Implementing the Observer Design Pattern as an Expressive Language Construct

Taher Ahmed Ghaleb, Khalid Aljasser and Musab Al-Turki

Information and Computer Science Department
King Fahd University of Petroleum and Minerals
Dhahran 31261, Saudi Arabia
Emails: {g201106210, aljasser, musab}@kfupm.edu.sa

***Abstract*—Observer is a commonly used design pattern as it carries a lot of reusability and modularity concerns in object-oriented programming and represents a good example of design reuse. Implementing the observer design pattern (and several other design patterns) is known to typically cause several problems such as implementation overhead and traceability. In the literature, several approaches have been proposed to alleviate such problems. However, these approaches only considered the implementation of a specific scenario of the observer pattern, which is concerned with having a single subject with multiple observers. In addition, the code used to implement this pattern was scattered throughout the program, which complicated implementing, tracing and reusing them. In this paper, we: A) provide a systematic classification of all possible scenarios of the observer design pattern and B) introduce a novel approach to implement them using an expressive and easy-to-use construct in Java. The proposed observer construct is built as a language extension using the *abc* extensible compiler. We illustrate through several observer scenarios how the construct significantly simplifies the implementation and improves reusability.**

Keywords–Design Patterns; Aspect-Oriented Programming; Extensible Compiler; Language Extension; Observer Pattern.

I. INTRODUCTION

Object-oriented (OO) design patterns [1] are reusable solutions that reorganize OO programs in a well-structured and reusable design. They originally were implemented using OO features, such as polymorphism and inheritance. After Aspect-oriented (AO) programming languages emerged, researchers started to employ AO constructs to make the implementation more reusable and modular.

Despite the wide range of applications of design patterns, manually implementing them may lead to several problems including most notably implementation overhead, traceability and code reusability [2]. A programmer may be forced to write several classes and methods to achieve trivial behaviors, which leads to a sizable programming overhead, scattering of actions everywhere in the program, and reducing program understandability. Although design patterns make design reusable, the code (or at least part of the code) used to implement them cannot be reused later.

Although the observer pattern has been widely used in practice, a systematic investigation of methods of its implementation considering all the potential observing behaviors was missing in the literature. In particular, the only implemented scenario of the observer pattern in the literature is the one having a single subject with multiple observers, where the association of observers to subjects is subject-driven. For instance, implementations of the observer design pattern in [3] and [4] were illustrated using the example of having $Line$, $Point$ and $Screen$ classes, where the observing protocol is implemented in a way that a single subject can have a list of observers. This particular example actually shows a different case where many subjects (i.e., $Lines$ and $Points$) can be observed by a single observer (i.e., $Screen$). Another issue of conventional implementations of the observer pattern is concerned with the indirect way of implementing the pattern. In other words, programmers in such approaches cannot deal with the pattern as a recognizable unit in programs. Instead, they are required to build an observing protocol and apply it to each instance interested in observing a particular subject, which leads to increased dependencies in the programmer's code.

Motivated by this, we aim in this paper to address these issues while making two main contributions. First, we systematically study and classify the possible scenarios of applying the observer design pattern. This classification is essential for gaining a comprehensive understanding of the structure, design and usage of the observer pattern. Second, we introduce a novel approach to implement the observer design pattern (with all its possible scenarios) as an identifiable language construct. This approach is implemented as a language extension providing a very expressive and easy-to-use observer construct. The implementation of the observer pattern in this approach becomes more explicit and is significantly simplified as shown in the typical examples presented in the paper. Consequently, this implementation approach promotes code correctness by reducing chances of making programming errors (both in the implementation of the pattern and the code using the pattern), resulting in increased productivity, enhanced modularity, and reduced dependencies between modules.

The rest of the paper is organized as follows. Section II describes the observer pattern and presents a systematic classification of its possible scenarios. Section III describes the syntax and semantics of the proposed construct of the observer design pattern and how it can be applied. In Section IV, we discuss the characteristics of our approach and present some potential improvements to be considered in the future. Related work is then presented in Section V. Finally, Section VI concludes the paper and suggests possible future work.

II. SCENARIOS OF THE OBSERVER DESIGN PATTERN

The observer design pattern allows monitoring changes in some components of the program, called subjects, to notify other parts of the program, called observers. It consists of two main components: subjects and observers. In general, the observer pattern may define a many-to-many dependency between subjects and observers, in which changes in the states of subjects cause all their respective dependents (i.e., observers) to be notified and updated automatically.

 ISBN: 978-1-61208-438-1

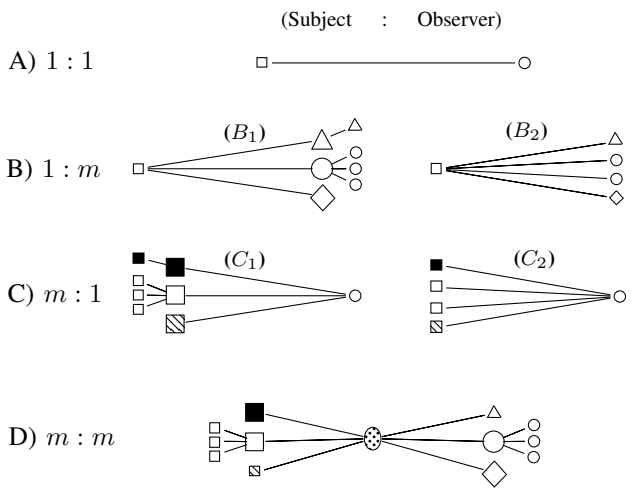

Figure 1. Scenarios of the observer design pattern
[small shape = instance, big shape = class]

Typically, however, the observer pattern represents the case in which a subject maintains a set of observers, and notifies them whenever it has changes in its state [1] (i.e., one subject - many observers). This case is actually limited to one scenario in which the association of observers to subjects is made on basis of subjects. In other words, observing a list of subjects by an observer requires each of these subjects to utilize an individual observing protocol containing a single observer in its list. The proper alternative way to implement such a case would be to have another observing protocol that can associate a list of subjects for any interested observer (i.e., an observer-oriented protocol). Another problem of this implementation is the *instance-level* application in which every instance of an observer class has to explicitly be assigned to the observed subject. This would be better achieved using a *class-level* association of observers to subjects. This means that a subject can be observed by a class, and then all instances of that class will implicitly be assigned to the list of observers of that subject.

Below, we present a systematic classification of the different scenarios of using the observer design pattern.

A. Single Subject - Single Observer

In this case, an observer can only observe a single subject, and the subject can only be observed by one observer as shown in Figure 1(A). This kind of observing is said to be a $1 : 1$ association, where a notification of a state change of the intended subject is sent to the corresponding observer. This scenario is also viable when a certain subject has many attributes, and a certain observer is interested in observing a single attribute of that subject. Therefore, the association of the observer to the subject in this case is also considered as *one-to-one*. This association of a single observer to a single subject can be applied using an *instance-level* observing. However, it can also be applied using a *class-level* observing provided that the observing and observed classes are singletons (i.e., each of them has a single instance).

B. Single Subject - Multiple Observers

This is the common scenario of the observer pattern that describes the case where a single subject can be observed by a set of observers of different types (Figure 1(B)). This means that whenever the subject changes its state, all its dependent observers are notified. For example, when a central database has changes in its data, all dependent applications to this database are notified. In addition, observing a single attribute of a certain subject by many observers is another case of this scenario. This scenario can be applied at two different levels:

1) Class-level: This case happens when a single subject is observed by many observing classes (each with a single instance or multiple instances) as shown in Figure 1(B.B_1). The association of the subject to all corresponding observer classes is said to be $1 : m$. The other case of this scenario occurs when a single subject instance or an attribute of that subject is observed by an observing class with multiple instances as shown in the same sub-figure where the subject is observed by the *circle* class. This leads to the implicit application of the observing logic to all instances of that class to have an association of $1 : m$ as well.

2) Instance-level: Here, a subject instance/attribute can have a list of observing instances (either of the same or different class types) in a $1 : m$ association (demonstrated in Figure 1(B.B_2)). In this case, every instance should explicitly be listed as an observer to the corresponding subject. It should be noted that some instances of the same class may be interested in observing the intended subject while the others may not.

These two levels of association can actually happen together, where a subject can be observed by different class types and at the same time by instances of other classes. Moreover, the *class-level* observing can be applied when all instances of an observing class need to participate in the observing, whereas it is required to apply an *instance-level* observing when only some of the instances are interested in observing that subject.

C. Multiple Subjects - Single Observer

It is common to have one observer that has the responsibility of observing several subjects at the same time. For example, a weather station class may observe different classes for temperature, humidity, wind, etc. As presented in Figure 1(C), this association can be represented as $m : 1$ where the multiple subjects can either be of the same class or different classes. Similar to the previous scenario, *class-level* and *instance-level* observing can be applied in this scenario as shown in Figure 1(C.C_1) and (C.C_2), respectively. Here, an observer instance can observe either a single subject class (all instances of that class are implicitly observed), a list of subject classes, a list of subject instances (from same or different class types), or a set of attributes of a certain subject.

D. Multiple Subjects - Multiple Observers

This scenario encompasses all the previous cases in an $m : n$ association. This kind of association is demonstrated in Figure 1(D) and occurs when several observers intend to observe many subjects. This can also be applied as a *class-level* observing or an *instance-level* observing or both together. When subjects (same or different class types) have more than one attribute to observe, then we might have a combination of several scenarios. An example of this scenario can be described by having a set of class and instance observers interested in observing a class subject, an instance subject and an attribute of a subject.

 ISBN: 978-1-61208-438-1

III. Observer as a Language Construct

In this paper, we propose a novel approach for implementing the observer design pattern as a language extension. This language extension conveys the idea of having an expressive construct that allows explicit application of the pattern using recognizable easy-to-use statements. Actually, such an extension can be built on top of any AO programming language by means of extensible compilers.

In this work, the implementation of the language extension is conducted using the *abc* extensible compiler [5]. *Abc* employs *Polygot* [6] (a Java extensible compiler framework) as a frontend, and extends it with AspectJ constructs. This allows programmers to extend the compiler's syntax and semantics of both: Java and AspectJ. The immediate concrete implementation of the observer construct is based on AO constructs, which is automatically generated using the parameters passed through the construct statements. This implementation is more modular compared with pure OO implementations in Java since it uses the crosscutting facilities provided in AspectJ [7][8]. Modularity in our approach is improved with the use of the high-level and parametrized observer construct that makes using this pattern more expressive and intuitive. At its final stages, *abc* transforms AspectJ Abstract Syntax Tree (AST) into Java AST while preserving the aspect information, and then performs all required weaving with the help of the *Soot* analysis and transformation framework [9] that is used as a backend.

A. Syntax

The observer pattern language construct is designed to be as abstract and modular as it could possibly be while maintaining high accessibility to programmers and users. Moreover, the construct allows applying all possible scenarios of the observer pattern expressively with the least amount of code. Its syntax is defined using the following EBNF notation:

```
<LetObserve>::= "let" <annotated_id_list>
                "observe" <extended_id_list>
                ["exec" <method_invocation>] ";"
```

The observer construct consists of three parts: (1) a list of one or more observers specified by a comma-separated list of class and/or object identifiers given by `<annotated_id_list>`; (2) a list of one or more subjects given by `<extended_id_list>` specified by a comma-separated list of any combination of class and object identifiers and attribute names or even the wildcard (*) to refer to all attributes within the subject to be observed; and finally (3) a single optional notification method given by the `<method_invocation>` non-terminal. Each of the two non-terminals `<annotated_id_list>` and `<extended_id_list>` has its own production rules defined in our extension (as shown below). The `<method_invocation>` and `<name>` non-terminals are already defined in the Java 1.2 parser for CUP [10] employed by *abc*. The production rules that define the non-terminal `<annotated_id_list>` are given as follows:

```
<annotated_id_list>::= <id> {"," <id>}
<id>::= ("class" <name> | <name>)
```

; where the `class` keyword is used to distinguish between class and object identifiers (especially when declared with the same names). The non-terminal `<extended_id_list>` defines an extension to the non-terminal `<id>`. This extension allows programmers to assign subject names, determine certain attributes of them to observe, or use the wildcard ($\star$) to refer to all attributes within a subject to be observed, as follows:

```
<extended_id_list>::= <ext_id> {"," <ext_id>}
<ext_id>::= <id> ["(" ("*" | <attrib_list>) ")"]
<attrib_list>::= <name> {"," <name>}
```

As an example statement that can be generated by this syntax, the following statement:

```
let screen1, class Log observe line1(length), class Point(*);
```

sets up an object $screen1$ and a class Log as observers for changes in length attribute of object $line1$, and any change in state of any object of class $Point$. From now on, we refer to such statements that can be generated by this syntax as '$let - observe - exec$' statements.

In general, the construct can directly support the application of all the scenarios of the observer pattern as described in Section II above (with both: class-level and instance-level observing). In its current implementation, however, scenarios involving mixed usage of *instance-* and *class-level* observing can be specified by multiple separate '$let - observe - exec$' statements, rather than a single statement, which is to be improved upon in future versions of the implementation (See Section IV-C).

B. Application

To show the implementation of the observer construct and how it can be applied, we define three Java classes and several instances of them in Table I: $Line$ and $Point$ as subjects while $Screen$ as an observer. In the $Application$ class, we create some instances of these classes to utilize them in the *instance-level* application of the construct. Some scenarios of the observer pattern require all instances of a class to observe subjects (i.e., *class-level* observing), while some others need every instance to have its own observing logic (i.e., *instance-level* observing). The observer construct provides both *class-* and *instance-level* observing. The general structure of the observer construct is as follows: observers (classes and instances) are placed after the let keyword, subjects (classes and instances) after the $observe$ keyword, and, optionally, the notification method after the $exec$ keyword.

1) Class-level Observing: The *class-level* observing can be applied as follows:

```
let class Screen observe class Line, class Point;        (-1-)
```

In this kind of observing, programmers can indicate that one class is observing a subject class or a set of subject classes. Consequently, all instances of the observing class will be notified when any instance of the subject(s) has state changes. This application shows a case of the *class-level* version of *Multiple Subjects - Multiple Observers* scenario that is applied using only one statement.

2) Instance-level Observing: The observing logic in the *instance-level* version of the observer pattern is accomplished instance-wise. This means that each constructed object of the observing class may observe various subjects with a different number of attributes of each subject. One form of this kind of observing is to observe a single attribute of a single subject, as follows:

```
let screen1 observe line(length) exec resize(length);   (-2-)
```

 ISBN: 978-1-61208-438-1

TABLE I. FOUR JAVA CLASSES: TWO SUBJECTS, AN OBSERVER, AND AN APPLICATION

First Subject Class

```
class Line {
  Color color;
  int length;
  void setLength(int len){
    this.length = len; }

  void setColor(Color c){
    this.color = c; }
}
```

Second Subject Class

```
class Point {
  int x, y;
  void setPos(int x, int y){
    this.x = x;
    this.y = y; }
}
```

Observer Class

```
class Screen {
  public void resize(int len){
    System.out.println("Resizing
                  with the new length: " + len); }
  public void display(String str){
    System.out.println(str); }
}
```

Application

```
Line line = new Line();
Point point = new Point();
Screen screen1, screen2, screen3 = new Screen();
```

This case refers to the *Single Subject - Single Observer* scenario in which the programmer has to specify the observing instance, the subject and the notification method that will receive the change of the state of the specified attribute of the subject and send it directly to the corresponding observer. Another form is to observe multiple attributes of single subject by one observing instance. This form represents the *Multiple Subjects - Single Observer* scenario with the case of observing many attributes of a subject using one statement, as shown in the following application:

```
let screen2 observe line(color,length) exec display;    (-3-)
```

The restriction of this application is that the programmer has to define only one notification method (with a $String$-type parameter) to refresh the observing instance with the state changes of all attributes of the subject. If the programmer did not specify the notification method, the compiler is built to assume that there exist a method called '$display$' in the observing class will do the job.

Last form is to observe multiple subjects with all their attributes using one statement as shown below. This form also represents the *Multiple Subjects - Single Observer* scenario but now with the case of having many subjects with either single or multiple attributes per each. This could be accomplished by either not specifying the attributes at all, or by using the wildcard (*) to refer to all attributes. With respect to specifying the notification method, cases of the previous form also apply here.

```
let screen3 observe line, point(*);                     (-4-)
```

C. Semantics and Code Translation

After parsing '$let - observe - exec$' statements and matching them with the given syntax of the construct, the compiler then moves into other compilation passes that are concerned with the construct semantics. During these passes (with the help of the type system), the compiler starts recognizing class types, instances, attributes and methods used in the construct application by carrying out scoping and type-checking operations. If such checking is passed successfully, the compiler then carries out the code conversion (or rewriting). Otherwise, a semantic exception is generated by the compiler.

1) Variable Scoping: The compiler checks the validity of each element of the observer construct (i.e., classes, instances, attributes and the notification method) to see whether they are not defined or out-of-scope. The compiler in such cases will generate a semantic exception. Another check is conducted when the construct is applied without specifying a notification method. In this case, a programmer has to define a notification method named $display$ in the observing class to be responsible for refreshing it with the changes happened. If such a method is not defined, the compiler will also produce a semantic exception.

2) Type checking: In this process, the compiler is going to pick the class included in the observer construct, and checks its eligibility. For instance, when the programmer uses an observer construct for primitive types, the compiler will check and produce an appropriate alert message showing that only classes or instances can be applied. Also, when programmer use the *instance-level* observing form, then the argument type of the notification method must match the type of the observed attribute. For the case of applying the construct with a default notification method, the compiler would expect programmers to define a method called $display$ in observing class that accepts the changes as a $String$ type.

3) Node Translation and Code Conversion: After achieving all checks successfully, the compiler starts converting LetObserve nodes into their corresponding aspect declaration nodes that the original AspectJ compiler can deal with. This node translation is actually executed through a code conversion pass of the compiler where each '$let - observe - exec$' statement is converted into a specialized aspect that contains the proper crosscutting concerns of the observing statement as shown in Table II.

Every auto-generated aspect is assigned a name of the form '$ObserverProtocol\_\#$', where the hash symbol refers to a sequence number that will be assigned for each auto-generated observing aspect. The newly generated node (i.e., the aspect declaration) is created outside the class that contains the application of the observer construct. Indeed, aspects generated for *class-level* observing purposes have a different implementation style from the ones used for *instance-level*.

- ***Class-Level* Observing**: As shown in Table II.A, an aspect is generated for the '$let - observe - exec$' statement (1). This aspect implements the observing logic for all instances of the supplied observer class in the statement. Therefore, a list of observers (Line 3) is employed to hold a reference copy for every newly created object of that observer class. Object construction joinpoints are crosscutted using the pointcut declared in Lines 5-6 and are advised in Lines 8-10. Whenever a subject has changes on its associated attributes, the $subjectChange$ pointcut (declared in Lines 12-14) will be executed. Consequently, every instance of that observing class will be notified (this task is accomplished by the advice declared in Lines 16-24). After a successful generation of the desired aspects, the compiler replaces '$let - observe - exec$' statements by empty statements (i.e., semicolons ';').
- ***Instance-Level* Observing**: In *instance-level* observing, an aspect is also generated for the '$let-observe-exec$' statement (2) as shown in Table II.B. This aspect

TABLE II. AUTO-GENERATED ASPECT FOR THE OBSERVER PATTERN CONSTRUCT

A. *Class-level* Observing

```
1  protected privileged aspect ObserverProtocol_1
2  {
3     private List observers = new ArrayList();
4     //---------------------------------------
5     protected pointcut newInstance(Screen obs):
6       execution(Screen.new(..)) && target(obs);
7
8     after(Screen obs): newInstance(obs){
9        observers.add(obs);
10    }
11    //---------------------------------------
12    protected pointcut subjectChange() :
13       set(* Line.*)  ||
14       set(* Point.*) ;
15
16    after(): subjectChange() {
17       Iterator it = observers.iterator();
18       while (it.hasNext()){
19          Screen obs = (Screen)it.next();
20          obs.display(
21                thisJoinPoint.getSignature() +
22                                " changed..");
23       }
24    }
25 }
```

B. *Instance-level* Observing

```
1  protected privileged aspect ObserverProtocol_2
2  {
3     private Screen obs;
4
5     public void addObserver(Screen obs) {
6        this.obs = obs;
7     }
8     //---------------------------------------
9     public interface Subject {}
10
11    declare parents: Line      implements  Subject;
12
13    protected pointcut subjectChange(Subject s) :
14       (
15          set(* Line.length)
16       ) && target(s);
17
18    after(Subject s): subjectChange(s) {
19         obs.resize(((Line) s).length);
20    }
21 }
```

has only one observer field (Line 3) that holds a reference copy of the observing instance that will be assigned via the *addObserver* method, which will be invoked at the client application (In particular, at the line(s) where the '$let - observe - exec$' statement is written in the source code). Once the subject has changes in its attributes, the $subjectChange$ pointcut declared in Lines 13-16 is executed. As a result, the observing instance is notified (the advice declared in Lines 18-20 will do this task) using the notification method that was already associated with the statement of the observer construct. In addition, this aspect has a public $Subject$ interface (Line 9) that will be implemented by all observed (Subject) classes. This interface can then used in place of subject classes to capture changes of any subject implementing it. After generating this aspect successfully, the '$let - observe - exec$' statement is replaced by a method-call statement, as follows:

```
ObserverProtocol_2.aspectOf().addObserver(screen1);
```

IV. RESULTS AND DISCUSSION

After implementing our language extension to the examples presented in this paper, we have addressed some of the issues discussed in the literature related to modularity and implementation overhead, and describe how they are handled (at least partially) in our approach. Furthermore, we could identify the characteristics of the proposed observer construct in addition to some future improvements to it.

A. Addressed issues

1) Implementation Overhead: This issue was occurred with several traditional implementations of design patterns (such as, the use of OO or AO constructs) as programmers have to have in mind how the implementation of design patterns should work with their functional code. In our approach, the programmer is not concerned about the concrete implementation of the pattern since it is automatically generated by the extended compiler based on the parameters provided via the pattern construct statements. This lets programmers save time and space and, subsequently, focus on their functional parts of the code (i.e., enhanced productivity). Through the examples illustrated in this paper (and other not reported examples), we strongly believe that our approach will outperform other implementations of the observer pattern proposed in the literature in terms of lines of code (LOC) if applied to larger applications.

Although in Meta-AspectJ [11] programmers can abstract the overall implementation of the observer pattern in AspectJ with fewer lines of code, this would end up with a complex (not expressive) abstraction that imposes users to be aware of the aspects that would be generated. In our approach, programmers are not aware of what is happening inside the aspects. All what they need in our approach is to specify the observing/observed classes or instances along with the desired attributes and notification methods.

2) Modularity: In our approach, modularity is witnessed by separating the implementation of the observer design pattern from the implementation of the actual logic of the application. This means that the actual implementation of the observer pattern is not visible to the programmer and it is also isolated from one application to another. This allows programmers (at different clients) extend, alter and maintain their applications of this design pattern modularly without being aware of what is happening in the background.

It can be observed that our implementation of the observer design pattern satisfies all modularity properties (i.e., locality, reusability, composition transparency, and (un)pluggability) firstly used by Hannemann and Kiczales [3], and thereafter used by Rajan [4], Sousa and Monteiro [12], and Monteiro and Gomes [13]. Our implementation is localized since the overall implementation code of the observer pattern is automatically generated in the compiler background, which means that it is totally separated from the pattern application. It is also reusable because it can be applied to various scenarios without the need to duplicate the source code. Composing a class or an instance in our observer construct will not interfere at all with other classes or instances. Finally, adding (plugging) or removing (unplugging) an application of the observer pattern in a given system using the proposed construct will not require programmers to do changes on other parts of that system.

 ISBN: 978-1-61208-438-1

B. Features

1) Hybrid approach: Our approach combines different features of the approaches proposed in the literature under one roof. It is implemented as a Java/AspectJ extension that summarizes plenty of code in few-keywords constructs, just like meta-AspectJ [11]. Additionally, it automatically generates aspects according to the information provided as parameters in applied construct, adapted from the parametric aspects [14].

2) Expressiveness: The syntax of design patterns is clear, concise and expressive in a way it does not require importing packages, building classes (or aspects), or worrying about something missing in the design principle of the design pattern. All what programmers need to learn in our approach is the construct syntax used for implementing the observer pattern, and also how to apply each scenario using that construct. Furthermore, the readability and writeability is highly improved as the written code becomes shorter and more self-explanatory. So, the absence of dependencies makes it very easy to revise the code for the sake of maintenance.

3) Supporting different levels of application: Supporting different levels of applications of the constructs helps programmers decide where and how to apply constructs. The *class-level* application is beneficial if all instances of a certain class needs to apply the observing functionality, whereas *instance-level* application is useful when certain instances of a class need to apply the observing logic, or when each instance needs to have its own logic.

C. Improvement Considerations in the Future

1) Optimization: As described in the paper, our extended compiler creates a separate aspect for each application of the pattern construct. This potential duplication of generated aspects would require more processing from the compiler, since each aspect node can contain a set of internal AST nodes, which may lead to more compiler passes to be executed. This problem can be resolved in the future by generating a single observer-protocol aspect to handle the implementation of all applications of the observer construct, by employing a particular pointcut and advice for every construct application.

2) Application of the other scenarios: Although the observer construct itself is general enough to capture any observing behavior directly, the current implementation of it does not allow intermixing *class-level* and *instance-level* observing scenarios in a single statement. This means that the implementation requires such scenarios to be specified by more than one '$let - observe - exec$' statement). For example, if observing one subject by multiple observers is needed, then the programer will have to apply the observer construct to each observer with the intended subject (i.e., it will be applied as a set of one-to-one scenarios). Alternatively, the construct should be improved in future to support the other scenarios using single-statement applications.

3) Disabling of pattern application: In our approach, programmers can apply constructs anywhere in their programs. However, to disable a pattern for a certain target, programmers are required to search for the construct application in the program and then comment or remove it. This kind of disabling suffers from a traceability overhead, as the efficient way to this end is to have disable/enable constructs in the future that can automate this action.

V. Related Work

Hannemann and Kiczales [3] used AO constructs to improve the implementations of the original 23 design patterns using AspectJ. They provided an analysis and evaluation of the improvement achieved to the implementation of the patterns according to different metrics, which also have been addressed later by Rajan [4] using Eos extended with the *classpect* construct that unifies class and aspect in one module. When compared with Hannemann's implementation in terms of lines of code and the intent of the design patterns, Rajan observed that Eos could efficiently outperform AspectJ in implementing 7 of the design patterns, while being similar for the other 16 patterns. In addition, the *instance-level* advising feature supported by Eos *classpects* was another advantage over AspectJ. This feature allows a direct representation of runtime instances without the need to imitate their behavior. Another work was also done by Sousa and Monteiro [12] with CaesarJ that supports family polymorphism. Their approach employs a *collaboration* interface that can hold a set of inner abstract classes, and some second level classes: the implementation and binding parts. Also, their results demonstrated positive influence of the *collaboration* interface on modularity, generality, and reusability over those with AspectJ. Gomes and Monteiro [15] and recently in [13] introduced the implementation of 5 design patterns in *Object Teams* compared with that in Java and AspectJ. Regardless of *Object Teams* goals, it showed a powerful support in implementing design patterns efficiently, and with more than one alternative. The entire conversion of aspects into teams was described in detail in their work. The common issue with all these different approaches is that they suffer from the implementation overhead and traceability problems as the concrete implementation of design patterns is required to be manually written by programmers, which may reduce their productivity.

Another approach was introduced by Zook et al. [11]. This approach uses code templates for generating programs as their concrete implementation, called Meta-AspectJ (*MAJ*). Development time is reduced in this approach since it enables expressing solutions with fewer lines of code. With respect to design patterns, *MAJ* provides some general purpose constructs that reduce writing unnecessary code. However, programmers cannot explicitly declare the use of design patterns at certain points of the program, which may also lead to a traceability problem.

Another trend, which is close to our approach, was introduced by Bosch [2], who provided a new object model called *LayOM*. This model supports representing design patterns in an explicit way in C++ with the use of layers. It provides several language constructs that represent the semantics of 8 design patterns and can be extended with other design patterns. Although *LayOM* could resolve the traceability problem and enhance modularity, it lacks expressiveness as it has a complicated syntax consisting of message forwarding processes that might confuse programmers. Our approach seems to provide a similar power to *LayOM*, but, in contrast, the observer construct in our approach has a more concise, expressive, easy-to-use and easy-to-understand syntax.

Hedin [16] also introduced a new technique that is slightly similar to *LayOM* but using rules and pattern roles. The rules and roles can be defined as a class inheritance and specified by attribute declarations. Doing so, it enables the

 ISBN: 978-1-61208-438-1

extended compiler to automatically check the application of patterns against the specified rules. However, the creation of rules, roles, and attributes has a complex syntax that lacks expressiveness and requires an extensive effort to learn and build them.

Another extensible Java compiler is *PEC*, which was proposed by Lovatt et al. [17]. Design patterns in *PEC* were provided as marker interfaces. A class must implement the proper ready-made interface in order to conform to a certain design pattern. After that, the *PEC* compiler will have the ability to check whether the programmer follows the structure and behavior of that pattern or not. However, the *PEC* compiler does not reduce the effort needed to implement design patterns (i.e., it suffers from implementation overhead). Instead, it allows programmers to assign the desired design pattern to a given class and then implement that pattern manually. Eventually, the compiler will just check the eligibility of that implementation.

Budinsky et al. [18] introduced a tool that automates design pattern implementation. Each design pattern has a certain amount of information like name, structure, sample code, when to use, etc. The programmer can supply information about the desired pattern, then its implementation (in C++) will be generated automatically. This approach allows programmers to customize design patterns as needed, but the modularity and reusability is missed, and it suffers from the traceability problem as well.

VI. Conclusion

This paper introduces two contributions in regards to the observer design pattern. Firstly, it presents a detailed classification of all possible scenarios of the observer pattern that might be utilized in various kinds of applications. Secondly, a new approach for implementing the observer pattern in Java is proposed to cover a partial set of the scenarios introduced. This approach is developed as a language extension (Java/AspectJ extension) using *abc*. The syntax, semantics and application of the proposed observer construct are illustrate in detail, and by means of typical examples, we demonstrate how the implementation of the observer pattern using this approach has been simplified and has become conciser, more expressive and more modular. The capabilities and advantages of the proposed approach seem promising and we anticipate that our approach will supersede current approaches.

We hope in the future to improve this approach by following the recommendations provided in the paper. This includes supporting the application of other scenarios of the observer pattern using a single statement rather than many. Resolving current issues of the observer pattern implementation such as optimization and application disabling will also be taken into account in the future. Furthermore, an evaluation of our proposed approach compared to with other approaches proposed in the literature is another important objective that we hope to achieve in future. Moreover, we aim to develop constructs for implementing other software design patterns to make their implementations more expressive and modular.

References

[1] J. Vlissides, R. Helm, R. Johnson, and E. Gamma, "Design patterns: Elements of reusable object-oriented software," Reading: Addison-Wesley, vol. 49, 1995, p. 120.

[2] J. Bosch, "Design patterns as language constructs," Journal of Object-Oriented Programming, vol. 11, no. 2, 1998, pp. 18–32.

[3] J. Hannemann and G. Kiczales, "Design pattern implementation in Java and AspectJ," in ACM Sigplan Notices, vol. 37. ACM, 2002, pp. 161–173.

[4] H. Rajan, "Design pattern implementations in Eos," in Proceedings of the 14th Conference on Pattern Languages of Programs. ACM, 2007, pp. 9:1–9:11.

[5] P. Avgustinov et al., "abc: An extensible aspectj compiler," in Transactions on Aspect-Oriented Software Development I. Springer, 2006, pp. 293–334.

[6] N. Nystrom, M. R. Clarkson, and A. C. Myers, "Polyglot: An extensible compiler framework for Java," in Compiler Construction. Springer, 2003, pp. 138–152.

[7] A. Mehmood and D. N. Jawawi, "Aspect-oriented model-driven code generation: A systematic mapping study," Information and Software Technology, vol. 55, no. 2, 2013, pp. 395–411.

[8] N. Cacho et al., "Blending design patterns with aspects: A quantitative study," Journal of Systems and Software, vol. 98, 2014, pp. 117–139.

[9] R. Vallée-Rai et al., "Optimizing java bytecode using the soot framework: Is it feasible?" in Compiler Construction. Springer, 2000, pp. 18–34.

[10] "Java 1.2 parser for CUP." [Online]. Available: https://github.com/Sable/abc/blob/master/aop/abc/src/abc/aspectj/parse/java12.cup Last access: September 15, 2015.

[11] D. Zook, S. S. Huang, and Y. Smaragdakis, "Generating AspectJ programs with meta-AspectJ," in Generative Programming and Component Engineering. Springer, 2004, pp. 1–18.

[12] E. Sousa and M. P. Monteiro, "Implementing design patterns in CaesarJ: an exploratory study," in Proceedings of the 2008 AOSD workshop on Software engineering properties of languages and aspect technologies. ACM, 2008, pp. 6:1–6:6.

[13] M. P. Monteiro and J. Gomes, "Implementing design patterns in Object Teams," Software: Practice and Experience, vol. 43, no. 12, 2013, pp. 1519–1551.

[14] K. Aljasser and P. Schachte, "ParaAJ: toward reusable and maintainable aspect oriented programs," in Proceedings of the Thirty-Second Australasian Conference on Computer Science-Volume 91. Australian Computer Society, Inc., 2009, pp. 65–74.

[15] J. L. Gomes and M. P. Monteiro, "Design pattern implementation in Object Teams," in Proceedings of the 2010 ACM Symposium on Applied Computing. ACM, 2010, pp. 2119–2120.

[16] G. Hedin, "Language support for design patterns using attribute extension," in Object-Oriented Technologys. Springer, 1998, pp. 137–140.

[17] H. C. Lovatt, A. M. Sloane, and D. R. Verity, "A pattern enforcing compiler (PEC) for Java: using the compiler," in Proceedings of the 2nd Asia-Pacific conference on Conceptual modelling-Volume 43. Australian Computer Society, Inc., 2005, pp. 69–78.

[18] F. J. Budinsky, M. A. Finnie, J. M. Vlissides, and P. S. Yu, "Automatic code generation from design patterns," IBM Systems Journal, vol. 35, no. 2, 1996, pp. 151–171.

 ISBN: 978-1-61208-438-1