Observations on the Observer Pattern

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

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 several design principles. This paper discusses what went wrong and why it probably went wrong. Possible suggestions are given for teaching the Observer pattern and for teaching design patterns in general.

Categories and Subject Descriptors: D.1.5 [**Programming Techniques**] Object-oriented Programming; D.2.10 [**Software Engineering**]: Design—*Design Patterns*; K.3.2 [**Computers and Education**]: Computer and Information Science Education —*Computer science education*

General Terms: Design, Languages, Education

Additional Key Words and Phrases: Design Patterns, Pedagogical Patterns

ACM Reference Format:

Köppe, C. 2010. Observations on the Observer Pattern. jn 2, 3, Article 1 (May 2010), 14 pages.

1. INTRODUCTION

Design patterns are recognized as common knowledge in the software engineering field [IEEE Computer Society 2004]. They help to implement basic OO-principles and offer many advantages such as best practices, a common vocabulary, and approved solutions. From a pedagogical perspective, they help in teaching software design and modeling [Rasala 1997] and should be integrated in computer science curricula [Astrachan et al. 1998].

While some research and case studies exist which discuss how to teach design patterns (e.g. [Clancy and Linn 1999; Morse and Anderson 2004]), there seems to be no common best approach. We have experience which is not documented. 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 [Brooks 1995]) and violate some basic principles while trying to implement others.

This work presents a case study based on a course 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 did not adhere to the OO-principles previously taught to them. Finally, suggestions are discussed how the teaching of this pattern (and design patterns in general) could be improved to achieve that students better understand the usage of patterns. This should help the students to deliver more correct solutions which also follow the basic OO-principles.

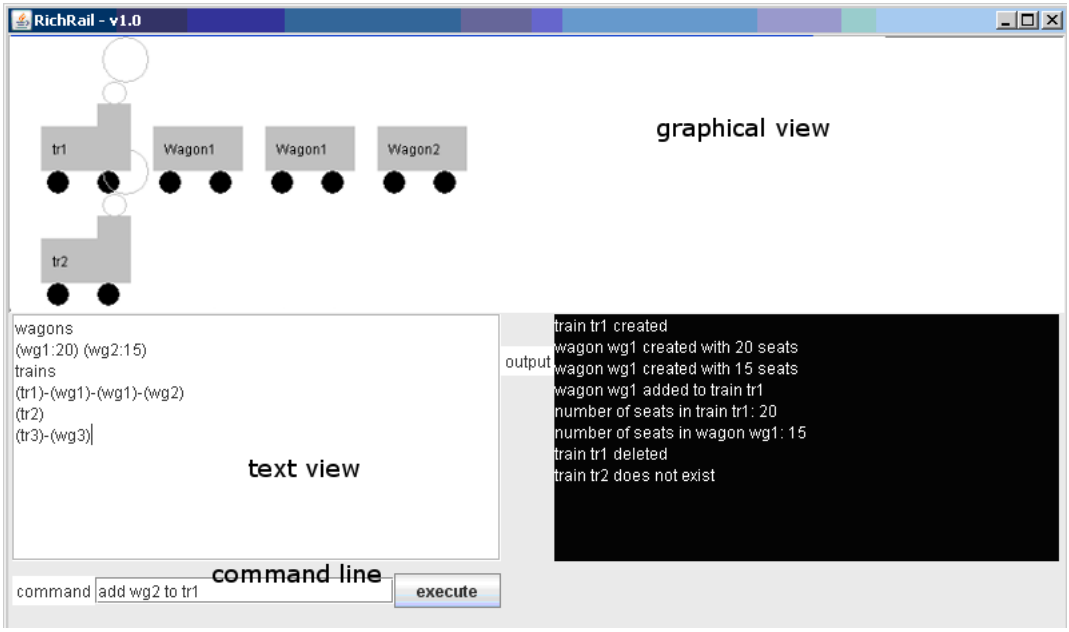

Fig. 1. GUI prototype of solution

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 (such as collection of best practices, common vocabulary etc.) and showing some examples. These examples included the patterns Factory, Observer, Singleton, and Adapter in more detail and an overview of other patterns.
The group consisted of undergraduate students of Computer Science with the focus on Software Engineering. The course 'Patterns and Frameworks' was taught in the second half of the third year within the specialization module 'Software Architecture and Design'. At this point of study the students were familiar with object-oriented concepts as encapsulation, loose coupling, separation of concerns, and also basic architectural principles as layering. They were introduced to Model-View-Controller and used this in different projects.

The students were given an assignment that would require the exploration of common design patterns (from books like [Gamma et al. 1995] and other sources) and the application of their OO-knowledge. It was explicitly chosen to not asking for specific patterns, but just issuing some problems so that the students have to discover for themselves which patterns can be used best to solve the problems. Most of the design patterns were applied correctly, e.g. command, facade, factory, singleton, and interpreter. 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. A couple of improvements and new functionality were described which had to be implemented by the students. The new requirements included also different representations of the domain model (the trains and wagons created), formulated as follows:

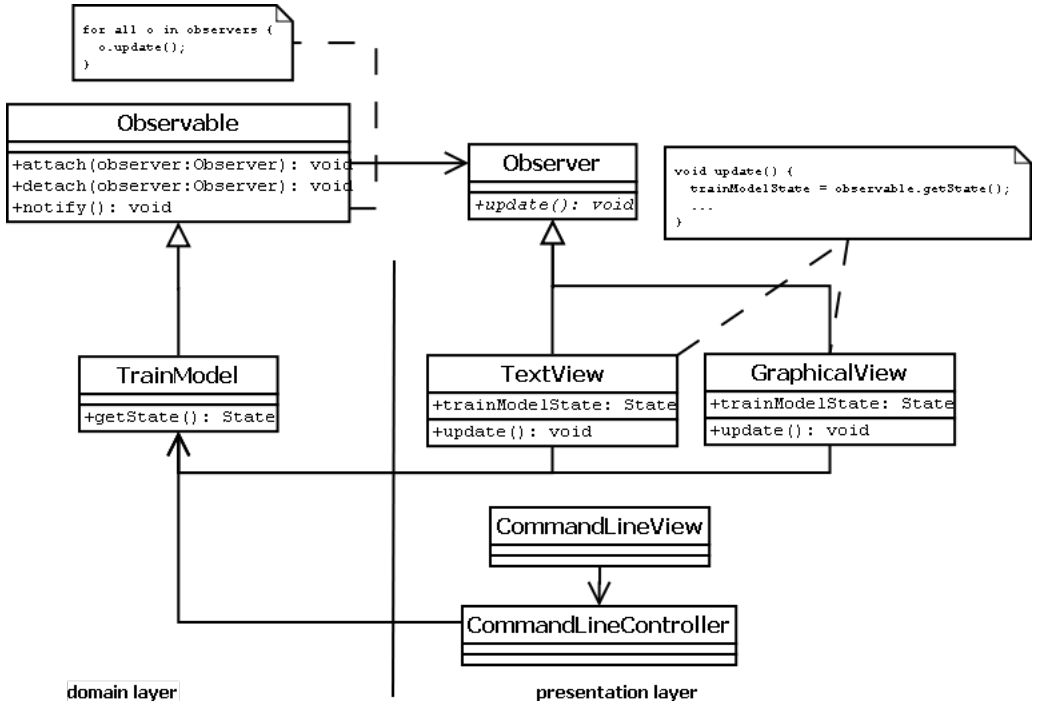

Fig. 2. Expected solution incl. optional MVC-part for the command line (only relevant parts shown)

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 easy to integrate
- ...

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." [Gamma et al. 1995]. 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 to be 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 would 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 indeed 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 expected to follow the design principles they had been taught:

—**Loose Coupling** - designing so that connections among different parts of the program are kept to a minimum [McConnell 2004]

—**The Rules of Layering** - Layers should only be aware of the next lower layer and callbacks are used for bottom-up communication to minimize coupling between different layers [Buschmann et al. 1996]

—**Information Hiding** - hide design and implementation decisions from the rest of a program in one place [Parnas 1972; McConnell 2004]

—**Separation of Concerns** - different or unrelated responsibilities should be separated from each other within a software system [Dijkstra 1982; Buschmann et al. 1996]

Surprisingly, 4 out of the 5 groups tried to implement the Observer pattern, but none of them got it completely right as was to be expected. Even the 5th group tried to implement it as well but got stuck and they decided to proceed without using the Observer pattern.

Conventions. In some books the two important abstractions contained in the Observer pattern are observer and subject. We prefer to use the term observable rather than subject. However, they may be considered as synonyms.

3. ANALYSIS OF THE SOLUTIONS

The source code of the delivered solutions was examined and compared with the expected solution. The class diagrams in the subsequent sections were made by the author to help analysing the students' solutions. It was also taken into account for this analysis if the requirements seemed to be fulfilled when the solutions were presented at the end of the course in a live demo, as well as the comments the students gave themselves while doing their presentations.

Group 1

Figure 3 provides an overview of the relevant parts of this group's solution. They chose to use MVC and introduced controllers for the different views, even though this is not necessary. Each controller has a connection to either a textual view or a graphical view (via *JTrain*). The controller also knows the model. Except for introducing controllers as observers instead of the graphical representations of the model, this seems all right.
They also include events that occur in the model (the class `PoolChangeEvent`). These events are also known to the controller, which is unusual. To determine how they arrived at this particular model 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. In the attachRollingStock and detachRollingStock routines it calls `notifyObservers` after setting `super.setChanged()` as the following code fragment shows.

```
public class Train extends RollingStock
...
protected void attachRollingStock(RollingStock rs)
{
  rollingStockItems.add(rs);
```

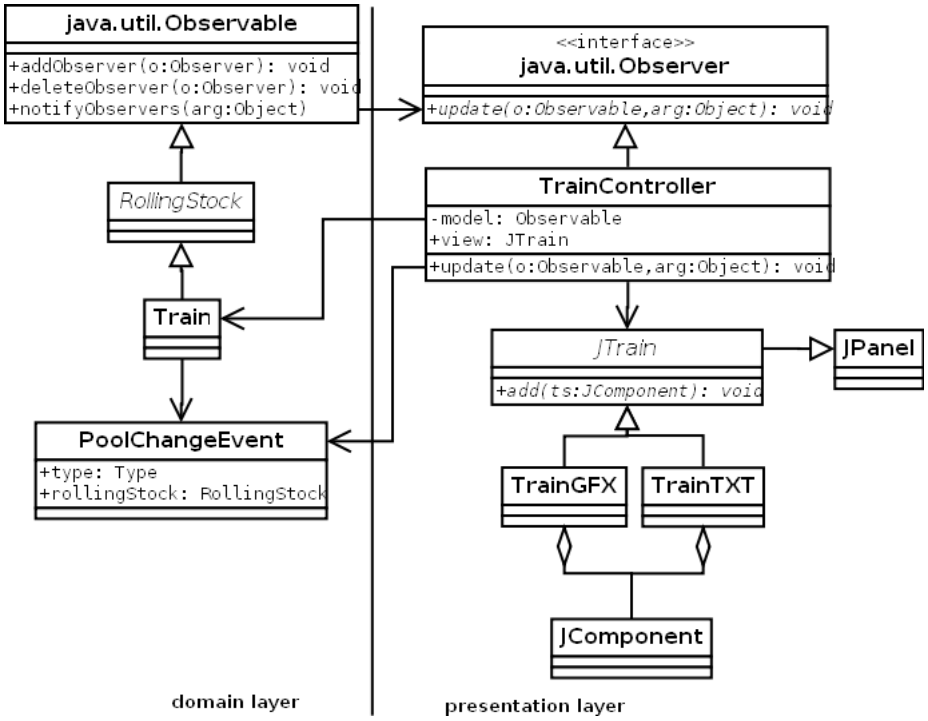

Fig. 3. Implementation group 1 (only relevant parts shown)

```
  setChanged();
  notifyObservers(new PoolChangeEvent(Types.add, rs));
}
```

TrainController – the observer – knows the model, as can be seen in Figure 3. However, it does not use the model. Instead the event which had taken place is also sent to the observer (using a new PoolChangeEvent with attribute *Types.add*, see code), telling it what happened and also sending the rollingStock newly added 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 the following code fragment.

```
15: public class TrainController
           implements Observer, Controller
...
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 the fact that it is cast in line 34). The usual reason to include an observable is that more than one subject is observed by one observer, which is not the case here. However, this observer in fact doesn't do anything with the observable.

(2) The second parameter is 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:

```
public class TrainGFX extends JTrain
...
@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, which results in two independent models being created: the real domain model and an additional model in the view layer of the application. Thus it is possible to change the model in the view layer throwing it out of sync with the domain model (and there is no way to repair this apart from removing it from the view layer model).

Conclusion group 1. The way the Observer pattern is implemented here leads to a 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, the introduction of a second, redundant domain model may lead to inconsistencies.
None of the view objects needed to be aware of the event mechanism. This should be hidden in the domain layer.

Group 2

As Figure 4 shows, group 2 preferred to make their own implementation of the pattern, not using the `java.util`-classes/interfaces.

First, they did not include the notify()-method as defined in the book of the GoF [Gamma et al. 1995]. Nevertheless, the method is correctly implemented in the observable-class (`RoolingStockPool`, the facade of the domain). Checking for all concrete observers yields only one implementation: the `RichRailController`. This controller should not be an observer. However, during the classroom presentation of their running system the refresh of both textual and graphical representations of the model did indeed work (as required in the assignment criteria), so we take a look at what happens when the observer gets his update-message:

```
15: public class RichRailController implements
RollingStockPoolObserver
```

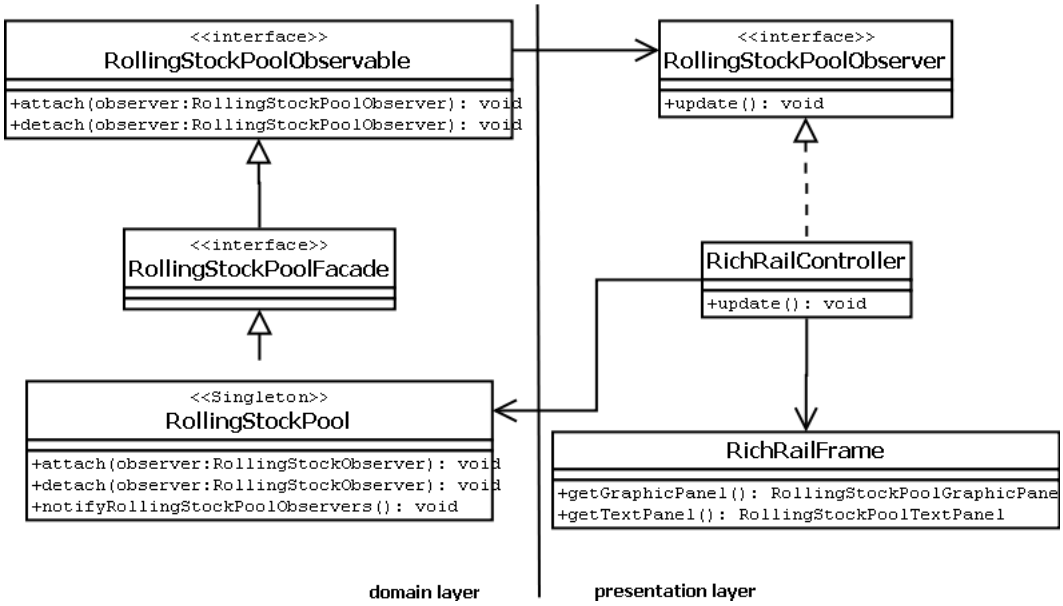

Fig. 4. Implementation group 2 (only relevant parts shown)

```
...
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());
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++;
```

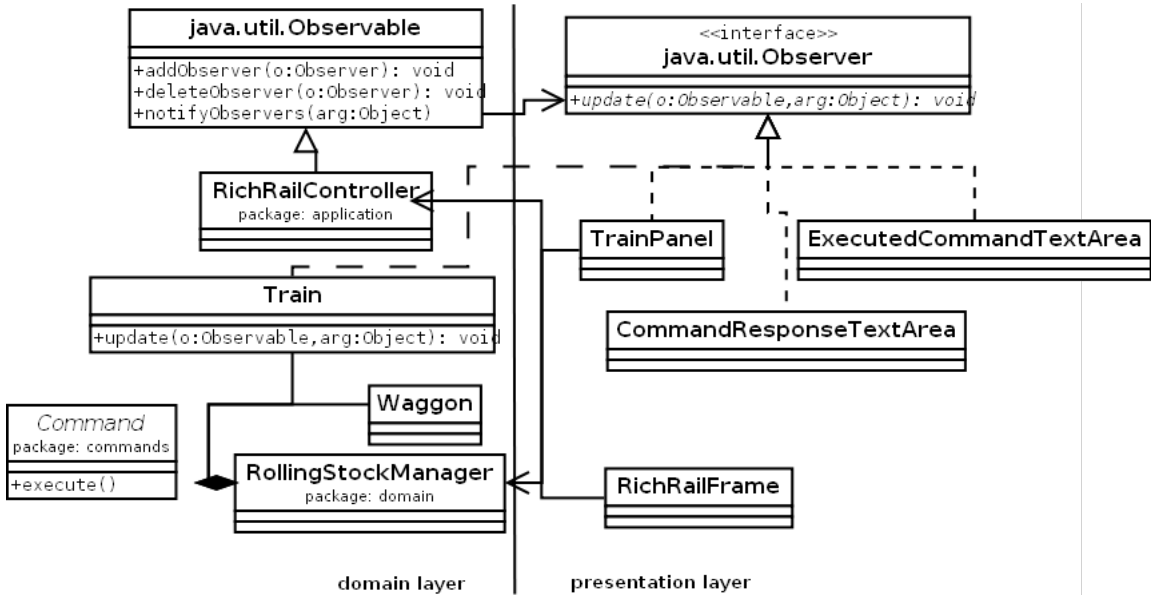

Fig. 5. Implementation group 3 (only relevant parts shown)

```
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 clear 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 pattern 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 has to be done not only in the new representation but also in the controller-class.
On the positive side: this controller was located in the GUI-layer of the application, so at least the rules of layering were not violated.

Conclusion group 2. It is debatable whether this solution is a good one. As stated by the students, one of the reasons for choosing this particular solution is the fact that the model only has to be loaded once for both representations, thereby improving the performance and decreasing code redundancy. But this comes at 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 by adding an extra role here - the controller. In order to follow the separation of concerns principle for the two views, the view objects should have been responsible for updating themselves, not the controller.

Group 3

Figure 5 shows the implementation of group 3. Class `Train`, which is located 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 class Train extends RollingStock
                   implements Observer
...
public void update(Observable o, Object arg) {
  throw new UnsupportedOperationException("Not supported yet.");
}
```

This probably was one of the (wrong) choices the students made 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 display the executed commands and system responses after the execution of the commands. It is obvious that these two have nothing to do with the model and should therefore not observe it! But, as we will see in the next section, they are actually not observing the model proper.

The observers do not connect themselves to the observable, this is done by class `RichRailFrame` which unnecessarily adds another class as participant in their Observer pattern implementation.

The observable class is extended by class `RichRailController`. We expected that this would be done by the domain model, the class `RollingStockManager`. This is the only class which knows the domain model and occurring changes, 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 itself does not know the model, so it does not make any sense to observe it. Furthermore, the notifyObserver-method is called only by the command class (in the execute-method), as shown in following code fragment:

```
public class Command extends RollingStock
                   implements Observer
...
public void execute() {
  todo();
  Command returnCommand;
  if (errorCommand == null) {
returnCommand = this;
  } else {
    returnCommand = errorCommand;
    returnCommand.setCommandString(this.commandStr);
  }
  RichRailController.getInstance().notifyObservers(
                                       returnCommand);
}
```

This means that the notification of a change of the observable is *not* done by the observable itself, which is the only object which knows for sure that it has changed its state. Instead each time a command is executed, the observable is also 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.
Further examination of the code revealed that the returnCommand, which is given as argument in notifyObservers, is only used by the other two observer classes – CommandResponseTextArea and ExecutedCommandTextArea – to display the executed commands and their responses. The graphical representation in class TrainPanel gets the information from the domain model, which is located in `RollingStockManager` in package `domain`.

```
public class TrainPanel extends JPanel implements Observer {
```

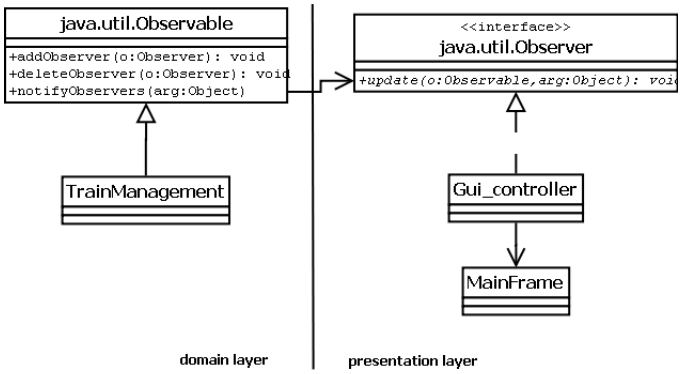

Fig. 6. Implementation group 4 (only relevant parts shown)

```
  public void update(Observable o, Object arg) {
            paintComponent(getGraphics());
  }

  @Override
  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    OldSchoolFactory factory = OldSchoolFactory.getInstance();

    RollingStockManager pool =
                    RollingStockManager.getInstance();
    Hashtable<String, Train> trains = pool.getTrains();
    Collection<Train> values = trains.values();
    Iterator<Train> trainIterator = values.iterator();
  ...
}
```

Conclusion group 3. The observable is not the model, but a controller. The model itself is used by only one of the observers. The notification of the observers is done by another class and not the model itself. This means that one can only hope the model has indeed changed, which is not always the case (if an incorrect command is issued, the model is not changed but the observers are notified anyway!). Many classes take part in this observer-implementation, greatly increasing the coupling between different layers and packages.
Moreover 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 hints at the fact that here, again, as with 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 action 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/Wagon(x,y)`).

```
public class Gui_Controller implements Observer {
...
public void paint() throws IOException{
  ...
  mf.setData(model.getData());
  split = str.split("-");
  drawTrain(split[0],tr);
```

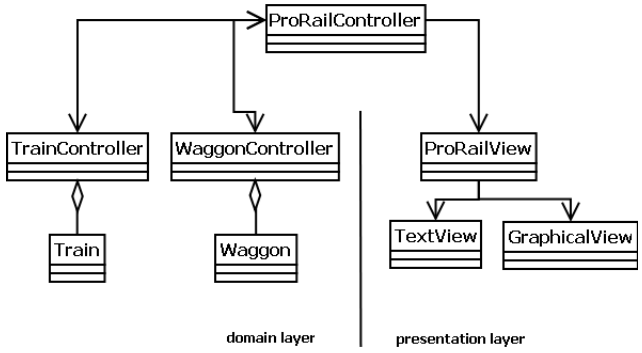

Fig. 7. Implementation group 5 (only relevant parts shown)

```
    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, this solution performs better, because the model is only loaded once. But it had been 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 notifyObservers(), in this case without an argument. So that part of the Observer pattern was well implemented by this group.

Conclusion group 4. The separation of observable and observer was implemented correctly by this group. But the separation of both representations (and therefore an increased configurability of the application) was not realized as expected. To add a new representation to the application (or remove an existing one), not only the representation has to be implemented, but also the code of the Gui_Controller class has to be changed at various places: in the paint-method code needs to be added or removed and the drawTrain/drawWagon methods have to be expanded or deleted. This makes maintenance of their system more complicated than necessary and therefore 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 use a different approach. But it is nevertheless interesting to take a look at what they did to implement the requirement of interchangeable and extendable representations.
Their design includes a controller to handle the models and to the connect to the different representations. After each command both representations are refreshed from within this controller. Actually this is a typical MVC implementation.
The 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. Unfortunately 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 did not implement all requirements.

4. CONCLUSION

This case study proves the statement "Novices don't infer patterns naturally" [Clancy and Linn 1999] to be correct. Actually, we as teachers had expected that they would and were 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 though performance was not an issue, some groups confused implementation with performance improvement, thereby violating some of the basic principles they had been taught.

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 integrating MVC and the Observer pattern. The responsibility of the controller was interpreted in different ways, leading to different implementations. It may therefore be safe to assume that they had no proper understanding of MVC.

In two groups the controller was responsible for reacting to user interactions and changing the model, but was also responsible for drawing the graphical representation and the textual representation incorporating a number of different tasks, which is in violation of the separation of concerns-principle.

Only one group managed to separat the 2 representations (group 1). Though all groups realized that the requirements asked for the usage of the Observer pattern, but when it came to the implementation stage they lost sight of what makes the Observer pattern so eminently suitable. As a result they introduced more coupling and therefore unnecessary complexity.

All groups that used the implementations offered by Java in the java.util package had problems implementing their solution correctly. So it seems that these implementations do not help.

5. SUGGESTIONS 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 mainly on the patterns, they seem to forget to check if the overall design still follows the principles learned. So the design implications of all patterns should be discussed with the students as well, as suggested by Rasala [Rasala 1997]. 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". The techniques as suggested by Joshua Kerievsky in "Refactoring to Patterns" [Kerievsky 2004] could be used for this purpose.

It also can be supported by using some of the patterns collected by the pedagogical pattern project and described in [Bergin et al. 2010; Bergin et al. 2010b; 2010a; Bergin et al. 2010]. 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* [Bergin et al. 2010]. The learning effect would be even greater if some student groups found the solutions of the patterns themselves or were close to finding them.

Another advantage of this teaching method would be that the focus will be 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 [Clancy and Linn 1999].

Some pedagogical patterns were already succesfully used in the course. The students were given the assignment and then had two weeks to make a first version of their program design. This was presented by all groups to all other groups. The presentations included explanations of design decisions and discussions of all groups' initial designs. Here the pattern *Explore for Yourself* [Bergin et al. 2010] was used and the design improvements by all groups which we observed after this session were noticable in terms of principles followed and the number of correctly used design patterns.

The objectives and the advantages of the Observer pattern have to be made clearer. It seems that just mentioning the parts of the pattern and the applicability as described in 'Design Patterns' [Gamma et al. 1995] is not enough. All groups recognized that the Observer pattern offers a solution for their problem in the given context, but their implementations did not yield the solutions the teachers had expected.

To make this more obvious to the students, another assignment could be given in which yet another representation has to be added to the application already developed. While solving this additional assignment the students should record the number of changes (and their places) needed. This could be compared with the change amounts and solutions of other groups, which offers insight into 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 repeatedly implement solutions to problems in well defined contexts. They should also 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 [Clancy and Linn 1999] 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 [Carey and Carlson 2002]. 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 may also be applied for the usage of patterns, saying that following the principles is leading. Now it seemed that the students thought that the simple fact that a pattern is used automatically produces a correct design, which of course isn't the case.

The connection as well as the differences between MVC and the Observer pattern should be made clearer. 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 to explain 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* [Bergin et al. 2010] could be used to do so.

Acknowledgements

The author would like to thank his shepherd Dave West for his valuable comments and feedback provided during the shepherding of this work. I would also like to thank all participants of the "Design & Process" workshop at PLoP'10 for their insightful comments and discussion during the conference: Ognjen Šobajić , Ralph Johnson, Mike van Hilst, Ali Raza, and André Saúde. Finally, Rob van der Pols helped with his corrections to improve this work.

REFERENCES

ASTRACHAN, O., MITCHENER, G., BERRY, G., AND COX, L. 1998. Design patterns: an essential component of cs curricula. In *SIGCSE '98: Proceedings of the twenty-ninth SIGCSE technical symposium on Computer science education*. ACM, New York, NY, USA, 153–160.

BERGIN, J., ECKSTEIN, J., MANNS, M. L., AND SHARP, H. 2010a. Feedback patterns. `http://www.pedagogicalpatterns.org/`.

BERGIN, J., ECKSTEIN, J., MANNS, M. L., AND SHARP, H. 2010b. Patterns for active learning. `http://www.pedagogicalpatterns.org/`.

BERGIN, J., ECKSTEIN, J., MANNS, M. L., SHARP, H., AND SIPOS, M. 2010. Teaching from different perspectives. `http://www.pedagogicalpatterns.org/`.

BERGIN, J., ECKSTEIN, J., MANNS, M. L., AND WALLINGFORD, E. 2010. Patterns for gaining different perspectives. `http://www.pedagogicalpatterns.org/`.

BROOKS, JR., F. P. 1995. *The mythical man-month (anniversary ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

BUSCHMANN, F., MEUNIER, R., ROHNERT, H., SOMMERLAD, P., AND STAL, M. 1996. *Pattern-oriented Software Architecture - A System of Patterns*. John Wiley & Sons, Chichester.

CAREY, J. AND CARLSON, B. 2002. *Framework process patterns: lessons learned developing application frameworks*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

CLANCY, M. J. AND LINN, M. C. 1999. Patterns and pedagogy. In *SIGCSE '99: The proceedings of the thirtieth SIGCSE technical symposium on Computer science education*. ACM, New York, NY, USA, 37–42.

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

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design Patterns*. Addison-Wesley, Boston, MA.

IEEE COMPUTER SOCIETY. 2004. *Software Engineering Body of Knowledge (SWEBOK)*. Angela Burgess, EUA.

KERIEVSKY, J. 2004. *Refactoring to Patterns*. Pearson Higher Education.

MCCONNELL, S. 2004. *Code Complete, Second Edition*. Microsoft Press, Redmond, WA, USA.

MORSE, S. F. AND ANDERSON, C. L. 2004. 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*.

PARNAS, D. L. 1972. On the criteria to be used in decomposing systems into modules. *Commun. ACM 15,* 12, 1053–1058.

RASALA, R. 1997. Design issues in computer science education. *SIGCSE Bull. 29,* 4, 4–7.