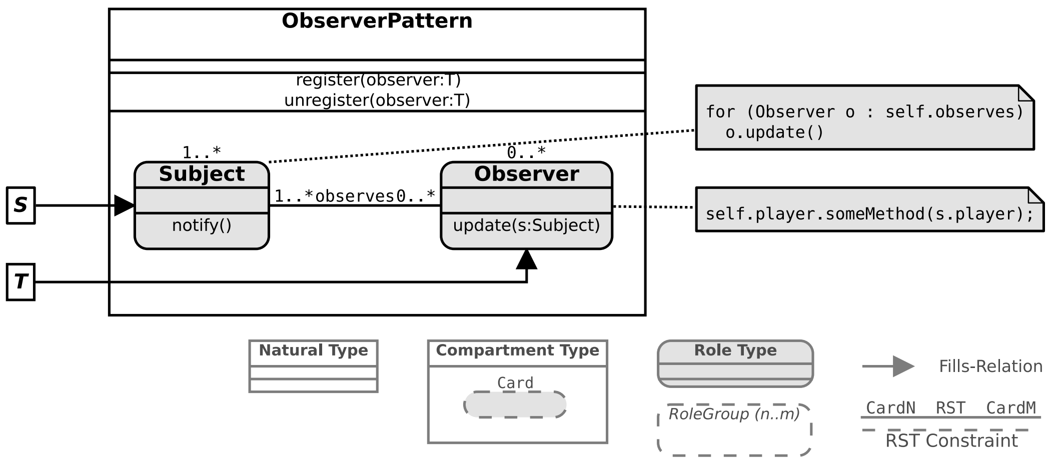
Professor: Prof. Dr. Uwe Aßmann
Lecturer: Dr.-Ing. Sebastian Götz
Tutor: Dr. rer. nat. Marvin Triebel

Task 1 Role-based Design Pattern Catalog

This exercise focuses on finally applying the *Compartment Role Object Model* (CROM) [3] to formalize and compose the various design patterns discussed in this course. In detail, the first task is to create role-based models of various design patterns, ultimately, creating a role-based design pattern catalog similar to Riehle's design pattern catalog [4].

a) Design a *compartment type* and role model for the `Observer` design pattern.

Solution: The `Observer` design pattern encompasses two roles, i.e., `Subject` and `Observer`. The corresponding CROM model is depicted below.

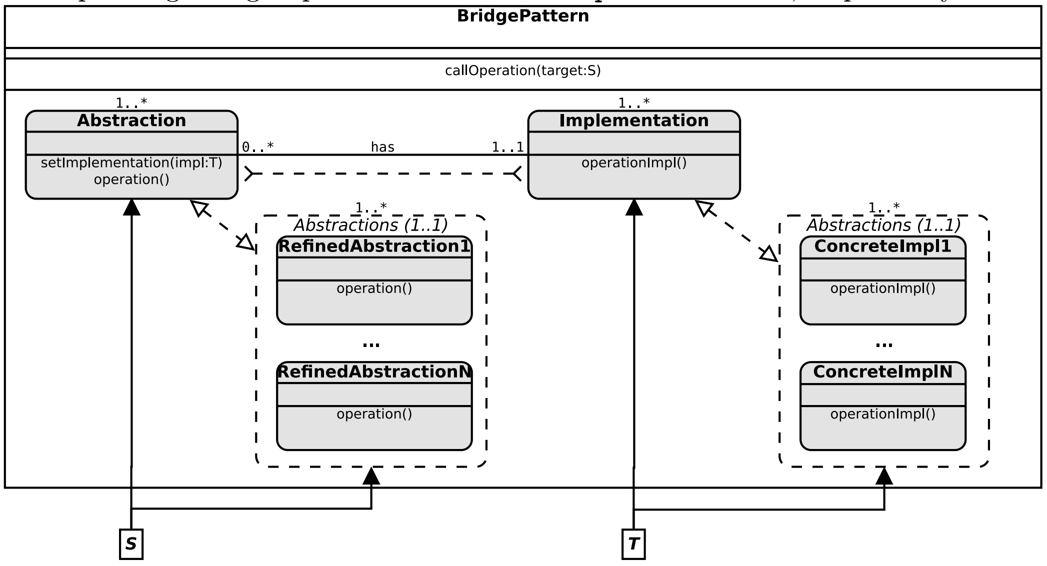


```
for (Observer o : self.observes)
  o.update()
```

```
self.player.someMethod(s.player);
```

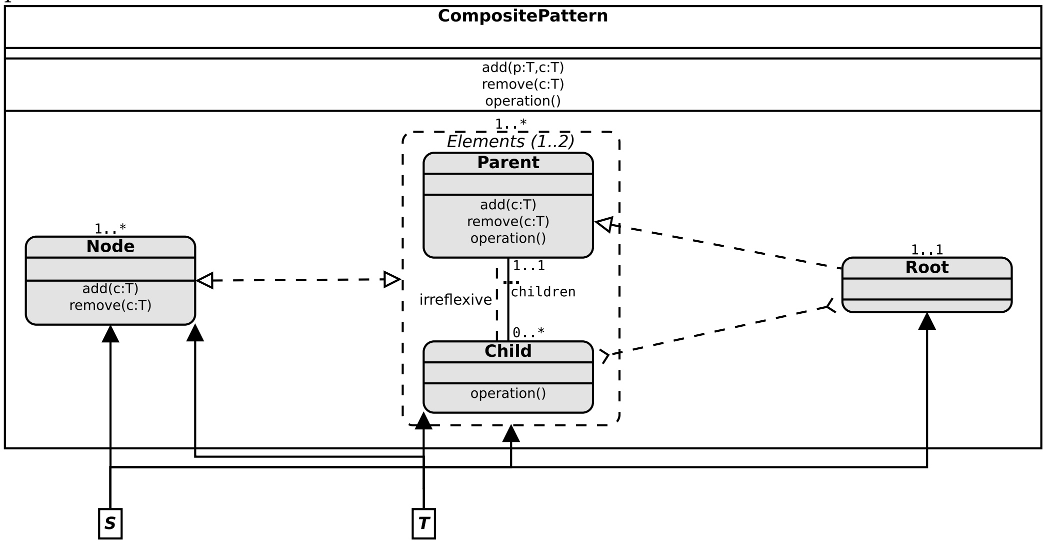
b) Design a *compartment type* and role model for the `Bridge` design pattern.

Solution: This design pattern involves two role hierarchies, i.e., one for abstractions and one for implementations, as shown below. There is a *role-prohibition* between the `Abstraction` and the `Implementation` role type. Moreover, if an object plays the `Abstraction` or `Implementation` role, then it must play a role in the corresponding role groups `Abstractions` or `Implementations`, respectively.



c) Design a *compartment type* and role model for the `Composite` design pattern.

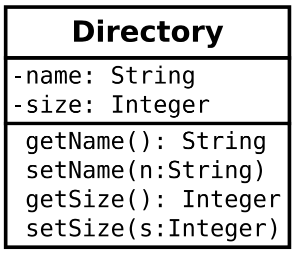
Solution: This design pattern establishes a graph or tree structure by means of `Parent` and `Child` role types, which are linked with an *irreflexive* or *acyclic* relationship *one-to-many.* Each `Node` must be either a `Parent` or a `Child` (or both), as indicated by the `Elements` role group. Moreover, there must be one `Root`, which is prohibited to be a `Child` but must be a `Parent`.

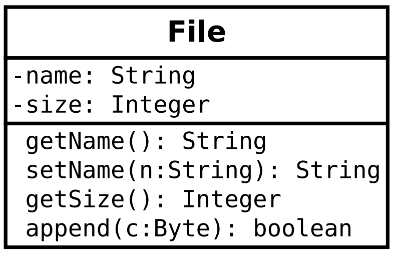


Task 2 Applying and Composing Design Patterns

Applying a role-based design pattern entails assigning the domain classes to play the appropriate role types from the design pattern.

In detail, this task focuses on a **file system** including *files* and *directories*. Both files and directories have a modifiable name and return their size in Bytes. While files return the size of their content, directories should return the accumulated size of all containing directories and files. The properties of the `File` and `Directory` classes are depicted below, whereas both are considered natural types.

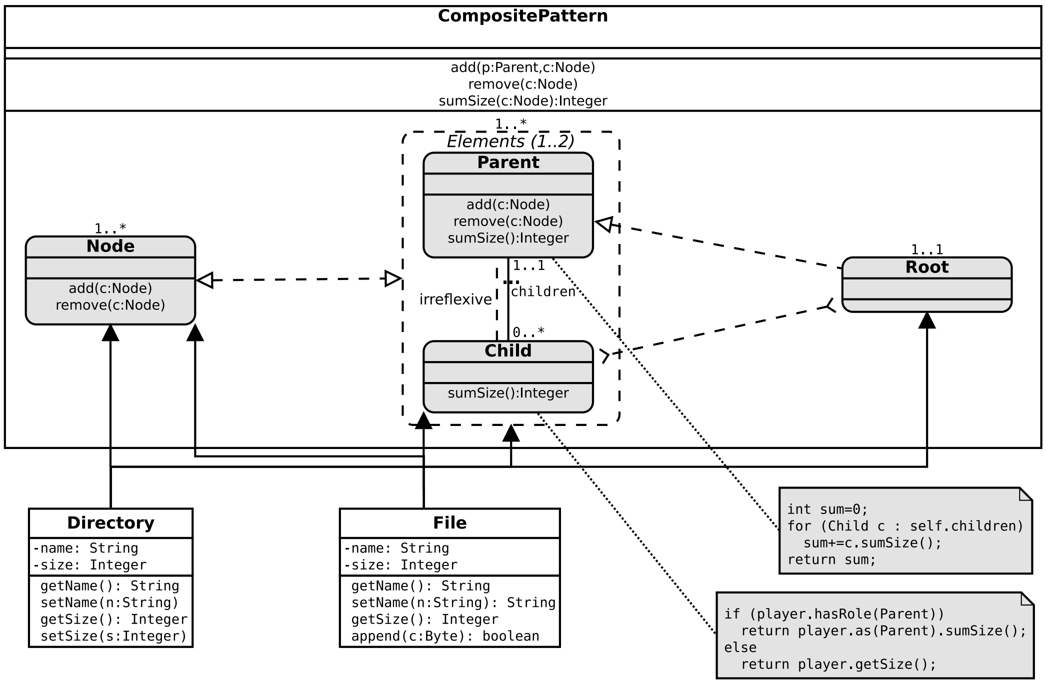




Thus far, the `Directory` class does not encompass files and directories. Yet, neither `Directories` nor `Files` do notify changes to their name or size to their parent directories. Moreover, size changes should be propagated up to the root of the directory hierarchy.

a) First, apply the role-based `Composite` design pattern to organize the file system's structure, such that `Directories` can contain both `Files` and other `Directories`.

   Solution: To apply the `Composite` design pattern, we simply declare that the `Directory` fills all role types, whereas the `File` only fills the `Child` and `Node` role types. Moreover, the `sumSize()` operation must be implemented in `Parent` and `Child`. The `Parent` simply iteratively calls `Child.sumSize()` to summarize its size. In contrast, the `Child` must dispatch to the `Parent` role, if it is simultaneously played, and otherwise calls `getSize()` on its `player`. This ensures that the size is accumulated throughout all levels of the tree. The resulting application is depicted in the following CROM model.

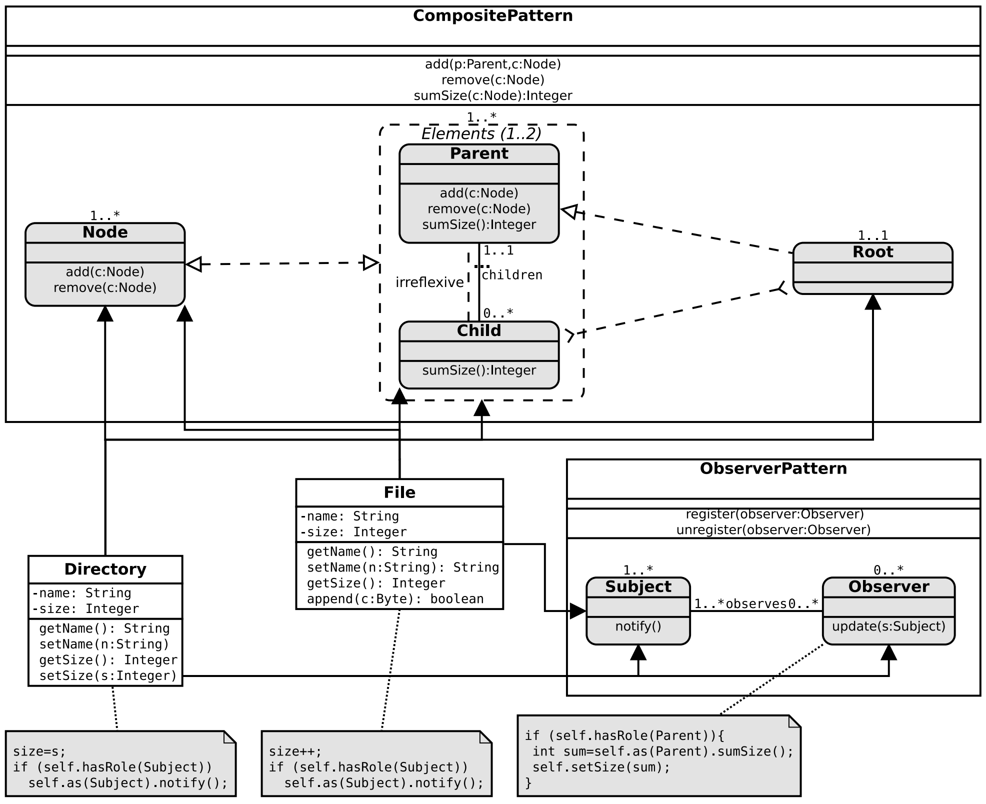


```
int sum=0;
for (Child c : self.children)
  sum+=c.sumSize();
return sum;
```

```
if (player.hasRole(Parent))
  return player.as(Parent).sumSize();
else
  return player.getSize();
```

b) Second, apply the role-based `Observer` design pattern to propagate any size changes of a `File` up to its parent `Directory`.

Solution: Adding the `Observer` design pattern, requires to let the `Observer` role type be filled by `Directory` and the `Subject` role type by both `Directory` and `File`. In addition, the `update()` method is implemented to fetch the `Parent` role from the `Directory` playing the `Observer` role and call `sumSize()` to accumulated and `setSize()` to update the directory's size. Finally, the `setSize()` method of the `Directory` and the `append()` method of the `File` must be augmented to invoke the `Subject.notify()` method after their execution. Notably though, this change can also be done by implementing both methods in the `Subject` role type. Nonetheless, the main downside of this approach, is the separate maintenance of both the composite structure and the `observes` relationship, which can easily lead to inconsistencies between the two.



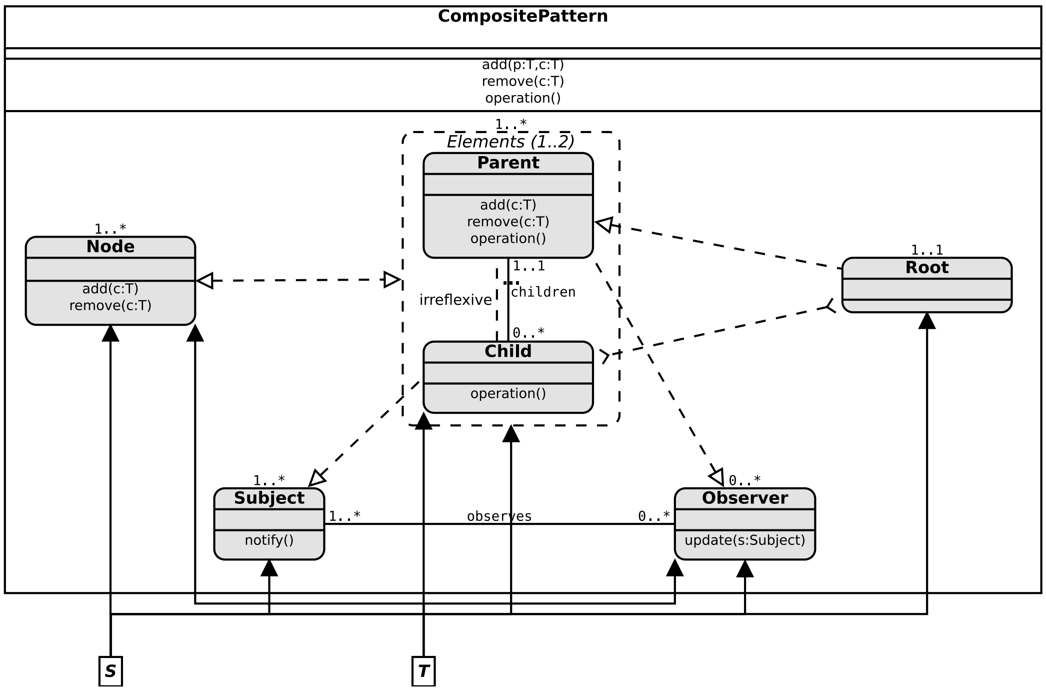
```
size=s;
if (self.hasRole(Subject))
  self.as(Subject).notify();
```

```
size++;
if (self.hasRole(Subject))
  self.as(Subject).notify();
```

```
if (self.hasRole(Parent)){
 int sum=self.as(Parent).sumSize();
 self.setSize(sum);
}
```

c) Combine the role models of both the `Observer` and the `Composite` compartment type by means of *role constraints*, to create a sound combination of both design patterns in a new compartment type.

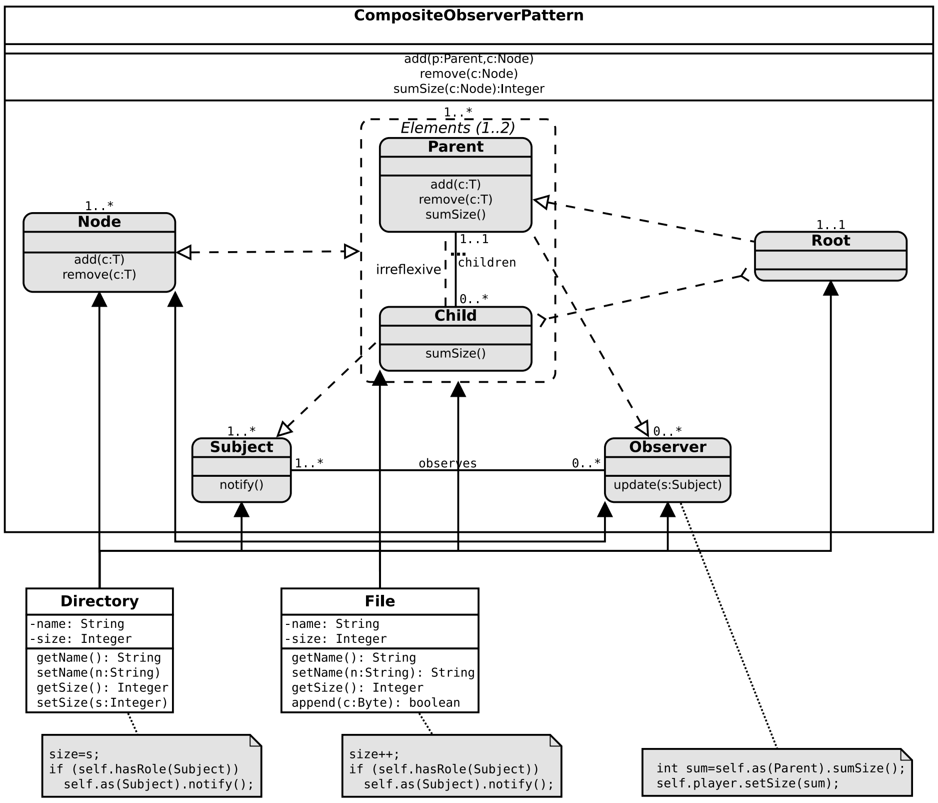
Solution: Utilizing role constraints it is rather simple to combine the role models of the `Composite` design pattern and the `Observer` design pattern. In detail, a new compartment type is created containing all role types, relationships, and role constraints of both role models. Then both role models can be linked, by declaring that any `Child` must be a `Subject` and any `Parent` must be a `Subject`, whereas the `observes` relationship should mirror the `children` relationship.[1] The resulting CROM model for the combined design pattern is as follows:



[1]An advanced version of CROM introduces *inter-relationship* constraints for exactly that purpose.

d) Reapply the combined design pattern to the `File` and `Directory` classes of the file system. Compare this solutions to their separate application.

Solution: The application of the combined `CompositeObserver` design pattern, is done similar to the previous separate application, i.e., by defining the fulfillments of the `Directory` and `File` classes accordingly. In contrast, the combined implementation, can internally ensure that for each `Child` added to a `Parent` a corresponding `Subject` is added and observed by the parent's `Observer` role. Consequently, this guarantees that the `Parent` role exists in the `Observer.update()` method. Nevertheless, the main benefit of this approach, is the congruent management of both the composite structure and `observes` relationship, which can be guaranteed by only exposing the `add()` and `remove()` methods of the `Parent` role type to its player. In conclusion, only the `setSize()` method of the `Directory` and the `append()` method of the `File` must be augmented to invoke the `Subject.notify()` method after their execution.



```
size=s;
if (self.hasRole(Subject))
  self.as(Subject).notify();
```

```
size++;
if (self.hasRole(Subject))
  self.as(Subject).notify();
```

```
int sum=self.as(Parent).sumSize();
self.player.setSize(sum);
```

References

[1] Dirk Bäumer, Dirk Riehle, Wolf Siberski, and Martina Wulf. The role object pattern. In *Washington University Dept. of Computer Science*. Citeseer, 1998.

[2] Erich Gamma. Extension object. In *Pattern languages of program design*

3, pages 79–88. Addison-Wesley Longman Publishing Co., Inc., 1997. URL https://www.ecs.syr.edu/faculty/fawcett/handouts/CSE776/PatternPDFs/ExtensionObject.pdf.

[3] Thomas Kühn, Böhme Stephan, Sebastian Götz, and Uwe Aßmann. A combined formal model for relational context-dependent roles. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, pages 113–124. ACM, 2015. doi: 10.1145/2814251.2814255. URL http://dl.acm.org/citation.cfm?id=2814255.

[4] Dirk Riehle. A role-based design pattern catalog of atomic and composite patterns structured by pattern purpose. Ubilab Technical Report 97.1. 1., Union Bank of Switzerland, Zürich, Switzerland, 1997. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.496.7976&rep=rep1&type=pdf.

[5] Yannis Smaragdakis and Don Batory. Implementing layered designs with mixin layers. In *European Conference on Object-Oriented Programming*, pages 550–570. Springer, 1998. URL citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.23.7182&rep=rep1&type=pdf.