Observations on the Observer Pattern

Christian Köppe
Hogeschool Utrecht
Institute for Information & Communication Technology
christian.koppe@hu.nl

September 26, 2010

Abstract

A group of students who should have been familiar with basic design principles and MVC all failed to implement the Observer design pattern correctly while at the same time violating different design principles. This paper discusses what went wrong and why it probably went wrong. We finally formulate possible consequences for teaching the Observer pattern and for teaching design patterns in general.

1 Introduction

Design patterns are recognized as common knowledge in the software engineering field [IEE04]. They help to implement basic OO-principles and offer a lot of advantages as e.g. best practices, a common vocabulary, and approved solutions. From a pedagogical perspective, they help in teaching software design and modeling [Ras97] and should be integrated in computer science curricula [AMBC98].

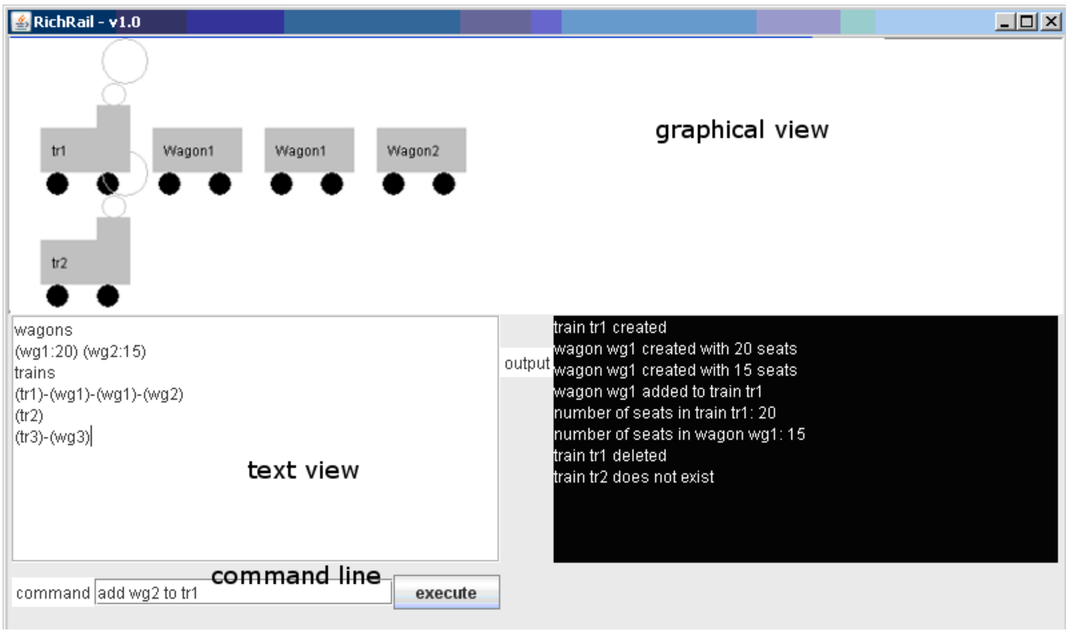
While some research and case studies exist which discuss how to teach design patterns (e.g. [CL99, MA04]), there seems to be no common best approach. Furthermore, the integration and consistency of a design and the concrete implementations of this design seem to be overlooked. In our experiences, these solutions often lack conceptual integrity (as introduced by [Bro95]) and violate some basic principles while trying to implement others.

This work presents a case study based on a students assignment. The design decisions of the students are shown and compared with an expected solution and show some unexpected differences. Especially the implementation of the Observer design pattern led to problems and violations of OO-principles. We will then elaborate why we think the students failed to understand and implement this pattern and to still follow the OO-principles previously taught to them. Finally, consequences are discussed of how the teaching of this pattern (and design patterns in general) could be improved to achieve that students better understand the usage of patterns and are able to deliver more correct solutions which also follow the basic OO-principles.

2 Context

A group of students who should have been familiar with OO and MVC were taught design patterns. This was done by telling them the main purposes of design patterns (as e.g. collection of best practices, common vocabulary etc.) and showing some examples. They were given an assignment that would require the exploration of common design patterns (from books like [GHJV95] and other sources) and the use of their OO knowledge. Most of the design patterns were applied correctly, but they all failed in applying the Observer pattern correctly. What went wrong and why?

The students were given the source code from an existing simple Java application for the administration of trains (locomotives) and wagons (rail cars) which they had to improve. The assignment included a screenshot from a GUI prototype, shown in figure 1. There were a couple of improvements and new functionality described which had to be implemented by the students.



```
wagons
(wg1:20) (wg2:15)
trains
(tr1)-(wg1)-(wg1)-(wg2)
(tr2)
(tr3)-(wg3)
```

```
train tr1 created
wagon wg1 created with 20 seats
wagon wg1 created with 15 seats
wagon wg1 added to train tr1
number of seats in train tr1: 20
number of seats in wagon wg1: 15
train tr1 deleted
train tr2 does not exist
```

```
add wg2 to tr1
```

Figure 1: GUI prototype of solution

One of the new requirements included different representations of the domain model (the trains and waggons created), as described in following assignment fragment:

> They (the end-users) also want the possibility of displaying their trains in different ways.
> ...
> Furthermore these requirements have to be included in the new software:
> - The display of the existing trains incl. wagons and wagontypes has to be interchangeable, meaning that also other displaytypes should be easily integratable
> - ...

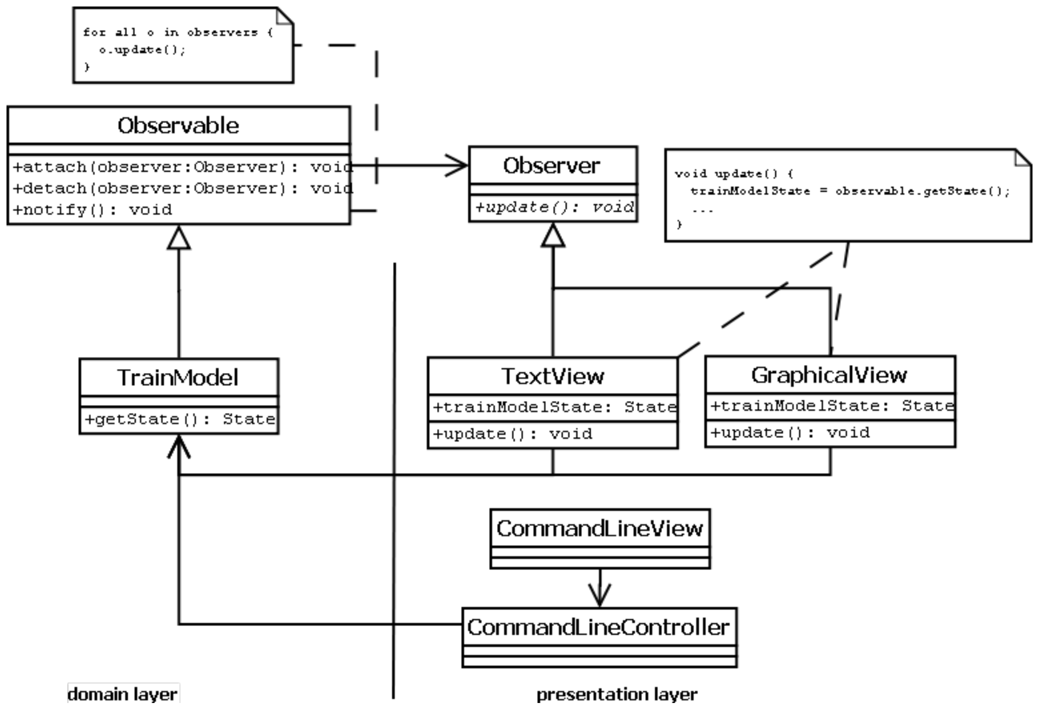
Gamma et. al state that the Observer pattern is applicable "when a change to one object requires changing others, and you don't know how many objects need to be changed." [GHJV95]. In this assignment, changes to the model require changes to the representations and it is not known how many representations need to be changed. The Observer pattern therefore offers the solution to this problem (among other patterns which are not further discussed here).

The following grading criteria were communicated to the students:

- Required functionality implemented
- Design
  - Good structured code (components, layers etc.)
  - Principles followed (high cohesion, low coupling)
  - Sensitive and reasonable usage of design patterns
- Comprehensible coding, no 'code smells' (as described in the refactoring lesson)
- Not graded: performance, fancyness

It was expected that the students implement the domain model as observable and the two representations (text and graphical) as observers as shown in figure 2. Actually, MVC (and therefore controllers) is not needed for this part of the assignment as there is no user interaction between these representations and the model.
MVC can well be used for the implementation of the command line interface, as this interacts with the model. This is also shown in figure 2.

The students were also expected to follow design principles which have been taught to them:


```
for all o in observers {
  o.update();
}
```

```
void update() {
  trainModelState = observable.getState();
  ...
}
```

Figure 2: Expected solution (only relevant parts shown)

- **Loose Coupling** - designing so that connections among different parts of the program are held to minimum [McC04]
- **The Rules of Layering** - Layers should only be aware of the next lower layer and call-backs are used for bottom-up communication to minimize coupling between different layers [BMR$^+$96]
- **Information Hiding** - hide design and implementation decisions from the rest of a program in one place [Par72, McC04]
- **Separation of Concerns** - different or unrelated responsibilities should be separated from each other within a software system [Dij82, BMR$^+$96]

Remarkably, 4 of the 5 groups tried to implement the Observer pattern, but none of them got it completely right as expected. The last group tried to implement it as well but got stuck, so they decided to do it without using the Observer pattern.

Conventions

In some books the two important abstractions contained in the observer pattern are observer and subject. In this work the term observable is used instead of subject, but they can be seen as synonyms.

3 Analysis of the groups solutions

Group 1

Figure 3 provides an overview of the relevant parts of this groups solution. As one sees they chose to use MVC and introduced controllers for the different views, even if they are not necessary. Each controller has a connection to either a textual view or a graphical view (via *JTrain*). The controller also knows the model. Besides introducing controllers as observers instead of the graphical representations of the model, this looks okay.
Interesting is, that they also have events which occur at the model (the class `PoolChangeEvent`). These events are also known to the controller, which is unusual. To determine why this is modeled

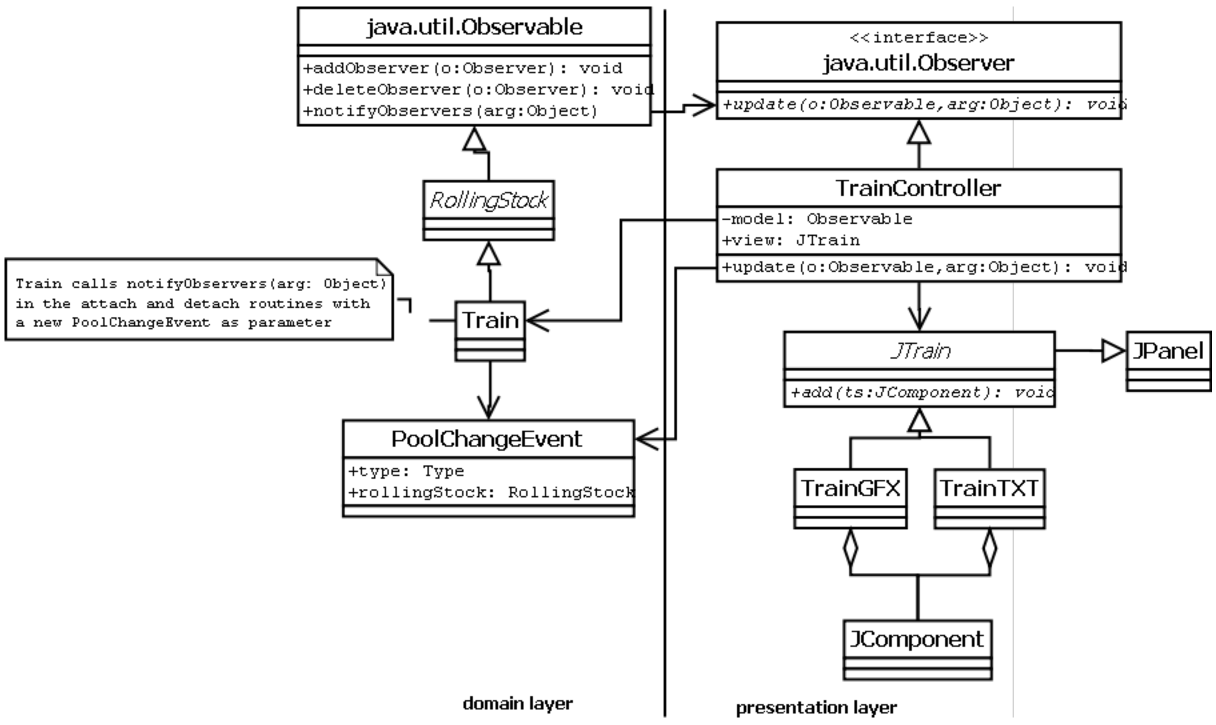

java.util.Observable
+addObserver(o:Observer): void
+deleteObserver(o:Observer): void
+notifyObservers(arg:Object)
RollingStock
Train calls notifyObservers(arg: Object) in the attach and detach routines with a new PoolChangeEvent as parameter
Train
PoolChangeEvent
+type: Type
+rollingStock: RollingStock
<<interface>>
java.util.Observer
+update(o:Observable,arg:Object): voi
TrainController
-model: Observable
+view: JTrain
+update(o:Observable,arg:Object): void
JTrain
+add(ts:JComponent): voic
JPanel
TrainGFX
TrainTXT
JComponent
domain layer
presentation layer

Figure 3: Implementation group 1 (only relevant parts shown)

this way we will take a look at some code fragments.

This group used the Observer pattern implementations provided in the java.util package. Class `Train` is implementing `java.util.Observable` (via Rollingstock) and calls `notifyObservers` on different places after setting `super.setChanged()` as following code fragment shows.

```
protected void attachRollingStock(RollingStock rs)
{
  rollingStockItems.add(rs);
  setChanged();
  notifyObservers(new PoolChangeEvent(Types.add, rs));
}
```

The TrainController knows the model, as can be seen in figure 3. Surprisingly he doesn't do anything with it. Instead the event which had taken place is also sent to the Observer (in parameter *Types.add*), telling him what happened and also sending the just added rollingStock to the Observer. This increases the coupling between Observer and Observable. The method `notifyObservers(Object arg)` calls update(Observable o, Object arg) on all connected observers. This is also the only method in the interface `java.util.Observer`. This implies that one always has to send the Observable and propably some arguments (encapsulated in an object) which can be misleading for students. To decrease the coupling between Observer and Observable, a null-object should have been used.

The TrainController implements `java.util.Observer`, and therefore the `update(Observable o, Object arg)`-method, which is shown in following code fragment.

```
30:  @Override
31:  public void update(Observable o, Object arg) {
32:
33:    PoolChangeEvent event = (PoolChangeEvent)arg;
34:    Train t = (Train)o;
35:
36:    switch (event.type) {
37:      case add:
38:        view.add((JComponent)uiFactory.buildJTrainComponent(event.rollingStock));
39:        break;
40:      case removed:
41:        view.remove(event.rollingStock.getId());
42:        break;
43:    }
44:    view.invalidate();
45:    view.validate();
46:  }
```

There are a few interesting points here:

1. The Observable is sent but not used here (despite that it's cast in line 34). Probable reason for this is that this is the first parameter and it has to be sent. This Observer does in fact nothing with the Observable.

2. The second parameter is well used, but this has nothing to do with the Observable. It is instead the (encapsulated) event which occured on the Observable. Therefore, nothing of the Observable is used here and the Observer does not acquire information from the model.

3. The (unnecessary) casting of the JTrainComponent into a JComponent in line 38 which is added to another JComponent (the `view`) led to the examination of the `view`-object. This is of type JTrain, an interface. The concrete implementation of this object is using this interface and extends JComponent. The implementation of the `add`-method looks like this:

   ```
   @Override
   public void add(JComponent trainSegment) {
     jpScrollContent.add(trainSegment);
     jpScrollContent.invalidate();
     jpScrollContent.validate();
   }
   ```

   This shows that a JComponent-representation of the domain model object is added to a container, leading to a second representation of the model in the view layer of the application! This way it's possible to change the model in the view layer bringing it out of sync with the domain model (and there's no way to repair this beside removing it from the view layer model).

Conclusion group 1

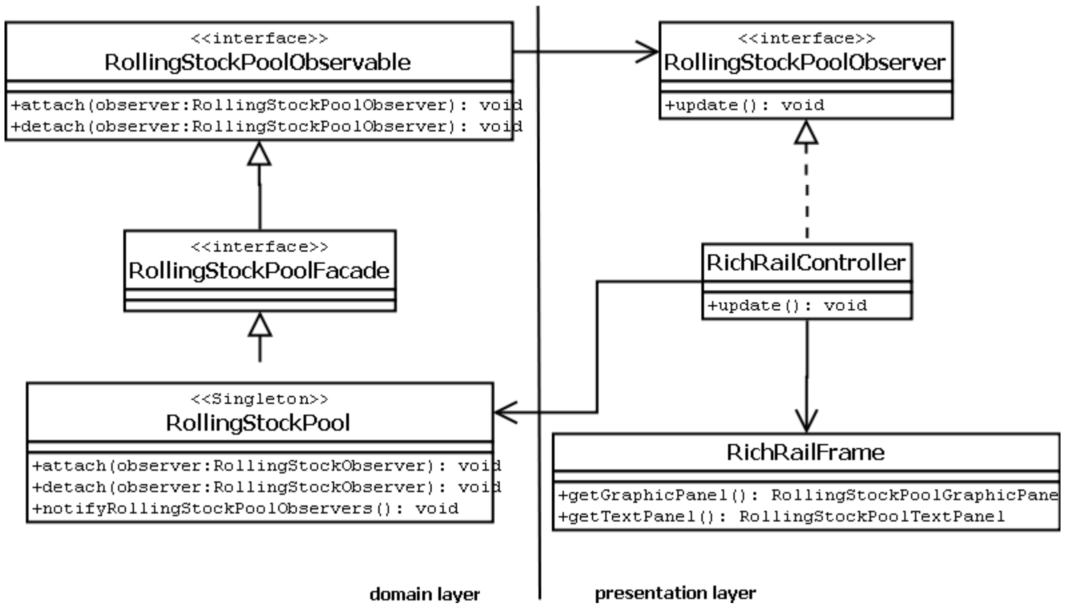
The way the Observer pattern is implemented here leads to tighter coupling between view and model than necessary. Sending a *message* with parameters from the domain layer to the view layer (using the `update`-method) clearly violates one of the principles of layering (do not talk to higher layers). This differs from a simple notification (or a callback).
Furthermore there exist 2 versions of the domain model, introducing the possibility of inconsistency and increasing redundancy.
None of the view objects had to be aware of the event mechanism. This should be hidden in the domain layer.

Group 2

As shown in figure 4, this group had chosen for their own implementation of the pattern, so they're not using the `java.util`-classes/interfaces.

First thing here is that they did not include the notify()-method as defined in the book of the GoF [GHJV95]. This is anyway correctly implemented in the Observable-class (`RoolingStockPool`, the facade of the domain).
Looking for all concrete Observers gives only one implementation: the `RichRailController`. At the classroom presentation of their running system the refresh of both textual and graphical representation of the model did work (as requested in the assignment criteria), so we take a closer look on what happens if the Observer gets his update-message:

```
44: public void update()
45: {
46:   Collection<Train> allTrains = RollingStockPool.getInstance().getAllTrains();
47:   Collection<Wagon> allDecoupledWagons = RollingStockPool.getInstance().getAllDecoupledWagons();
48:
49:   // reset the graphic & text panels
50:   frame.getGraphicPanel().clear();
51:   frame.getTextPanel().clear();
52:
53:   // have the graphic & text panels draw all the trains
54:   int track = 0;
55:   for (Train train : allTrains)
56:   {
57:     frame.getGraphicPanel().addTrain(track, train.getId());
```



<<interface>>
RollingStockPoolObservable
+attach(observer:RollingStockPoolObserver): void
+detach(observer:RollingStockPoolObserver): void
<<interface>>
RollingStockPoolObserver
+update(): void
<<interface>>
RollingStockPoolFacade
RichRailController
+update(): void
<<Singleton>>
RollingStockPool
+attach(observer:RollingStockObserver): void
+detach(observer:RollingStockObserver): void
+notifyRollingStockPoolObservers(): void
RichRailFrame
+getGraphicPanel(): RollingStockPoolGraphicPane
+getTextPanel(): RollingStockPoolTextPanel
domain layer
presentation layer

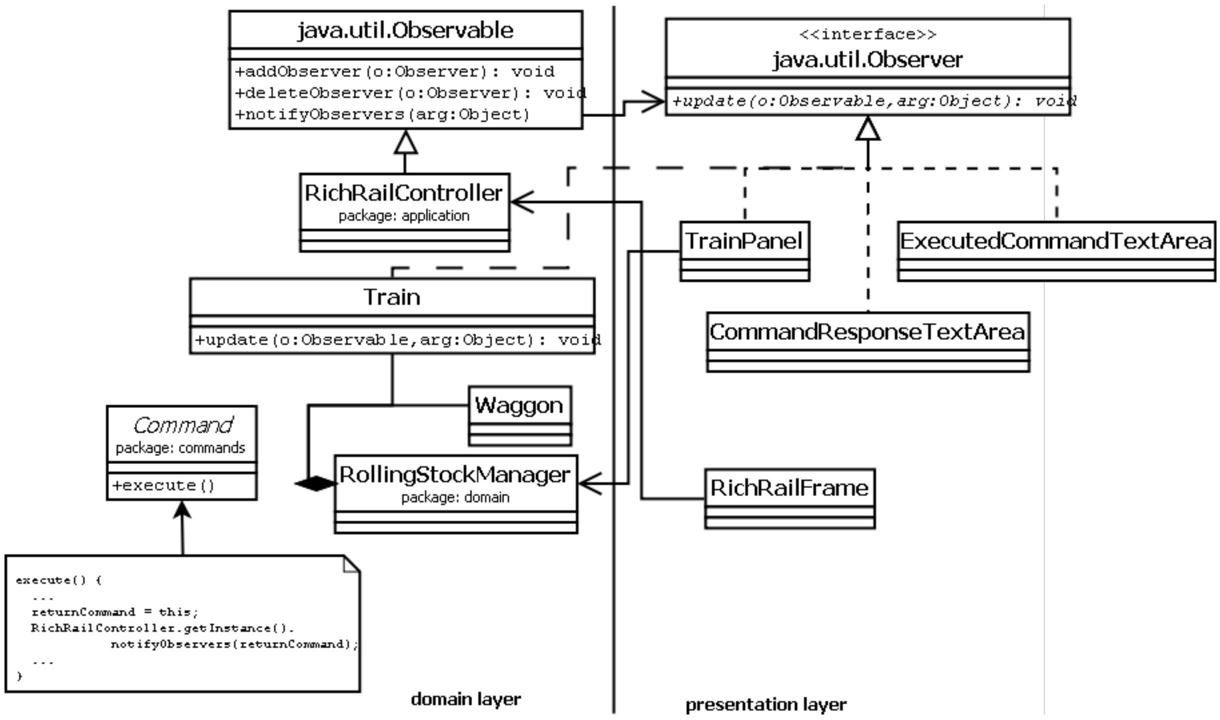
Figure 4: Implementation group 2 (only relevant parts shown)

```
58:     frame.getTextPanel().addTrain(train.getId());
59:     int slot = 0;
60:     for (Wagon wagon : train.getWagons())
61:     {
62:       frame.getGraphicPanel().addWagon(track, slot++, wagon.getId());
63:       frame.getTextPanel().addCoupledWagon(train.getId(), wagon.getId());
64:     }
65:     track++;
66:   }
67:
68:   // have the graphic & text panels draw all uncoupled wagons
69:   for (Wagon wagon : allDecoupledWagons)
70:   {
71:     frame.getGraphicPanel().addWagon(track, -1, wagon.getId());
72:     frame.getTextPanel().addWagon(wagon.getId(), wagon.getAmountOfSeats());
73:     track++;
74:   }
75:
76:   frame.getGraphicPanel().draw();
77:   frame.getTextPanel().draw();
78: }
```

In lines 46 and 47 the Observer correctly gets the information from the model. But then it starts to clean both the graphical and the textual representation and fills them with the current information from the model. So effectively this class is responsible for updating both representations.
This introduces higher coupling between this controller (which functions as Observer) and all representations and therefore introduces a third player in this Observer implementation. It also highly decreases the interchangeability of the representations, as requested in the requirements. If a new representation has to be added or an existing one has to be changed, this not only has to be done in the new representation but also in the Controller-class.
Good thing here is that this Controller was located in the GUI-layer of the application, so at least the rules of layering are not violated.

Conclusion group 2

Actually, one can discuss if this solution is a good one. As stated by the students, one of the reasons why they had chosen this solution is the fact that the model only has to be loaded once for both representations, improving the performance and decreasing code redundancy. But this comes with the cost of higher coupling of the representations via the Controller-class (in the role of the Observer). So, one of the goals of the Observer pattern, low coupling of subject and observer, has been reached, but at the same time higher coupling has been introduced with introducing an extra role here - the controller. If the concerns were separated, the view objects would have been responsible for updating themselves, not the controller.



```
execute() {
  ...
  returnCommand = this;
  RichRailController.getInstance().
           notifyObservers(returnCommand);
  ...
}
```

Figure 5: Implementation group 3 (only relevant parts shown)

Group 3

Figure 5 shows the implementation of group 3. We first notice that class `Train` in the domain-package implements the java.util.Observer interface. But this class is actually not used as Observer, as shown in following code fragment:

```
public void update(Observable o, Object arg) {
  throw new UnsupportedOperationException("Not supported yet.");
}
```

This probably was one of the (wrong) ways the students went while exploring how to implement the Observer pattern.
The other three classes from the presentation package (TrainPanel, CommandResponseTextArea, and ExecutedCommandTextArea) implement the Observer-interface. But only one of these classes, TrainPanel, is actually the representation of the model. The other two just show the executed commands and system responses after the execution of the commands. So these two have nothing to do with the model and shouldn't therefore be observing it! But, as we will see in the next section, they aren't observing the model.

The Observers are not connecting themselves to the Observable, this is done by class `RichRailFrame` which unnecessarily adds another class as participant in their Observer implementation.

The Observable class is extended by class `RichRailController`. We expected that this was done by the domain model, the class `RollingStockManager`. This is the only class which has knowledge of the domain model and if changes occured, so this class should be observed and this class should notify the observers that they need to update themselves. But instead the controller is the Observable. The controller self does not know the model, so it doesn't make sense to observe it. Furthermore, the notifyObserver-method is called only by the Command class (in the execute-method), as shown in figure 5.

This means that the notification of a change of the observed subjects is *not* done by the observable self, which is the only object which knows for sure that it has changed its state. Instead each time a command is executed also the observable is requested to notify its observers. The fact that the subject (in this case the `RichRailController`) has changed does not automatically lead to an update of the observers. So any modification to the model without using the Command-class has no effect on the representations.

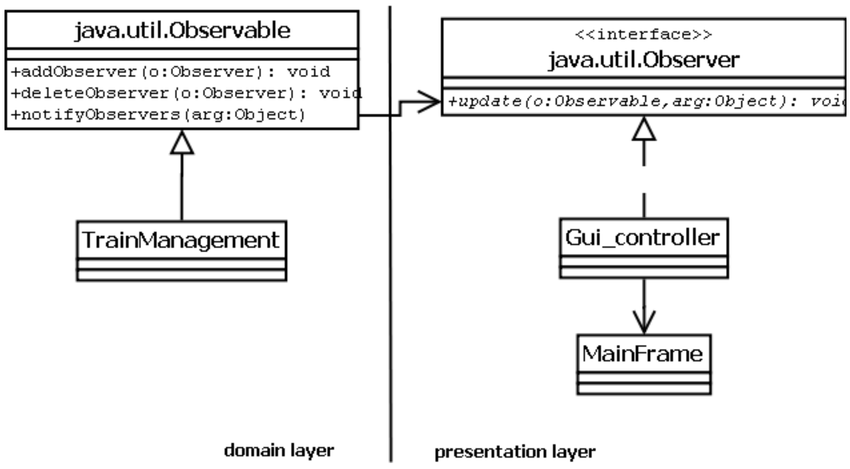

java.util.Observable
+addObserver(o:Observer): void
+deleteObserver(o:Observer): void
+notifyObservers(arg:Object)
<<interface>>
java.util.Observer
+update(o:Observable,arg:Object): void
TrainManagement
Gui_controller
MainFrame
domain layer
presentation layer

Figure 6: Implementation group 4 (only relevant parts shown)

Further examination of the code exposed that the returnCommand, which is given as argument in notifyObservers, is not used. Instead each representation gets the information from the domain model, which is located in `RollingStockManager` in package `domain`. The returnCommand is well used by the other two Observer classes (CommandResponseTextArea and ExecutedCommand-TextArea) to display the executed commands and their responses.

Conclusion group 3

The observable is not the model, but a controller. The model self is used by only one of the observers. The notification of the observers is done by another class and not the model self. This means that one can only hope that indeed the model has changed, which is not always the case (if an incorrect command is committed then the model is not changed but the observers are notified anyway!). Many classes take part in this observer-implementation, highly increasing the coupling between different layers and packages.
Also other classes are made part of the pattern implementation, violating the separation of concerns and information hiding principles.

Group 4

Figure 6 shows the implementation of group 4. The model is implemented in class `TrainManagement`, which is also the Observable. Class `Gui_controller` is the only observer, which already gives a hint that here again, as in group 2, the changes of the model and the following refresh of the representations are done by one class. Again, we expected that the views (located in the MainFrame class) will refresh themselves after being notified.
As we take a look in the update-method of the Observer class, we see that the only thing done is calling another method (`paint()`). In this method we see that both representations, graphical and textual, are refreshed in the controller class (via `mf.setData(x)` and `drawTrain/Waggon(x,y)`).

```
public void paint() throws IOException{
  ...
  mf.setData(model.getData());
  split = str.split("-");
  drawTrain(split[0],tr);
  for(int i=1;i<split.length;i++){
    wg++;
    mf.setData("wg:"+wg);
    drawWagon(split[i],wg, tr);
  }
  ...
}
```

This leads to high coupling between this controller-class and the two representations. Again, for performance, this solution is better, because the model is just loaded once. But it was explicitly stated in the assignment that performance is not a requirement.
After a command in the model was executed (e.g. a new train was added), the model class called

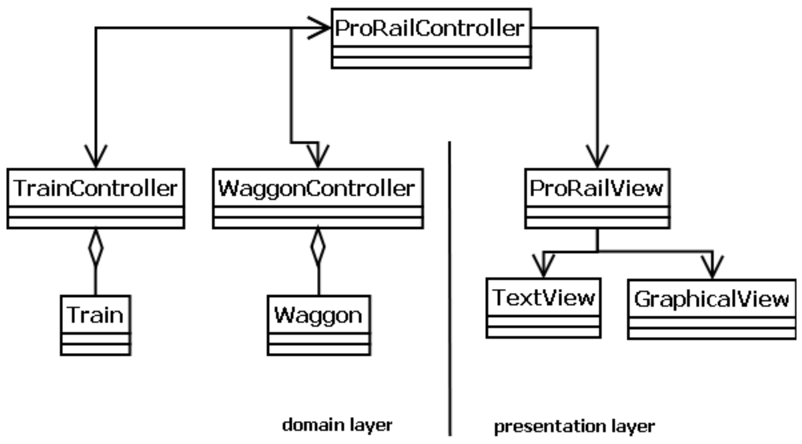


Figure 7: Implementation group 5 (only relevant parts shown)

notifyObservers(), in this case without an argument. So that part of the observer pattern was well implemented in this group.

Conclusion group 4

The separation of observable and observer was implemented correctly from this group. But the separation of both representations (and therefore an increased configurability of the application) was not realized as required. So every time a new or changed representation is added to the application (or an existing one is removed), not only the representation has to be implemented, but also the source of the controller has to be changed on different places, making maintenance more complicated and error-prone.

Group 5

This group didn't implement the observer pattern. According to their own explanation, they got stuck while trying to use it and decided to do it another way. But it is anyway interesting to take a look what they had done to implement the requirement of interchangeable and extendable representations.
They have a controller which takes care of the models and has the connections to the different representations. After each command both representations are refreshed from within this controller. Actually this is a typical MVC implementation.
Problem here is that both representations are strongly coupled to the controller as is the domain model. So again, any changes in the representation or additional representations require code changes in the controller.

Actually, they correctly implemented the mechanism that a class gets information from the model after the model has changed to represent it with the new state. Only it was in the wrong class, as it should be implemented in the view and not in the controller.

Conclusion group 5

It seems that this group thought that the usage of MVC does make the usage of the observer pattern obsolete without recognizing that they violated some principles this way and didn't implement all requirements.

4 Conclusion

This case study proves the statement "Novices don't infer patterns naturally" [CL99] to be correct. Actually, we as teachers had expected that they do and have been proved wrong. We summarize what went wrong in this section and then discuss possible consequences for teaching this pattern and propably patterns in general.

Even if explicitly not required, some groups explained their implementation with performance improvement. But at the same time they violated some basic principles (which have been taught to them).

Most groups used a controller, which was responsible for the refresh of the different UI-representations of the model. In 2 groups this controller also fulfilled the role of the observer, while in 1 group the controller was the observable. In another group the controller was responsible for connecting observers and observable.
This leads to the assumption that the students had problems to integrate MVC and the observer pattern. The responsibility of the controller was interpreted in different ways, leading to different implementations.

In 2 groups the controller was responsible for reacting on user interactions and changing the model, but was also responsible for the drawing of the graphical representation and the textual representation, therefore including different tasks, which violates the separation of concerns-principle.

Only one group implemented a clear separation of the 2 representations (group 1). So all groups recognized that the requirements asked for the usage of the observer pattern, but if it came to the implementation, they lost the goals and advantages of the observer pattern and implemented it in a way which does not help in the given context. This due to introducing more coupling and therefore unnecessary complexity.

All groups which used the implementations offered by Java in the java.util package had problems implementing their solution correctly. So it seems that these implementations doesn't help. Furthermore a couple of problems are related to these implementations: java.util.Observable is implemented as a class and not an interface. This means that if one uses this implementation and wants to observe classes located in a class hierarchy, that also superclasses which are not observed have to implement the Observable-class, even if they are not being observed. This violates Liskov's substitution principle [Lis87].

5 Consequences for teaching the Observer Pattern

Design patterns are part of the design of a software system, so they can't be taught and applied without focus on the overall design. The case study shows that if the students focus mostly on the patterns, they seem to forget to check if the overall design still follows the learned principles. So the design implications of all patterns should be discussed with the students as well, as suggested by Rasala [Ras97]. He also suggests to "insist that students polish the design aspects of their programs before handing in their work" and to "make the software project course a quality design experience not just a rite-of-passage".

This can be supported by using some of the pedagogical patterns as described in [BEM$^+$, BEMSb, BEMSa, BEMW]. In the pattern *Prefer Writing* they suggest to let students rewrite their programs. This could also be used for letting them improve their design.
Another useful pattern is *Round and Deep*, which makes use of the variety of the students' own experiences to deepen their own understanding of the concept and to provide alternative perspectives for other students. This could be applied by letting them implement small solutions to problems as described in some design patterns (without knowing the patterns yet). The teacher can use this experience to initiate a discussion on the differences, advantages, and disadvantages of the students solutions. This way it is much easier to relate the solutions suggested by design pattern descriptions to their own experience, which is also described in the pedagogical patterns *Reflection* and *Linking Old to New* [BEMW]. The learning effect is even bigger if some student groups found the solutions of the patterns themselves (or were close to it).

Another advantage of this teaching method would also be that the focus can be put on design first and the patterns could then be related to this experience. This prevents the students from applying patterns without knowing what they're actually doing and helps them to develop the necessary abstract understanding [CL99].

Some pedagogical patterns were already succesfully used in the course. The students got the assignment and then had 2 weeks to make a first version of their program design. This was presented by all groups to all other groups including explanations of design decisions and discussions of all groups initial designs. Here the pattern *Explore for Yourself* [BEMW] was used and the design improvements by all groups which we observed after this session were noticable.

The objectives and the advantages of the Observer pattern have to be made more clear. It seems that just mentioning the parts of the pattern and the applicability as described in 'Design Patterns' [GHJV95] are not clear enough. All groups recognized that the Observer pattern offers a solution for their problem in their context, but their implementation doesn't solve their problem as expected.

To make this more obvious to the students, another assignment could be given were another representation has to be added to the application already developed. While solving this additional assignment the students should record the amount of changes (and their places) needed. This could be compared with the change amounts and solutions of other groups, which offers insights in the real advantages of correctly implementing the Observer pattern.

In general it has to be made clear that design patterns are best practices used to implement solutions to problems in well defined contexts. They also should fit into the design or help with design, but they should not be leading. If not implemented correctly and not helping in meeting the requirements, they are of no value. However, basic OO-principles are still leading, so even if patterns are applied, the implementation should be validated against these principles. Obviously this is an important part in teaching patterns, as all solutions implemented by the groups violated basic principles. This is also supported by [CL99] with the statement that "Instruction can't focus only on patterns".

A good pattern for doing this is '*It's still OO to me*' (originally intended for framework development), published in [CC02]. It is about "recognizing that working on an object-oriented framework doesn't suddenly give you an exemption from good OO practices and that fixing bad OO practices is more difficult and potentially embarassing with frameworks than with software". This can also be applied for the usage of patterns, saying that following the principles is leading. Now it seemed that the students thought that the pure fact that a pattern is used makes the design good, which of course isn't the case.

The connection between MVC and the Observer pattern has to be made more clear, as well as the differences. Especially the role of the controller is important here. The fact that the controller has different roles in different student implementations makes evident that there is need in explaining this relation better.
An interactive roleplay of the observer pattern, intergrating the MVC pattern as well, could help to evolve a better understanding. The pedagogical patterns *Physical Analogy* and *Role Play* [BEMW] could be used to do this.

References

[AMBC98] Owen Astrachan, Garrett Mitchener, Geoffrey Berry, and Landon Cox. Design patterns: an essential component of cs curricula. In *SIGCSE '98: Proceedings of the twenty-ninth SIGCSE technical symposium on Computer science education*, pages 153–160, New York, NY, USA, 1998. ACM.

[BEM$^+$] Joseph Bergin, Jutta Eckstein, Mary Lynn Manns, Helen Sharp, and Marianna Sipos. Teaching from different perspectives. `http://www.pedagogicalpatterns.org/`. Retrieved June 06, 2010.

[BEMSa] Joseph Bergin, Jutta Eckstein, Mary Lynn Manns, and Helen Sharp. Feedback patterns. http://www.pedagogicalpatterns.org/. Retrieved June 06, 2010.

[BEMSb] Joseph Bergin, Jutta Eckstein, Mary Lynn Manns, and Helen Sharp. Patterns for active learning. http://www.pedagogicalpatterns.org/. Retrieved June 06, 2010.

[BEMW] Joseph Bergin, Jutta Eckstein, Mary Lynn Manns, and Eugene Wallingford. Patterns for gaining different perspectives. http://www.pedagogicalpatterns.org/. Retrieved June 06, 2010.

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented Software Architecture - A System of Patterns*. John Wiley & Sons, Chichester, 1996.

[Bro95] Frederick P. Brooks, Jr. *The mythical man-month (anniversary ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[CC02] James Carey and Brent Carlson. *Framework process patterns: lessons learned developing application frameworks*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[CL99] Michael J. Clancy and Marcia C. Linn. Patterns and pedagogy. In *SIGCSE '99: The proceedings of the thirtieth SIGCSE technical symposium on Computer science education*, pages 37–42, New York, NY, USA, 1999. ACM.

[Dij82] E. W. Dijkstra. Ewd 447: On the role of scientific thought. *Selected Writings on Computing: A Personal Perspective*, pages 60–66, 1982.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, Boston, MA, 1995.

[IEE04] IEEE Computer Society. *Software Engineering Body of Knowledge (SWEBOK)*. Angela Burgess, EUA, 2004.

[Lis87] Barbara Liskov. Keynote address - data abstraction and hierarchy. In *OOPSLA '87: Addendum to the proceedings on Object-oriented programming systems, languages and applications (Addendum)*, pages 17–34, New York, NY, USA, 1987. ACM.

[MA04] Scot F. Morse and Charles L. Anderson. Introducing application design and software engineering principles in introductory cs courses: model-view-controller java application framework. In *Sciences in Colleges, Volume 20, Issue 2, Pages: 190 - 201*, 2004.

[McC04] Steve McConnell. *Code Complete, Second Edition*. Microsoft Press, Redmond, WA, USA, 2004.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[Ras97] Richard Rasala. Design issues in computer science education. *SIGCSE Bull.*, 29(4):4–7, 1997.