International Journal of Software Engineering and Knowledge Engineering

https://doi.org/10.1142/S0218194021500327

An Extensible Compiler for Implementing Software Design Patterns as Concise Language Constructs

Taher Ahmed Ghaleb

School of Computing, Queen's University, Kingston, Canada
taher.ghaleb@queensu.ca

Khalid Aljasser

Information and Computer Science Department
King Fahd University of Petroleum and Minerals, Dhahran 31261, Saudi Arabia

Musab A. Alturki

Runtime Verification Inc., Urbana IL 61801, USA
King Fahd University of Petroleum and Minerals, Dhahran 31261, Saudi Arabia

Design patterns are generic solutions to common programming problems. Design patterns represent a typical example of design reuse. However, implementing design patterns can lead to several problems, such as programming overhead and traceability. Existing research introduced several approaches to alleviate the implementation issues of design patterns. Nevertheless, existing approaches pose different implementation restrictions and require programmers to be aware of how design patterns should be implemented. Such approaches make the source code more prone to faults and defects. In addition, existing design pattern implementation approaches limit programmers to apply specific scenarios of design patterns (e.g., class-level), while other approaches require scattering implementation code snippets throughout the program. Such restrictions negatively impact understanding, tracing, or reusing design patterns. In this paper, we propose a novel approach to support the implementation of software design patterns as an extensible Java compiler. Our approach allows developers to use concise, easy-to-use language constructs to apply design patterns in their code. In addition, our approach allows the application of design patterns in different scenarios. We illustrate our approach using three commonly used design patterns, namely Singleton, Observer, and Decorator. We show, through illustrative examples, how our design pattern constructs can significantly simplify implementing design patterns in a flexible, reusable, and traceable manner. Moreover, our design pattern constructs allow class-level and instance-level implementations of design patterns.

Keywords: Design patterns; singleton; observer; decorator; aspect-oriented programming; extensible compiler; Polyglot; abc.

1. Introduction

Object-oriented (OO) design patterns [1] are reusable solutions that restructure OO programs in a well-organized and reusable design. Design patterns were originally implemented using OO features, such as polymorphism and inheritance [2]. After the

emergence of aspect-oriented (AO) programming languages, researchers started to employ AO constructs to make the implementation more reusable and modular [3]. There are three categories of design patterns: creational, structural, and behavioral.

Despite the wide range of applications of design patterns, the manual implementation of design patterns may lead to several problems. For example, design pattern implementation may cause programming efficiency, traceability, and code reusability problems [4]. Programmers may need to write various classes and methods to achieve a simple behavior of a design pattern. Such a way leads to a sizable programming overhead, a scattering of actions everywhere in the program, and a reduction in program understandability. Although design patterns make the design reusable, the code (or at least part of the code) used to implement them cannot be reused later. Complex design patterns are implemented differently from one approach to another, such as Observer and Decorator. The implementation of simple design patterns, such as Singleton and Facade, only varies slightly from prior approaches.

Mayvan et al. [5] conducted a systematic study on research directions of design patterns. Previous approaches to implement design patterns make the source code more prone to faults and defects [6, 7, 8, 9]. In addition, existing design pattern implementation approaches limit programmers to apply specific scenarios of design patterns (e.g., class-level), while other approaches require scattering implementation code snippets throughout the program. Such restrictions negatively impact understanding, tracing, or reusing design patterns [10, 11]. Despite the practical usage and development of design patterns, research has paid little attention to supporting multiple implementation scenarios of design patterns while maintaining a reusable code. For example, the Observer design pattern has been implemented in the literature using only one scenario (i.e., a single subject with multiple observers). In such a scenario, the association of observers to subjects is subject-driven. For example, Observer is implemented using the *Line*, *Point* and *Screen* example provided by [3, 12], in which a single subject has a multiple of observers. However, this observing example has instead multiple subjects (i.e., *Lines* and *Points*) observed by a single observer (i.e., *Screen*). Moreover, conventional approaches employ an indirect way to implement design patterns. In other words, programmers in conventional approaches do not deal with a design pattern as a recognizable unit in programs. Instead, programmers may be required to manually implement, or at least to be aware of, certain design pattern protocols and apply them to every pattern instance. Such a way of protocol implementation may lead to increased dependencies within programs, which makes the code error-prone.

In this paper, we propose an extensible compiler that makes the implementation of design patterns simpler, more intuitive, and easier to apply (patented in [13, 14, 15]). We build an extensible compiler that implements our approach. Our approach significantly simplifies the application of design patterns and makes it more explicit. Our approach promotes code correctness by reducing the chances of making programming errors in both the implementation of a design pattern or the

code using it. Our approach can result in increased productivity, enhanced modularity, and reduced dependencies between modules. We implement our approach as an extensible Java compiler using *abc* (AspectBench Compiler) [16] with the *Polyglot* frontend [17]. To demonstrate our approach using three commonly used design patterns, namely Singleton (a creational design pattern), Observer (a behavioral design pattern), and Decorator (a structural design pattern). The selected design patterns convey various concerns that have been addressed in the literature. Our approach supports two levels of application of design patterns: the *class-level* and the *instance-level*. We compare our approach with a baseline approach proposed by Hannemann and Kiczales [3], which uses AspectJ constructs. Our approach outperforms the baseline approach in terms of lines of code, reduced dependencies, reduced implementation overhead, and supporting *instance-level* implementations.

An application of the approach proposed in this paper to the Observe design pattern was previously published. In this paper:

- We present the design of our extensible compiler and the required steps required to implement new design patterns as language constructs.
- We present an application of the proposed approach to the Singleton and Decorator design patterns, in addition to the Observer design pattern that was presented in our previous work.

The rest of this paper is organized as follows. Section 2 presents the design of our approach. Sections 3, 4, and 5 present the implementation of the Singleton, the Observer, and the Decorator design patterns, respectively, using our proposed approach. Section 6 discusses the prospective features of our approach. Section 7 presents the related work in the literature. Finally, Section 8 concludes the work and suggests possible future work.

2. The Proposed Approach

We propose an approach to implement design patterns as an extensible compiler. Our approach aims to reduce the programming overhead of design patterns and to improve the traceability and readability of design patterns. To this end, we develop a set of language constructs that allow programmers to apply design patterns easily. For each language construct, we define keywords, syntax, and semantics. Then, we develop our language constructs as a Java compiler extension using the *abc* compiler. In this section, we present the design of our approach.

2.1. *Design principles*

When we design our design pattern constructs using our proposed extensible compiler, we define a set of design principles as follows:

- **Flexibility:** our approach encourages building parametric constructs to allow the implementation of different scenarios of design patterns. For example, constructs can be designed to support *class-level* and *instance-level*

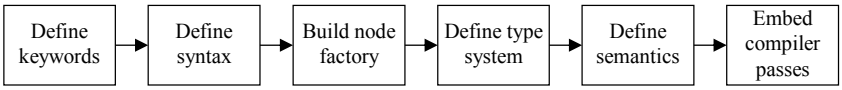

Fig. 1. Our methodology for extending the compiler

application of a certain design pattern. In addition, a design pattern construct should be designed to accept passing parameters by programmers, such as the names of specific classes, attributes, or methods.

- **Ease-of-use:** our approach enables developers to build expressive constructs that precisely capture the objective of design patterns in an intuitive manner.
- **Modularity:** our approach enables developers to build modular constructs that do not rely on other dependencies. For example, to apply a certain design pattern, programmers are not required to use other libraries or implementations of that design pattern, since everything is inclusive in the design pattern construct.

2.2. *Methodology of Extending the Compiler*

Fig. 1 depicts the methodology used in our approach to develop design pattern constructs. To develop a language construct for a certain design pattern, developers need to design appropriate (a) keyword(s), (b) syntax, (c) Abstract Syntax Tree (AST) node(s), (d) type system, (e) semantics, and (f) compiler passes. This methodology can be followed to design language constructs for any design patterns. In this paper, we describe each step in detail using the Singleton, Observer, and Decorator design patterns.

2.2.1. *Identification of Keywords:*

We define the keywords we use in our proposed design pattern constructs. We extend the *lexer* of the base Java compiler to include all our desired keywords. We extend the compiler's *lexer* in two steps. First, we create a list of *tokens* to represent our desired keywords as grammar terminals, as follows:

```
terminal Token SINGLETON, INSTANTIATE, AS,
                LET, OBSERVE, EXEC,
                DECORATE, WITH, TO;
```

Second, we extend the compiler's *lexer* by defining appropriate keywords and linking such keywords to the grammar tokens.

2.2.2. *Extending the Parser and Grammar:*

A grammar or syntax contains a set of formal rules that define how to construct new expressions of the language from a set of symbols. There are formal ways to

describe syntax in programming languages, such as Context-free grammar (CFG), Backus-Naur form (BNF), and extended BNF (EBNF) [18]. In our extensible compiler, considering that we use the *abc* extensible compiler with the *Polyglot* frontend, we extend the Polyglot Parser Generator (PPG). Grammar rules in PPG are represented using EBNF. In our extensible compiler, we extend the compiler's PPG grammar with new grammar rules that define how our design pattern constructs should be used by programmers. We make the syntax of our design pattern constructs simple but expressive and flexible to support different implementation scenarios. We extend Java grammar rules by adding our design pattern constructs as follows:

```
extend class_declaration ::= <singleton_decl>;
extend statement_without_trailing_substatement ::=
    <instantiate_stmt>
  | <let_observe_stmt>
  | <decorate_with_stmt>;
```

Each grammar rule is then linked with a certain AST *Node* in our extended *Node Factory*.

2.2.3. *Constructing the Node Factory and AST:*

The Node Factory of *abc* extends the Polyglot's Node Factory with AspectJ nodes. We extend the *abc*'s Node Factory with nodes that represent the components of our design pattern constructs. Each node in the Node Factory is responsible for accepting the parameters of a design pattern construct and creating the corresponding AST.

2.2.4. *Specifying the Type System:*

We define the type system of our constructs by extending the *abc*'s type system. Our types are generated automatically during the type building pass and then translated into appropriate base types in the following compiler pass.

2.2.5. *Semantics:*

We define the semantics that corresponds to each of our constructs, in which we specify how our constructs are transformed into AspectJ constructs. The AspectJ constructs are then transformed into native Java constructs.

2.2.6. *Embedding Our Compiler Passes:*

Compiler passes are a sequence of jobs that build, check, and transform the AST. We extend the *abc*'s compiler passes the same way *abc* extends the polyglot's compiler passes for Java. Each compiler pass is associated with a compiler visit that runs whenever a compiler pass is reached.

3. The Singleton Design Pattern

Singleton is a creational design pattern that allows classes to have only a single instance. All objects instantiated from a Singleton class refer to the same class instance. This design pattern is preferable when a system needs exactly one instance to maintain all required actions. Singleton contains a *private* instance and a *private* constructor. The constructor is declared *private* to prevent the instantiation of other class instances. In addition, Singleton maintains a public method that returns a single class instance.

3.1. *Syntax of the Singleton construct*

Our Singleton construct allows programmers to simply write the '*singleton*' modifier in the class declaration. To obtain a single instance of a Singleton class, programmers can use our '*instantiate*' keyword. Our syntax for the Singleton construct is as follows:

```
<singleton_decl> ::= [<modifiers>] "singleton" "class" <class_id> ["extends" <super_class>]
                     ["implements" <interfaces_list>] <class_body>
<instantiate_stmt> ::= "instantiate" <class_id> "as" <var_decls>;
```

3.2. *Applying the Singleton construct*

The following code snippet shows how to use our Singleton construct. The class *Sing* has the '*singleton*' modifier, which restricts it to have only one instance. To obtain a reference to that instance, we use the '*instantiate*' statement that gets a copy of the single instance of *Sing* and assigns it to the provided objects.

```
public singleton class Sing
{
   instantiate Sing as s1;
   public static void main(String[] args)
   {
      instantiate Sing as s2, s3;
   }
}
```

3.3. *Semantics of the Singleton construct*

The compiler parses '*singleton*' class declaration and the '*instantiate*' statement and then matches them with the extended compiler syntax. If the code complies with our extended syntax, the compiler proceeds to the next compiler pass. Otherwise, the compiler complains and produces appropriate error messages. We maintain proper variable scoping for '*instantiate*' statements to make sure that the Singleton class used is within the scope and could be accessed. In addition, we define type checking compiler pass to verify whether the '*singleton*' modifier is only used for class declarations. Moreover, we make sure that '*instantiate*' statements are

used with class types declared with the '*singleton*' modifier. Finally, if all semantic checks pass, the compiler transforms the Singleton AST nodes to proper AspectJ nodes, which in turn are transformed into Java AST nodes, as shown in Table 1.

Table 1. Auto-Generated Code for the Singleton design pattern

```
1  public class Sing{
2      private static Sing singleInstance = new Sing();
3      private Sing(){}
4      public static Sing getInstance(){
5        return singleInstance;
6      }
7      public static void main(String[] args){
8        Sing s1 = Sing.getInstance();
9        Sing s2 = Sing.getInstance();
10       Sing s3 = Sing.getInstance();
11     }
12 }
```

Programmers are unaware of this code transformation, since it is performed in the compiler back-end before producing the program byte code. Hence, our generated code is considered error-free and semantically correct.

4. The Observer Design Pattern

Observer is a behavioral design pattern that allows monitoring changes in some components of the program, called subjects, to notify other parts of the program, called observers. Observer 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. However, the conventional implementation of the Observer design has two problems:

- Observer maintains a set of observers to notify whenever a state change occurs (i.e., one subject - many observers) [2]. Such a case is limited to one scenario in which the association of observers to subjects is made on the basis of subjects. Therefore, observing a list of subjects by a single observer requires each of these subjects to utilize an individual observing protocol with only one observer in its list. A better alternative to implement such a scenario would be to have another observing protocol that can associate a list of subjects to any particular observer (i.e., an observer-oriented protocol).
- In the *instance-level* application, every instance of the Observer class has to explicitly be assigned to the observed subject. Using a *class-level* association of observers to subjects would rather solve such limitation. The *class-level* association allows a subject to be observed by a class, and then all instances of that class are implicitly assigned to the list of observers of that subject.

We provide more details about the possible application scenarios of the Observer design pattern in [19]. Hannemann and Kiczales [3] proposed an AspectJ implementation of the Observer design pattern in comparison with the native Java implementation. Their implementation was based on AspectJ constructs that improve modularity using aspect-oriented crosscutting facilities [20, 21]. In our extensible compiler, we use the AspectJ implementation proposed by Hannemann and Kiczales as a back-end implementation of our proposed constructs.

4.1. *Syntax*

The Observer design pattern construct is designed to be as abstract and modular as it could possibly be while maintaining high accessibility to programmers. Moreover, the construct allows applying all possible scenarios of the Observer design 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[a] 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>)
```

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 (`*`) 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>}
```

[a]https://github.com/Sable/abc/blob/master/aop/abc/src/abc/aspectj/parse/java12.cup

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 *screen*1 and a class *Log* as observers for changes in length attribute of object *line*1, and any change to the state of any object of the *Point* class. From now on, we refer to such statements that can be generated by this syntax as '*let* − *observe* − *exec*' statements.

In general, the Observer construct supports the application of all the *class-level* and *instance-level* observing scenarios, namely *one-to-one*, *one-to-many*, *many-to-one*, and *many-to-many* [19]. Still, implementing a mixture of *instance-* and *class-level* observing may be specified using multiple '*let* − *observe* − *exec*' statements.

4.2. *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 2: *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 design 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.

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

  ```
  let class Screen observe class Line, class Point;
  ```

  In the above observing form, programmers are able to indicate that one class is observing a subject class or a set of subject classes. Consequently, all instances of the observing class are notified whenever an instance of the subject(s) has its state changed. This application shows a case of the *class-level* version of *Multiple Subjects - Multiple Observers* scenario that is applied using only one statement.
- **Instance-level Observing:** The observing logic in the *instance-level* version of the Observer design 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 observing is to observe a single attribute of a single subject, as follows:

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

Table 2. 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 receives 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 a 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 in one statement, as shown in the following application:

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

The restriction of this application is that the programmer has to define only one notification method (with a *String* parameter) to refresh the observing instance with the state changes of all attributes of the subject. If the programmer does not specify a notification method, the compiler is built to assume that there exists a method called '*display*' in the observing class for that purpose.

The 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(*);
```

Table 3. Auto-generated Aspect for the Observer design 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(thisJoinPoint.getSignature() + " changed..");
21        }
22     }
23  }
```

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 }
```

4.3. *Semantics*

After parsing '*let* − *observe* − *exec*' statements and matching them with the given syntax of the Observer construct, the compiler then moves to 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 Observer construct application by carrying out variable scoping, type-checking, and node translation. If such checking is passed successfully, the compiler then carries out the code transformation (or rewriting). Otherwise, a semantic error is generated by the compiler.

4.3.1. *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 generates a semantic error. 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 that happened. If such a method is not defined, the compiler also produces a semantic error.

4.3.2. *Type checking:*
In this process, the compiler picks 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 checks and produces an appropriate alert message showing that only classes or instances can be applied. Also, when a programmer uses the *instance-level* observing form, 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* to observe the class that accepts the changes as a *String* type.

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

Every auto-generated *aspect* is assigned a name of the form '$ObserverProtocol\_\#$', where the hash symbol refers to a sequence number that is 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. *Aspects* generated for *class-level* observing purposes have a different implementation style from the aspects used for *instance-level*.

- *Class-Level* Observing: As shown in Table 3.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) is executed.

Consequently, every instance of that observing class is notified (this task is accomplished by the advice declared in Lines 16-22). 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 3.B. This *aspect* has a single observer field (Line 3) that holds a reference copy of the observing instance that is assigned via the *addObserver* method, which is invoked at the client application (i.e., everywhere the '$let-observe-exec$' statement is provided 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 triggers such a notification) 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 is 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);
```

5. The Decorator Design Pattern

Decorator is a structural design pattern that allows performing additional actions on methods. It has several advantages over sub-classing (i.e. inheritance), since extra actions and objects can be added or removed per object at runtime. Moreover, adding more than one decoration action is easier than doing that in sub-classing. Sub-classing may lead to the subclasses explosion problem. Moreover, the decoration precedence is important, and it is difficult to manage the precedence issue using sub-classing. The Decorator design pattern provides flexibility in defining and maintaining decoration precedence.

5.1. *Syntax*

Similar to the Observer design pattern, we design a more abstract and modular construct using our extensible compiler to implement the Decorator design pattern. Our Decorator construct supports the use of fewer lines of code than AspectJ to implement different levels of decorating. We use the following EBNF notation to represent the syntax of our Decorator construct:

```
<DecorateWith>::= "decorate" <class_id> "." <method_declaration>
                  "with" <block> [ "to" <object_ids_list> ";" ]
```

Such a syntax allows programmers to decorate all objects of a certain class (i.e., *class-level* decorating) or specific objects of the class (i.e., *instance-level* decorating). In both cases, our Decorator construct decorates a specific method of the class (i.e., to decorate N methods, N of `DecorateWith` statements should be used).

5.2. *Application*

Listing 1 shows an example of how programmers can apply the Decorator construct. In this example, our `DecorateWith` construct is applied to two versions of the same `display` method of the `Screen` class. First we pass a single *String* parameter to the `display` method and then we pass two *String* parameters to it.

- **Class-level Decorating:** Applying the *class-level* decorating is shown in Lines 2-4 of Listing 1. The method `display` of the `Screen` class is decorated with a dollar decoration (i.e., $$$). This way of decoration allows programmers to decorate a single method. Programmers need to write the name of the class in addition to the method name. In this case, all instances of the specified class are decorated, which reduces any possible implementation overhead. Whenever the `display(String)` method is invoked by a `Screen` object (Lines 6 and 8), the associated decoration is applied. Hence, instead of printing the strings `'111'`, `'222'` and `'333'`, the program prints the decorated versions of these strings, which appear like `$$$111$$$`, `$$$222$$$` and `$$$333$$$`, respectively.
- **Instance-level Decorating:** *Instance-level* decorating has a similar structure to the *class-level* decorating. The only difference is that *Instance-level* decorating allows to specify which objects to decorate using our Decorator construct. The first application of our *instance-level* decorating is given in Lines 11-14 of Listing 1 by applying a star decorator to the *String* parameter of the `display` method. The target objects here are `screen1` and `screen2`. The second application is shown in Lines 16-19, which employs a bracket decoration and targets the `screen1` object only. Decorating per instance is not supported by the original AspectJ implementation of the Decorator design pattern introduced by Hannemann and Kiczales [3]. We resolve such a limitation in our proposed extension by allowing our constructs to store reference copies of the decorated objects in the auto-generated *aspect*.

The invocation of the `display` method using two *String* parameters is shown in Lines 22, 23, and 24 of Listing 1. The `display` method invoked by `screen1` is decorated with both the star and the bracket decorators. Hence, the `display` method prints the doubly-decorated version of the strings (i.e. `'[[[*** aaa bbb ***]]]'`). We discuss the decoration precedence later in this paper.

Listing 1. Application of Decorator Construct

```
1  // Class-level Dollar Decorator
2  decorate Screen.display(String s) with {
3     s = "$$$" + s +"$$$";
4  }
5  // invoke the decorated method
6  screen1.display("111"); // the decorated version
7  screen2.display("222"); // the decorated version
8  screen3.display("333"); // the decorated version
9  //-----------------------------------------------
10 // Instance-level Star Decorator
11 decorate Screen.display(String fn, String ln) with {
12    fn = " *** " + fn;
13    ln = ln +" *** ";
14 } to screen1, screen2;
15 // Instance-level Bracket Decorator
16 decorate Screen.display(String fn, String ln) with {
17    fn = " [[[ " + fn;
18    ln = ln +" ]]] ";
19    } to screen1;
20
21 // invoke the decorated method
22 screen1.display("aaa", "bbb"); // both [] and * decorators
23 screen2.display("ccc", "ddd"); // only * decorators
24 screen3.display("eee", "fff"); // un-decorated version
```

5.3. *Semantics*

After parsing '*decorate−with*' statements and matching them with the given syntax of the Decorator construct, the compiler then moves to 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 Decorator construct application by carrying out variable scoping, type-checking, and node translation.

5.3.1. *Scoping:*

Our extended compiler checks whether the decorated class (or object) is defined within the scope of the '*decorate − with*' statement. Also, our extended compiler checks whether the decorated method exists in the provided class and whether the method accepts the number and type of arguments specified in the Decorator construct. In addition, the body of the Decorator construct should have access to local variables only declared within which; i.e., global variables are not allowed to be used since they are inaccessible when code transformation happens. An empty body implies an empty decoration.

5.3.2. *Type checking:*

The compiler collects the types included in the Decorator construct and verifies their eligibility. The compiler first checks that the Decorator construct is applied to methods defined inside class types. Second, the type of instances specified in the Decorator construct should match the class type of the decorated method.

5.3.3. *Node Translation and Code transformation:*

Our extended compiler transforms the `DecorateWith` node into corresponding Java and AspectJ nodes based on the parameters provided in the Decorator construct. An *aspect* declaration node named 'DecoratorProtocol' is generated to implement the concrete decoration. Similar to the Observer design pattern, every '*decorate* − *with* − *to*' statement is transformed into a specialized *aspect* that contains the cross-cutting concerns of the Java program as shown in Table 4. All the generated decorator *aspects* are assigned the name 'DecoratorProtocol_#', where the hash symbol represents a sequence number assigned per each Decorator construct.

Table 4. Auto-Generated Aspect for Decorator Pattern

A. *Class-level* Decorating

```
1   protected privileged aspect DecoratorProtocol_1
2   {
3      protected pointcut decoratedMethod
4         (String s):
5            call(* (Screen).display(String)) && args(s);
6
7      void around (String s):
8         decoratedMethod(s) {
9              s = " $$$ " + s + " $$$ ";
10             this.proceed(s);
11        }
12  }
```

B. *Instance-level* Decorating

```
1   protected privileged aspect DecoratorProtocol_2
2   {
3      private List decs = new ArrayList();
4
5      public void addDecorator(Screen dec) {
6         decs.add(dec);
7      }
8      protected pointcut decoratedMethod
9         (String s, Screen dec):
10           call(* Screen.display(String)) && args(s) && target(dec);
11
12     void around(String s, Screen dec):
13        decoratedMethod(s, dec) {
14           if(decs.contains(dec)) {
15              s = " *** " + s + " *** ";
16        }
17        this.proceed(s, dec);
18     }
19  }
```

- *Class-Level* Decorating As shown in Table 4 (A), a single *aspect* is generated for each *class-level* Decorator construct. This *aspect* declares a generic pointcut (Lines 3-6) that captures the *joinpoints* of decorated methods when they are invoked by objects of the class specified in the Decorator construct. Once the decorated method is called, the `around` advice (Lines 8-12) decorates the corresponding arguments of the method, and then proceed to the method call instantly. After that, the `DecorateWith` nodes are replaced by empty statements (i.e. semicolons `';'`).

- *Instance-Level* Decorating As shown in Table 4 (B), a single *aspect* is generated for each *instance-level* Decorator construct. This *aspect* maintains a list (Line 3) that holds reference copies of the instances associated with the pattern construct. The decoration registration is achieved by the `addDecorator` method (Lines 5-7), which is invoked at the client application. Whenever the decorated method is invoked by an object of the decorated class, the `decoratedMethod` pointcut (Lines 8-12) is triggered and, subsequently, the *around* advice checks the existence of the calling object in the list (Lines 14-20). If the target object is considered for decoration, then the corresponding arguments of its method are decorated (Lines 16-18), and then proceed to calling the method call instantly (Line 19). Otherwise, the original (un-decorated) method is called. The `DecorateWith` nodes are replaced by method calling statements for each decorated instance, as follows:

```
DecoratorProtocol_2.aspectOf().addDecorator(screen1);
DecoratorProtocol_2.aspectOf().addDecorator(screen2);
```

Decoration Precedence In the Decorator design pattern, it is important to determine the precedence of the decorators. This means that the decorators should be applied in a certain order. In our approach, the decoration precedence depends on the declaration sequence of Decorator constructs. In other words, the firstly declared Decorator construct has the highest precedence and, hence, is applied first.

6. Results and Discussion

In this section, we highlight the characteristics of our proposed extensible compiler. We compare our approach to the AspectJ implementation of design patterns proposed by Hannemann and Kiczales [3]. Moreover, we show how we address the major issues of implementing design patterns introduced in the literature.

6.1. *Implementation issues of design patterns*

Implementation overhead: The implementation overhead of design patterns is a major issue in conventional approaches (e.g., using OO or AO constructs). Conventional approaches to implement design patterns require programmers to invest time and effort to implement design patterns in addition to the functional code. In our approach, we address this issue by proposing an extensible compiler that supports the implementation of design patterns using concise language constructs. Our approach automatically generates the concrete implementation of design patterns, which saves the time and effort required to implement them manually. In addition, programmers can reuse our design pattern constructs anywhere in the program without the need to worry about dependencies. Finally, using our approach, it is easy to keep track of the implementation of design patterns, since our design pattern constructs are recognizable units that can assist in design pattern detection [22].

Amount of written code: Our proposed extensible compiler allows programmers to write concise implementation code of design patterns than any of the state-of-the-art approaches. We show how many lines of code required for a simple implementation of design patterns in the following:

- For *Singleton* design pattern, our Singleton construct does not require any additional lines of code to declare a class as Singleton. A class only needs a *singleton* modifier to be Singleton. Using the conventional OO approach, programmers need to write at least five lines of code for a simple implementation of the Singleton design pattern. In addition, a single line of code is required to instantiate n instances of the Singleton class using our approach, while n lines of code are required when using the conventional approach.
- For the *Observer* design pattern, it is clear from the example presented in section 4 that only one line of code is required to implement the Observer design pattern, for both *class-level* and *instance-level*. In contrast, AspectJ requires 16 lines of code for the *class-level* implementation and eighteen for the *instance-level*.
- For the *Decorator* design pattern, the presented example in Listing 1 clearly shows that the effective number of lines of code is three for both *class-level* and *instance-level*. In contrast, AspectJ requires ten lines of code for the *class-level* implementation and 18 for the *instance-level* implementation.

Modularity: In our approach, modularity is witnessed by separating the concrete implementation of design patterns from the actual design pattern application. Concrete implementations of design patterns are not visible to programmers and are isolated from one application to another, which allows programmers to add, remove, and maintain their design pattern applications in modular and easy-to-manage units.

6.2. *Characteristics of our approach*

Hybrid: Our approach combines different characteristics of the state-of-the-art approaches. Our approach is introduced as an extensible AspectJ compiler, which makes it able to abstract plenty of code using concise constructs. This feature is inspired by meta-programming language [23]. In addition, our approach auto-generates *aspects* based on the information passed as parameters by our proposed constructs. This feature is adapted from the parametric *aspects* [24]. Although our approach is an extensible AspectJ compiler, it supports a *instance-level* advising, which is inspired from the *classpect* model introduced in *Eos* [25].

Expressive: The syntax of our design pattern constructs is clear, concise, and expressive. Our proposed constructs do not require programmers to import any other packages, create other classes (or *aspects*), or worry about missing a design concept of design patterns. Programmers simply need to understand the syntax of our design pattern constructs and how to use them. Moreover, the readability

and writeability are highly improved as the written code is shorter and more self-explanatory. Hence, the absence of dependencies makes it very easy to maintain and reuse the source code.

Multi-level application: Our approach supports different levels of design pattern application. Our constructs help programmers to decide where and how to apply a certain design pattern. The *class-level* application is useful in the cases where all objects of a class need to employ the design pattern functionality (e.g., observing or decorating). On the other hand, the *instance-level* application is useful when only a subset of class objects need the design pattern functionality or when each object needs to have its own application of the design pattern.

6.3. *Complex design patterns*

Extending our compiler to implement more complex design patterns, such as *Visitor* might require more complicated factory nodes and compiler passes. In addition, for some design patterns, such as *Facade*, extending our compiler could be infeasible, since the eventual design pattern contracts would require almost the same amount of effort and lines of code to apply a certain design pattern. Hence, we encourage developers to assess the feasibility of extending our extensible compiler for such kinds of design patterns.

6.4. *Practical Implications*

Our approach would attract software developers to use our proposed design pattern constructs in their applications and to extend our extensible compiler further to support the implementation of other design patterns. Using our approach helps programmers maintain their code by keeping track of the design pattern constructs. In addition, our approach reduces fault and defect proneness, since programmers do not touch the back-end concrete implementation of the design patterns. Moreover, considering that our approach encourages the use of explicit language constructs, researchers can find it easy to locate design pattern instances, especially when dealing with large, undocumented software systems.

7. Related Work

Hannemann and Kiczales [3] used AO constructs to improve the implementations of the original 23 design patterns using AspectJ. They analyzed and evaluated the improvement achieved to the implementation of the patterns according to different metrics, which also have been addressed later by Rajan [12] using Eos extended by 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 seven 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. Sousa and Monteiro [26] proposed CaesarJ, which supports the family polymorphism. CaesarJ 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 a positive influence of the *collaboration* interface on modularity, generality, and reusability over those with AspectJ. Gomes and Monteiro [27] introduced the implementation of five design patterns in *Object Teams* compared with that in Java and AspectJ. Regardless of *Object Teams* goals, it showed powerful support in implementing design patterns efficiently, and with more than one alternative. The entire transformation 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, since the concrete implementation of design patterns is required to be manually written, or at least understood and followed, by programmers, which may reduce their productivity.

Another approach was introduced by Zook et al. [23]. 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 [4], who provided a new object model called *LayOM*. This model supports explicitly representing design patterns 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 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 [28] 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 class inheritance and specified by attribute declarations. Doing so enables the extended compiler to automatically check the application of patterns against the specified rules. However, the creation of rules, roles, and attributes makes syntax more complex and lacks expressiveness and requires extensive effort to learn and build them.

Lovatt et al. [29] proposed an extensible Java compiler called *PEC*. Design patterns in *PEC* are provided as *marker* interfaces. A class must implement a ready-made interface in order to apply a certain design pattern. Then, *PEC* checks the structure and behavior of such used design pattern. *PEC* allows programmers to assign their desired design pattern to a given class and then implement such design

pattern manually. However, *PEC* does not reduce the effort needed to implement design patterns (i.e., it suffers from implementation overhead).

Budinsky et al. [30] 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++) is generated automatically. This approach allows programmers to customize design patterns as needed. However, the modularity and reusability in their proposed approach are missed. Their approach also suffers from a traceability problem.

Aljasser [24] proposed an approach that uses parametric aspects (i.e., ParaAJ) to facilitate the implementation of three design patterns, namely Singleton, Observer, and Decorator. However, ParaAJ does not support different scenarios of design pattern implementations and fails to implement the Decorator design pattern. The author suggests that introducing programming language constructs to implement design patterns is necessary to simplify the development of recurring design patterns.

Batdalov and Nikiforova et al. [31] studied the features of programming languages and how such features can be used to reduce the complex implementations of design patterns. In addition, the authors proposed supplementary features to programming languages that may support better implementations of design patterns. However, their proposed features are orthogonal to existing OO features, which may require programmers to participate in writing the design pattern implementation code. Their study suggests that programming languages are still unable to make the implementation of design patterns easier. They also suggest that programming languages need to be extended to better support easier implementations of design patterns.

Springer et al. [32] proposed a context-oriented programming approach to implement three design patterns, namely Observer, Decorator, and Visitor. Their approach can deal with classes as layers, where partial methods of other classes can be added. Their approach supports multiple application levels of design patterns and resolves some conventional issues, such as object interactions and loss of object identity. Seidl et al. [33] proposed variability-aware design patterns that are automatically generated in software product line development. Despite the research invested to support the implementation of design patterns, previous approaches are not applied as recognizable units. Programmers still need to be aware of the design pattern implementation code to place their intended actions inside the proposed classes.

Gatrell et al. [6] reported that classes that implement design patterns are more fault-prone than normal classes. Sousa et al. [7] investigated the association between the use of design patterns and code smells. The authors found that conventional implementation of design patterns is associated with bad code smells. Similarly, Alkhaeir and Walter [8] found that the presence of code smells in design pattern classes increases the defect-proneness of source code. In addition, Onarcan et al. [9]

found that instances of design patterns have a high correlation with the priority of software defects. Furthermore, prior research has shown that manually implementing design patterns can affect program understandability and makes it challenging to spot the code portions in which a design pattern is used [10, 11, 34].

Therefore, it is important for programmers to avoid the manual implementation of design patterns. In this paper, we propose an extensible compiler that allows programmers to apply design patterns in their code using concise language constructs. Our approach allows language constructs to be used in a modular, easy-to-understand syntax. Our approach is based on source-to-source transformation of code [35, 36], in which our proposed design pattern constructs (i.e., meta-program representation [37]) are transformed into AspectJ code.

8. Conclusion

In this paper, we have proposed an extensible Java compiler to support the implementation of design patterns using concise, expressive language constructs. We have developed our extensible compiler using the *abc* extensible compiler. We have applied our approach to three commonly used design patterns, namely Singleton (a creational pattern), Observer (a behavioral pattern), and Decorator (a structural pattern). Our proposed approach addresses existing issues regarding design pattern implementation introduced by conventional approaches. For example, using conventional approaches, programmers need to maintain the classes and interfaces of a certain design pattern. In our approach, design pattern constructs are designed to have an expressive syntax, in which programmers can apply certain design patterns without caring about the concrete implementation of the design patterns. In addition, our approach provides programmers with two levels of application of design patterns: class-level and instance-level. We conclude that our approach would attract software developers to (i) use our proposed design pattern constructs in their applications and (ii) extend our extensible compiler further to support the implementation of other design patterns.

Future work. We aim in the future to improve our approach to (a) support further scenarios in which programmers can control the way design patterns are used and (b) optimize the code transformation process. We also aim to perform controlled experiments to empirically evaluate the productivity of programmers using our approach and other conventional or recent approaches.

References

[1] W. Pree and E. Gamma, *Design patterns for object-oriented software development* (Addison-wesley Reading, MA, 1995).

[2] E. Gamma, *Design patterns: elements of reusable object-oriented software* (Pearson Education India, 1995).

[3] J. Hannemann and G. Kiczales, Design pattern implementation in Java and AspectJ, in *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, **37**, (Seattle, USA, 2002), pp. 161–173.

[4] J. Bosch, Design patterns as language constructs, *Journal of Object-Oriented Programming* **11**(2) (1998) 18–32.
[5] B. B. Mayvan, A. Rasoolzadegan and Z. G. Yazdi, The state of the art on design patterns: A systematic mapping of the literature, *Journal of Systems and Software* **125** (2017) 93–118.
[6] M. Gatrell and S. Counsell, Design patterns and fault-proneness a study of commercial c# software, in *2011 Fifth International Conference on Research Challenges in Information Science*, IEEE, (Gosier, France, 2011), pp. 1–8.
[7] B. L. Sousa, M. A. Bigonha and K. A. Ferreira, An exploratory study on cooccurrence of design patterns and bad smells using software metrics, *Software: Practice and Experience* **49**(7) (2019) 1079–1113.
[8] T. Alkhaeir and B. Walter, The effect of code smells on the relationship between design patterns and defects, *IEEE Access* **9** (2020).
[9] M. O. Onarcan, Y. Fu *et al.*, A case study on design patterns and software defects in open source software, *Journal of Software Engineering and Applications* **11**(05) (2018) p. 249.
[10] A. D. Lucia, V. Deufemia, C. Gravino and M. Risi, Detecting the behavior of design patterns through model checking and dynamic analysis, *ACM Transactions on Software Engineering and Methodology (TOSEM)* **26**(4) (2018) 1–41.
[11] H. Yarahmadi and S. M. H. Hasheminejad, Design pattern detection approaches: a systematic review of the literature, *Artificial Intelligence Review* **53**(8) (2020) 5789–5846.
[12] H. Rajan, Design pattern implementations in Eos, in *Proceedings of the 14th Conference on Pattern Languages of Programs*, ACM, (Illinois, USA, 2007), pp. 9:1–9:11.
[13] T. A. Ghaleb, K. A. Aljasser and M. A. Alturki, Methods, computer readable media, and systems for compiling concise expressive design pattern source code (October 8 2019), US Patent 10,437,572.
[14] T. A. Ghaleb, K. A. Aljasser and M. A. Alturki, Method for compiling concise source code (January 26 2021), US Patent 10,901,711.
[15] T. A. Ghaleb, K. A. Aljasser and M. A. Alturki, Source code compiler system (January 26 2021), US Patent 10,901,712.
[16] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. De Moor, D. Sereni, G. Sittampalam and J. Tibble, abc: An extensible AspectJ compiler, in *International Conference on Aspect-Oriented Software Development, AOSD 2005*, (Chicago, USA, 2005), pp. 87–98.
[17] N. Nystrom, M. R. Clarkson and A. C. Myers, Polyglot: An extensible compiler framework for java, in *International Conference on Compiler Construction*, Springer, (Warsaw, Poland, 2003), pp. 138–152.
[18] C. J. H. J. Dick Grune, *Parsing Techniques: A Practical Guide* ("Springer-Verlag New York Inc.", 1999).
[19] T. A. Ghaleb, K. Aljasser and M. A. AlTurki, Implementing the Observer Design Pattern as an Expressive Language Construct, in *The Tenth International Conference on Software Engineering Advances*, ThinkMind, (Barcelona, Spain, 2015), pp. 463–469.
[20] A. Mehmood and D. N. Jawawi, Aspect-oriented model-driven code generation: A systematic mapping study, *Information and Software Technology* **55**(2) (2013) 395–411.
[21] N. Cacho, C. Sant'anna, E. Figueiredo, F. Dantas, A. Garcia and T. Batista, Blending design patterns with aspects: A quantitative study, *Journal of Systems and Software* **98** (2014) 117–139.

[22] D. Heuzeroth, T. Holl, G. Hogstrom and W. Lowe, Automatic design pattern detection, in *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, IEEE, (Portland, USA, 2003), pp. 94–103.
[23] D. Zook, S. S. Huang and Y. Smaragdakis, Generating AspectJ programs with meta-AspectJ, in *International Conference on Generative Programming and Component Engineering*, Springer, (Vancouver, Canada, 2004), pp. 1–18.
[24] K. Aljasser, Implementing design patterns as parametric aspects using ParaAJ: the case of the singleton, observer, and decorator design patterns, *Computer Languages, Systems & Structures* **45** (2016) 1–15.
[25] H. Rajan and K. Sullivan, Classpects in practice: A test of the Unified Aspect Model, tech. rep., Citeseer (2005).
[26] 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, (Brussels, Belgium, 2008), pp. 6:1–6:6.
[27] M. P. Monteiro and J. Gomes, Implementing design patterns in Object Teams, *Software: Practice and Experience* **43**(12) (2013) 1519–1551.
[28] G. Hedin, Language support for design patterns using attribute extension, in *European Conference on Object-Oriented Programming*, Springer, (Jyväskylä, Finland, 1997), pp. 137–140.
[29] 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., (Newcastle, Australia, 2005), pp. 69–78.
[30] F. J. Budinsky, M. A. Finnie, J. M. Vlissides and P. S. Yu, Automatic code generation from design patterns, *IBM Systems Journal* **35**(2) (1996) 151–171.
[31] R. Batdalov and O. Nikiforova, Towards easier implementation of design patterns, in *Proceedings of the 11th International Conference on Software Engineering Advances*, (Rome, Italy, 2016), pp. 123–128.
[32] M. Springer, H. Masuhara and R. Hirschfeld, Classes as Layers: Rewriting Design Patterns with COP: Alternative Implementations of Decorator, Observer, and Visitor, in *Proceedings of the 8th International Workshop on Context-Oriented Programming*, ACM, (Rome, Italy, 2016), pp. 21–26.
[33] C. Seidl, S. Schuster and I. Schaefer, Generative software product line development using variability-aware design patterns, *Computer Languages, Systems & Structures* **48** (2017) 89–111.
[34] M. G. Al-Obeidallah, M. Petridis and S. Kapetanakis, A structural rule-based approach for design patterns recovery, in *International Conference on Software Engineering Research, Management and Applications*, Springer, (London, UK, 2017), pp. 107–124.
[35] F. Amato and F. Moscato, Model transformations of mapreduce design patterns for automatic development and verification, *Journal of Parallel and Distributed Computing* **110** (2017) 52–59.
[36] K. Lano, S. Kolahdouz-Rahimi, S. Yassipour-Tehrani and M. Sharbaf, A survey of model transformation design patterns in practice, *Journal of Systems and Software* **140** (2018) 48–73.
[37] V. Štuikys and R. Damaševičius, Meta-programming task specification using feature-based patterns and domain program scenarios, in *Meta-Programming and Model-Driven Meta-Program Development*, (Springer, 2013) pp. 171–188.