Java Extensions for Design Pattern Instantiation

André L. Santos and Duarte Coelho

Instituto Universitário de Lisboa (ISCTE–IUL), ISTAR
Av. Das Forças Armadas, Edifício II ISCTE, 1649-026 Lisboa, PORTUGAL
andre.santos@iscte.pt
duarte.goncalo.coelho@gmail.com

Abstract. Design patterns are not easily traceable in source code, leading to maintainability and comprehension issues, while the instantiation of certain patterns involves generalizable boiler-plate code. We provide high-level language constructs addressing design patterns that transform source code by injecting a substantial part of their implementation at compile time. We developed proof of concept extensions addressing widely used design patterns, namely Singleton, Visitor, Decorator, and Observer, using annotations as the means to extend Java. We describe our Java annotations to support these design patterns and the associated source code transformations, demonstrating that it is possible to significantly reduce the necessary code to instantiate a pattern through the use of high-level constructs.

1 Introduction

Design patterns are widely used in software development and became an essential element in software reuse. Design patterns are language-independent, but paradigm-dependent, since they rely on certain programming language constructs that often are available in certain programming paradigms only. The focus of our work is on object-oriented design patterns [8].

Design patterns are used to aid the design of systems, often driven by variability and extensibility requirements. Issues related to maintainability and evolution may occur given that pattern instantiations are interleaved with the system domain and "fade away" into the source code [15]. This implies that pattern instantiations are hard to trace, mainly because they have no first-class representation in the source code in terms of programming constructs. Further, the presence of design patterns in source code may hinder understandability [13]. Certain patterns require substantial boiler-plate code to be written, as for instance, the abstract decorator class role in the Decorator pattern [8] that delegates all the calls to an enclosing reference (highly generalizable code).

Apart from a few exceptions, languages do not have dedicated constructs for representing design patterns in the source code. As a counter-example, the Iterator pattern [8] is supported by the libraries of mainstream object-oriented languages, such as Java and C#. On the other hand, some patterns do not make sense in the context of certain programming languages, simply because the

language constructs provide the means for solving the problem directly (e.g., the Visitor pattern [8] is not relevant in a language with multiple method dispatch).

Some approaches assist developers in the instantiation of patterns either through external programming languages (e.g., [2, 10, 1]) or IDE-integrated tools to guide and automate the process of implementing them through code generation (e.g., [9]). The former require using other programming paradigms, whereas the latter do not address pattern representation given that the trace is lost after the pattern is instantiated in the source code. The fact that there are tools capable of generating code that instantiates design patterns evidences that patterns are generalizable into higher-level abstractions, including dedicated language constructs (see a debate in [6]).

In this paper we describe an approach for generalizing design pattern instantiations for Java, providing high-level programming language constructs for instantiating them using annotations. As a proof of concept, we implemented support for widely used patterns, such as Singleton, Visitor, Decorator, and Observer, relying on an existing open-source project called Lombok[1]. This project provides the infrastructure for extending Java with generative annotations that perform compile-time AST transformations to inject class members and statements. Lombok provides extensions that enable developers to write code in a terse manner, as for instance support for getter and setter method injections, as well as some design patterns such as Value Object [7] and Builder [8]. We build on the Lombok infrastructure to address other design patterns that were not previously addressed, developing an extension that we refer to as JEDI[2].

Although the idea of having language constructs for design patterns is not new (e.g., [2, 6]), we are not aware of other approaches that address this problem relying only on object-orientation in Java, that is, with no resort to additional programming paradigms or external tools. We demonstrate the feasibility of the approach, showing that significant amounts of pattern-related code can be generated from simple declarations embodied in the form of annotations. These have the advantage of being dedicated language constructs that are traceable, while simultaneously serving the purpose of documentation. Empirical experiments have shown that documenting patterns in the source code is beneficial for system maintenance [14]. Therefore, besides facilitating pattern instantiation, the annotations also mitigate traceability and maintainability issues.

This paper proceeds as follows. Section 2 introduces a running example that is used throughout the paper. Section 3 presents project Lombok and briefly explains its infrastructure for transforming classes. Section 4 describes the Java extensions that we developed to support design patterns. Section 5 analyzes the transformations performed by our extensions. Section 6 discusses the benefits and limitations of our approach. Section 7 discusses related work, and Section 8 presents our conclusions.

---

[1] www.projectlombok.org

[2] Java Extensions for Design pattern Instantiation. Available at `github.com/andre-santos-pt/lombok-jedi`

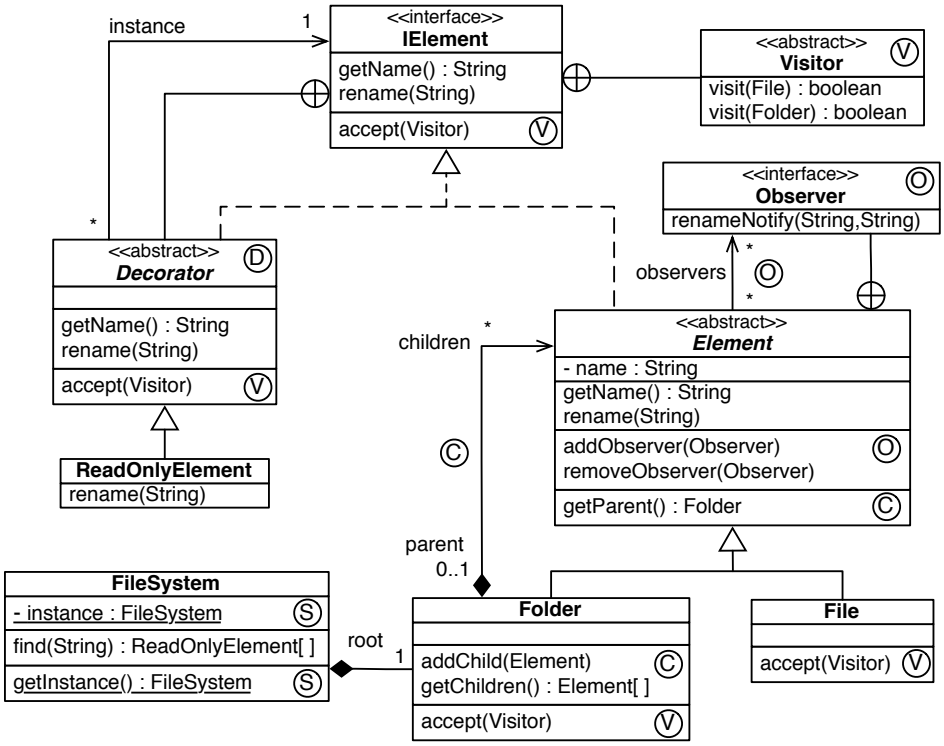

Fig. 1. UML class diagram describing the running example: a file system with files and folders. The operation compartments are divided according to the associated design pattern and the letter labels identify the pattern to which the types or members relate to (Singleton, Composite, Visitor, Decorator, Observer). Notation: $a\oplus \rightarrow b$ denotes that b is a nested classifier of a (in programming these are mapped to inner classes/interfaces).

2 Running example

In order to illustrate our approach we introduce a small running example involving several design patterns (see Figure 1), designed intentionally to be simple for clarity of presentation, and on the other hand, appropriate to demonstrate all of our Java extensions. Section 4 describes how our annotations are able to address the instantiation of each pattern, except for Composite, which we omit due to space constraints.

The example consists of a `FileSystem` that structures its `Elements` in a tree. The `FileSystem` class can only have a single instance (Singleton pattern), and holds a reference to the root `Folder`. The singleton property is ensured by having a static field `instance` that holds the unique instance, which can be obtained through the static operation `getInstance()` (there are no public constructors).

The class `Folder` is an `Element` that can have `Files` (leafs) and other `Folders` as children (Composite pattern). The methods for adding children and obtaining an `Element`'s parent relate to the instantiation of this pattern.

The interface `IElement` is yet a more abstract representation of `Element` objects. The `FileSystem` tree is traversable to iterate over its `Files` and `Folders` (Visitor pattern), by providing a specialization of the abstract class `IElement.Visitor`. The instantiation of this pattern requires the `accept` method to be defined by every visitable node (`File` and `Folder`).

Elements may be wrapped in read-only views that disallow renaming (Decorator pattern) using the class `ReadOnlyElement`. The instantiation of this pattern, since there may be other kinds of decorator objects, involves the abstract class `IElement.Decorator`, which implements `IElement` and holds a reference to the decorated instance, delegating every call to it. Notice that the `accept` method pertaining to the Visitor pattern also had to be included here for interface compatibility. As an example of a concrete decorator, the class `ReadOnlyElement` overrides the `rename(...)` operation to throw a runtime exception (disallowed operation).

Finally, the `Element` objects are observable with respect to rename events (Observer pattern) through the registration of `Element.Observer` objects. The methods for adding and removing observers pertain to this pattern, as well has the association `observers`.

Notice that in this example the elements pertaining to the essence of the domain that the model is capturing (i.e. the file system structure) are clearly outnumbered by infrastructural elements that are necessary to implement the desired functionally and extensibility properties. The Composite pattern is the only pattern whose elements inherently pertain to the domain. This means that the remaining patterns "bloat" the design with several elements that are essentially technical (*accidents* in software engineering [3]).

3 Project Lombok and AST Transformations

Lombok is an open source project whose main aim consists of reducing the amount of boiler-plate code that writing Java programs often involves. The goal is achieved through annotations that work as language extensions. At compile time, Lombok annotation processors interfere with the AST of the classes where annotations are present in order to perform transformations, such as introducing members (fields, methods, types) or modifying existing ones. The transformed ASTs are in turn compiled normally. Lombok inspired our work and served as the infrastructure for the realization of our language extensions.

Figure 2 illustrates two of the simplest Lombok annotations. The annotation `@Getter` has the purpose of injecting getter methods based on attributes, whereas the `@NonNull` injects null pointer validations on parameters. Hereinafter, when presenting examples of transformations, we include a box with the code that the programmer writes followed by another shadowed box that contains the code that actually compiles after the AST transformation, highlighting the injected code with gray color. Note that the programmer does not manipulate the source

```java
public class Element {
  @Getter
  private String name;

  public void rename(@NonNull String name) {
    this.name = name;
  }
}
```

```java
public class Element {
  @Getter
  private String name;

  public String getName() {
    return name;
  }

  public void rename(@NonNull String name) {
    if(name == null)
      throw new NullPointerException("...");
    this.name = name;
  }
}
```

Fig. 2. Example of Lombok extensions: Getter method injection (`@Getter`) and null pointer validation (`@NonNull`).

code of the transformed version of the classes. The injected members cannot be edited and are not even visible to the programmer. However, the injected members are accessible to other classes at compilation time, and hence, one may write code that uses them as if they have been manually written.

One of the key advantages of this approach is that the annotated classes become significantly less bloated, with fewer lines of code. Furthermore, annotations capture programmer intent with a dedicated construct (the annotation). The given example is rather simple, and hence, the amount of injected code is not impressive. However, in other cases such as the annotation for addressing the Value Object pattern [7], Lombok transforms classes so that the number of injected lines of code outnumbers manually written code by a factor greater than five for classes with a few attributes.

As portrayed by the Lombok authors, the technical solution may be regarded as a "hack", since Java annotations were not meant to affect program semantics. However, there are other approaches that rely on annotations as the means to mark parts of programs that are transformed by a third-party. For instance, transformations to enhance the class with concurrency control (e.g., [5]) or to perform runtime verifications (e.g., [11]). Lombok was designed for extensibility, enabling third-party developers to contribute with additional annotations and their associated AST transformations. We have used this extensibility mechanism to implement our Java extensions.

4 Java Extensions

We developed JEDI, a proof of concept implementation of Java extensions for design patterns. JEDI comprises a set of annotations whose names (including participant names) resemble the ones described in [8]. So far, we successfully addressed the patterns Singleton, Composite, Visitor, Decorator, and Observer. In this paper, we omit the description of Composite due to space constraints. The purpose of our annotations is not to fully automate the instantiation of design patterns, but instead to aid in their instantiation by providing constructs for their generalizable aspects. For each provided annotation we developed a Lombok handler that transforms the annotated classes. The transformation may involve injection or modification of fields, methods, or inner types (classes or interfaces).

The following subsections describe in detail how JEDI annotations can be used and the code transformations that are performed when applying them, using the example of Section 2. In some situations the injected code makes use of the `@NonNull` annotation (illustrated in Section 3) that would further trigger additional transformations, but whose result we do not expand for clarity and brevity of presentation.

4.1 Pattern Instantiation Properties

Before illustrating our annotations and the associated AST transformations, this section explains certain properties of the pattern instantiations that are important with respect to the use of the annotations in software development.

Validations. The annotation processors perform validations to ensure that the annotations are applied correctly and guarantee the correctness of the injected code. Without the validations the annotation placements would be fragile, since the annotations would not feel like language extensions if errors are not emitted when they are used incorrectly. For instance, given that the Singleton pattern requires that the class has no public constructors, there is a validation for checking this issue that emits a compile-time error in case of violation.

Priorities. Each annotation handler has a fixed priority that determines the order in which the transformations pertaining to the different annotations are performed on the types. This aspect is relevant since some patterns inject elements that are of interest to other patterns, and hence, have an effect on the associated transformation. For instance, the Decorator pattern has a higher priority than the Visitor pattern, given that it requires the decorated interface to be complete (Visitor adds operations to interfaces), so that the transformation addresses all of its methods.

Identifiers. All the identifiers of injected elements (fields, methods, or types) have default values that are either constant of inferred from other related elements. However, since design patterns are abstract solutions that are made concrete in a variety of situations, JEDI annotations were designed to offer a reasonable degree of adaptability allowing programmers to override the values for identifiers through annotation parameters. For instance, the default name for the operation for registering observers in the Observer pattern is by default

"addObserver", but this name can be set to other value. Throughout the paper, all the examples of annotation usage consider default values for identifiers.

Bidirectional traceability. The elements that are injected in the AST are themselves annotated with an annotation for bidirectional traceability purposes, so that every injected element can unambiguously be traced back to the annotation pattern that generated it. For clarity and brevity of presentation we do not include these annotations in the transformed code of the given examples.

4.2 Singleton Pattern

The singleton pattern is a solution that guarantees that there is a single instance of a given class at runtime [8]. The pattern is typically applied by storing the unique instance in a static field of the class that is accessed through a static method (that performs lazy instantiation), while the class has no public constructors available. In the example given in Section 2, the `FileSystem` class illustrates the Singleton pattern. The static field `instance` stores the unique instance, which is accessed through the static method `getInstance()`.

We provide the `@Singleton` annotation to aid on implementing the Singleton pattern (see Figure 3). This annotation can only be used on classes, implying the injection of the following elements: (a) a static field to store the singleton instance with the same type as the class, (b) an empty private constructor to override the default public parameterless constructor if none is defined, and (c) a static method to retrieve the singleton instance (initializing (a) on the first call using the parameterless constructor). The annotation validation ensures that the class has no public constructors.

```
@Singleton
public class FileSystem {
  ...
}
```

```
@Singleton
public class FileSystem {
  private static FileSystem instance; // (a)

  private FileSystem() { // (b)
  }

  public static synchronized FileSystem getInstance() { // (c)
    if(instance == null) {
      instance = new FileSystem();
    }
    return instance;
  }
  ...
}
```

Fig. 3. Singleton pattern support and transformations (`@Singleton`).

4.3 Visitor Pattern

The Visitor pattern is a solution to separate operations from an object structure [8]. The pattern instantiation is achieved by defining an abstract class, whose compatible objects are referred to as *visitors*. This class contains methods, often named `visit` and typically overloaded, that receive multiple object types (the visitable nodes) to which the nodes provide their instance. In the example of Section 2, the file system elements take the role of visitable nodes (`Folder` and `File`), whereas the abstract class `Visitor` has the visitor role.

We provide three related annotations to address the visitor pattern (see Figure 4). The `@Visitor` annotation is used to mark an interface that represents the set of visitable nodes. It injects an inner abstract class (a), that contains a method `visit(...)` returning true for each of the visitable node types (b), which are marked with the annotation `@Visitor.Node`. The annotation validation ensures that these types are compatible with a type annotated with `@Visitor`. By injecting each `visit(...)` method, we solve the problem of having to define manually each operation, which is one of the visitor's implementation negative consequences [8]. Additionally, an `accept(...)` operation declaration is injected into the interface (c) with a parameter of type equal to (a).

Visitable nodes may have child visitable nodes. The annotation `@Visitor.Children` is used to mark the visitor node fields that store the children nodes of the current node, so that the visitor traversal can be propagated to them. The annotation validation ensures that type of the visitor children fields must be either of a visitor node or of a collection of visitor nodes (compatible with `java.util.Collection`). On each visitable node type an `accept(...)` method is injected whose body contains a call to the `visit(...)` operation (d). In case a visitor node has children, the method body also includes a loop for invoking the `accept(...)` operation on each child (e).

4.4 Decorator Pattern

The decorator pattern [8] is an alternative solution to inheritance comprising an abstract class that represents *decorator* objects (that conform to a given interface), containing a reference to an object to which all the interface calls are delegated. In the example given in Section 2, the class `IElement.Decorator` represents decorators of `IElement` objects.

Figure 5 demonstrates the application of our `@Decorator` annotation on the `IElement` interface. The annotation validation ensures that it can only be used on interfaces. The annotation injects an abstract class representing the abstract decorator that implements the annotated interface (a), composed of: (b) an instance field for storing a reference to the decorated object, (c) a public constructor that receives the reference to the decorated object, and (d) an implementation of every method of the interface where the calls are delegated to the decorated object. By generating all the delegating calls, we significantly reduce the lines of code that otherwise would have to be written and maintained manually. Notice that in this case the injection is performed after the Visitor injections (priority issue

```
@Visitor
public interface IElement {
  ...
}
```

```
@Visitor
public interface IElement {
  ...
  // (a)
  abstract class Visitor {
    public boolean visit(File node) { // (b)
      return true;
    }

    public boolean visit(Folder node) { // (b)
      return true;
    }
  }

  void accept(Visitor visitor); // (c)
}
```

```
@Visitor.Node
public class File extends Element {
  ...
}
```

```
@Visitor.Node
public class File extends Element {
  ...
  public void accept(@NonNull Visitor visitor) { // (d)
    visitor.visit(this);
  }
}
```

```
@Visitor.Node
public class Folder extends Element {
  ...
  @Visitor.Children
  @Composite.Children
  private List<Element> children;
}
```

```
@Visitor.Node
public class Folder extends Element {
  ...
  @Visitor.Children
  private List<Element> children;  // (e)

  public void accept(@NonNull Visitor visitor) { // (d)
    if(visitor.visit(this)) {
      for(Element child : children) { // (e)
        child.accept(visitor);
      }
    }
  }
}
```

Fig. 4. Visitor pattern support and transformations (`@Visitor`, `@Visitor.Node`, `@Visitor.Children`).

```java
@Decorator
@Visitor
public interface IElement {
  String getName();
  void rename(String name);
}
```

```java
@Decorator
@Visitor
public interface IElement {
  String getName();
  void rename(String name);
  void accept(Visitor visitor);

  // (a)
  abstract class Decorator implements IElement {
    private final IElement instance;  // (b)

    public Decorator(@NonNull IElement instance) {  // (c)
      this.instance = instance;
    }

    public String getName() { // (d)
      return instance.getName();
    }

    public void rename(String name) { // (d)
      instance.rename(name);
    }

    public void accept(@NonNull Visitor visitor) { // (d)
      instance.accept(visitor);
    }
  }
}
```

Fig. 5. Decorator pattern support and transformations (`@Decorator`). This example evolves the Visitor example presented in Figure 4, demonstrating the effect of annotation processing priority. Given that Visitors precede decorators, the injected Decorator takes into account the previously injected `accept` method (dashed line).

explained previously), and hence, the `accept(Visitor)` operation is considered in the abstract decorator class.

We also provide the `@Wrapper` annotation that is a variant with a slightly different purpose than the decorator pattern. This annotation follows a more flexible approach regarding method delegation. Instead of generating an abstract class, we can directly annotate the class that wraps the decorated object. This alternative requires the class whose objects we want to decorate to be defined in an annotation parameter (e.g., `@Wrapper(classType=Collection.class)`). The annotation injects a delegating method for each public method of the target class that is not manually defined.

4.5 Observer Pattern

The Observer pattern [8] is an effective way for objects (subjects) to communicate events of interest to other objects (observers) without depending directly on their classes. In the example of Section 2, the method `rename(String)` from the class `Element` (subject) illustrates an observable event notified through observer objects that are compatible with the `Element.Observer` interface.

Figure 6 illustrates the annotations for the Observer pattern on the method `rename(String)` of the class `Element` of the running example. We provide the `@Observable` and `@Observable.Notify` annotations to aid on the instantiation. The former is used to annotate methods whose execution represents an event of interest that we want to enable observer objects to be notified of. The latter is used to mark the variables that hold the objects that we wish to include in the notification. We only support the implementation pertaining to the subject par-

```
public abstract class Element implements IElement {
  ...
  @Observable
  public void rename(@Observable.Notify String name) {
    @Observable.Notify
    String oldName = name;
    this.name = name;
  }
}
```

```
public abstract class Element implements IElement {
  ...
  // (a)
  public interface Observer {
    void renameNotify(String name, String oldName);
  }

  // (b)
  private final List<Observer> observers = new ArrayList<>();

  public void addObserver(@NonNull Observer o) { // (c)
    observers.add(o);
  }

  public void removeObserver(@NonNull Observer o) { // (c)
    observers.remove(o);
  }

  @Observable                              // (d)
  public void rename(@Observable.Notify final String name) {
    @Observable.Notify
    final String oldName = this.name; // (d)
    this.name = name;
    for(Observer observer : observers) {  // (e)
      observer.renameNotify(name, oldName);
    }
  }
}
```

Fig. 6. Observer pattern support and transformations (`@Observable` and `@Observable.Notify`).

ticipant, given that the aspects related to observer objects are problem-specific and are not suitable for being generalized.

The purpose of the `@Observable` annotation is to create the elements for collaboration between the subject's event types and its observers, by generating the following elements in the subject class: (a) an inner interface representing the observable event, (b) a field that stores a collection of objects of type (a) to which the event notification is sent, and (c) methods to subscribe and unsubscribe the notification of the event. The structure of the injected interface is derived from the annotated elements. Each observable event has a corresponding operation in the interface, whose parameters are determined by variables annotated with `@Observable.Notify` (either parameters or local variable declarations). Each of these variables is augmented with the final modifier in order to guarantee their immutability (d). Finally, the body of the methods annotated with `@Observable` is augmented with the event notification to its subscribers (e).

We offer the possibility of using an existing interface, rather than having a newly injected one. If an inner interface already exists with the same name, such an interface is considered instead. The parameters of the `@Observable` annotation allow programmers to further customize the implementation of the observer pattern, namely with respect to point of notification (beginning or end of the method), interface to be used (existing or injected), and association of interface operations to events.

5 Analysis

In this section we analyze our running example with a focus on the amount of injected lines of code (LOC), and the relation between each Java extension and the transformed code. Table 1 presents the classes of the running example that were used as illustration throughout Section 4, in terms of manually written LOC, and LOC that were effectively compiled considering the transformations (manual plus injected code). The amount of injected LOC is decomposed, discriminating the LOC according to the design pattern they pertain to. Recall that the injected code resembles what otherwise would be written by hand when not using our extensions. Looking back to Figure 1, notice that every element in the

	Manual	Injected	Compiled	Singleton	Composite	Visitor	Decorator	Observer
`FileSystem`	17	**10**	27 (159%)	10	-	-	-	-
`Element`	26	**22**	48 (185%)	-	8	1	-	13
`Folder`	14	**17**	31 (221%)	-	10	7	-	-
`File`	7	**7**	14 (200%)	-	4	3	-	-
`IElement`	7	**24**	31 (443%)	-	-	9	15	-
Total	71	**80**	151 (213%)	10	22	20	15	13

Table 1. Overview of the number of lines of code in the running example classes, discriminating between manually written and injected code, decomposing the latter according to the related pattern.

diagram labeled with a letter was obtained through a transformation. Although we omitted the description of our support for the Composite pattern, here we include the result of applying it in the running example.

The efective number of LOC that define the classes is significantly higher than the manual code, more than twice in this example. This factor is by no means generalizable, given that the domain elements of the example were minimal, and hence, the weight of the injected code is high. Some of the transformations perform an injection whose size in terms of LOC is constant despite the elements where the annotations are applied, whereas the injected code of other transformations grows linearly with the size of the annotated elements. The latter are more powerful because they spare more effort when writing code, facilitate maintenance, and reduce the size of files significantly. The former are not as beneficial with this respect, but nevertheless, share the advantage of having a dedicated language construct that consists of an unambiguous representation of the pattern (traceability), which is guaranteed to be instantiated uniformly.

The Singleton extension is an example of a constant transformation, given that no matter how large is the annotated class, the injected code always has the same size. The value of 10 injected LOC for `FileSystem` will be the same in every other class. Both the Composite and Observer extensions fall into this category, too. On the other hand, the Visitor and Decorator extensions are cases where the larger the number of elements is (visitor nodes and interface operations, respectively), the larger the injected code. Notice the case of `IElement` where these two patterns were applied, resulting in an effective number of LOC than is more that four times larger than the manually written code. Therefore, these extensions are more powerful in terms of the transformation of source code.

6 Discussion

The novelty of our approach does not pertain to the form of instantiating design patterns, but instead in the automatization of their instantiation according to common idioms. Although we believe that our language constructs are a powerful abstraction, bringing the implementation of design patterns to the programming language level has some drawbacks, as pointed out by John Vlissides in a debate on the issue of having patterns as language constructs [6]. The more automation we aim at, the less flexible the pattern instantiation becomes, given that code generation approaches that bridge higher levels of abstraction to lower ones necessarily have to compromise flexibility to some extent. Even though we took into account the possibility of parameterizing pattern instantiations, our solutions will naturally not fit any context that a programmer might come up with. However, when certain patterns need to be instantiated in such a way that the annotations did not anticipate, programmers can always implement them manually without benefiting from the transformations.

We argue that the traceability benefit of having the annotations present in the source code consists of an important advantage, given that the documentation of design patterns in the code has revealed beneficial for system maintenance

[14]. Annotations are types in the programming language, and the associated validations ensure that they are applied in the correct locations and obey to other constraints. In this way, annotations can be seen as a structured form of documentation and compliance verification, and hence, they also consist of a robust means to document and enforce design issues. This is an advantage when compared with unstructured documentation text present in source code comments, which is somewhat fragile and easily becomes outdated, or external artifacts such as design documents, which often suffer from the problem of architectural erosion [12].

Given that our annotations indicate the patterns and their roles we believe that they are easy to understand from a code reading perspective, since the programmer is basically attaching labels to code elements using a familiar construct (the annotations). Further, the existence of dedicated language constructs also promotes pattern learning and experimentation. However, we believe that the language extensions in some cases do not dismiss programmers of having to understand how the patterns actually work internally.

We demonstrated how some of the widely used patterns are suitable to be addressed in language extensions. Other potentially more specific patterns (e.g., concurrency, or related to a particular platform) could also be addressed with this mechanism. The implementation of our extensions was by no means technically trivial, given that it had to be based directly on the compiler API. A more friendly abstraction for writing transformations would make easier to define extensions. However, we envision that this kind of extensions would be developed by specialized programmers and packaged as if they were libraries, in order to have some degree of reliability and standardization.

7 Related Work

Previous works have proposed dedicated language constructs to address design patterns. Jan Bosch [2] proposed a design-level support for generating design pattern implementations. When the design is finished, the model is able to generate the equivalent C++ code. The problem with this approach is that it works as a code generation tool that only provides support at the design stage, and the generated code will resemble manual implementation. Since the C++ code does not keep up with the pattern instance specifications, as opposed to our approach, the problems of traceability and comprehension at the source code level are not addressed.

OpenJava [16] is a macro system for Java that offers a compile-time reflective means that can inject source code in a similar way as Lombok. Therefore, OpenJava could be an alternative means for implementing our approach for design pattern instantiation. However, it implies using syntax extensions to Java for the declaration of macro expansions, whereas Lombok does not (it relies on existing language constructs, the annotations).

FRED [9] is an environment that supports the implementation of design patterns in Java. The implementation of design patterns is aided through an

incremental sequence of tasks until all the mandatory tasks are completed. A task is considered to be the creation of small elements like classes, methods and fields. This incremental process has to be done every time one wants to instantiate a pattern, which can be time-consuming. Since the pattern instantiation is supported by the environment, we have no assistance if we use the resulting code on another Java development environment.

Using a different strategy for implementing design patterns, AspectJ[3] was proposed as a suitable means [10] with modularity improvements that make possible to encapsulate pattern instantiations in independent modules – the *aspects*. The main drawback of this approach is the fact that in order to instantiate the patterns programmers must have some technical skills with respect to AspectJ. As with our approach, the aspect-based pattern instantiations also address traceability at the source code level, because the pattern instantiations are given in well-defined entities (all the instantiations of a given pattern extend the same abstract aspect). However, issues pertaining to pattern inter-dependency and interaction might consist of an issue, as reported by a study on the scalability of pattern modularity using the aspect-based approach [4].

JavaStage [1] is an extension to Java that encompasses programming constructs to represent *roles*. The notion of role has a dedicated module that may define fields and methods that enhance the classes to which the role is bound (using a declarative-style primitive on their definition). The definition of the role modules have a similar purpose than the AST transformations in our approach, whereas the role binding primitives relate to our annotations. Defining extensions using roles is a more elegant and easy way in contrast to the AST transformations used in our approach. Namely, the authors illustrate their approach with the Observer pattern. However, complex cases that require enhancements across multiple types, e.g., as our Visitor pattern transformation, might not be possible to address using roles due to the transformation complexity.

8 Conclusions

In this paper we described a set of Java extensions addressing widely used design patterns, whose instantiation can be achieved partly through source code transformation. We conclude that at least the patterns we presented here are suitable to be addressed with dedicated language constructs, given the considerable amount of elements that can effectively be generalized, as demonstrated in the example instantiations. The provided annotations consist of powerful high-level language constructs, which besides automating parts of the pattern instantiation, also mitigate pattern traceability and comprehension issues, given that patterns instances are represented by first-class entities. Although the extensions were proven to work, research on their suitability to real projects still has to be carried out to evaluate if the balance between automation and flexibility is satisfactory. As future work, we plan to refactor an existing framework using our annotations for this purpose.

[3] www.eclipse.org/aspectj

References

1. Barbosa, F.S., Aguiar, A.: Using roles to model crosscutting concerns. In: Proceedings of the 12th Annual International Conference on Aspect-oriented Software Development. pp. 97–108. AOSD '13, ACM, New York, NY, USA (2013)
2. Bosch, J.: Design patterns as language constructs. Journal of Object-Oriented Programming 11(2), 18–32 (1998)
3. Brooks, Jr., F.P.: No silver bullet - essence and accidents of software engineering. Computer 20(4), 10–19 (Apr 1987)
4. Cacho, N., Sant'Anna, C., Figueiredo, E., Garcia, A., Batista, T., Lucena, C.: Composing design patterns: A scalability study of aspect-oriented programming. In: Proceedings of the 5th International Conference on Aspect-oriented Software Development. pp. 109–121. AOSD '06, ACM, New York, NY, USA (2006)
5. Cachopo, J., Rito-Silva, A.: Versioned boxes as the basis for memory transactions. Sci. Comput. Program. 63(2), 172–185 (2006)
6. Chambers, C., Harrison, B., Vlissides, J.: A debate on language and tool support for design patterns. In: Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 277–289. ACM (2000)
7. Fowler, M.: Patterns of Enterprise Application Architecture. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2002)
8. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Pearson Education (1994)
9. Hakala, M., Hautamäki, J., Koskimies, K., Paakki, J., Viljamaa, A., Viljamaa, J.: Architecture-oriented programming using FRED. In: Proceedings of the 23rd International Conference on Software engineering. pp. 823–824. IEEE Computer Society (2001)
10. Hannemann, J., Kiczales, G.: Design pattern implementation in java and aspectj. In: Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications. pp. 161–173. OOPSLA '02, ACM, New York, NY, USA (2002)
11. Nobakht, B., de Boer, F., Bonsangue, M., de Gouw, S., Jaghoori, M.: Monitoring method call sequences using annotations. Science of Computer Programming 94, Part 3(0), 362 – 378 (2014)
12. Perry, D.E., Wolf, A.L.: Foundations for the study of software architecture. SIGSOFT Softw. Eng. Notes 17(4), 40–52 (Oct 1992)
13. Prechelt, L., Unger, B., Tichy, W.F., Brossler, P., Votta, L.G.: A controlled experiment in maintenance: comparing design patterns to simpler solutions. Software Engineering, IEEE Transactions on 27(12), 1134–1144 (2001)
14. Prechelt, L., Unger-Lamprecht, B., Philippsen, M., Tichy, W.F.: Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance. Software Engineering, IEEE Transactions on 28(6), 595–606 (2002)
15. Soukup, J.: Implementing patterns. In: Coplien, J.O., Schmidt, D.C. (eds.) Pattern Languages of Program Design, chap. Implementing Patterns, pp. 395–412. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA (1995)
16. Tatsubori, M., Chiba, S., Killijian, M., Itano, K.: OpenJava: A class-based macro system for Java. In: OOPSLA'99 Workshop on Reflection and Software Engineering (1999)