Replacing Object-Oriented Design Patterns with Intrinsic Aspect-Oriented Design Patterns

Pavol Bača and Valentino Vranić
Institute of Informatics and Software Engineering
Faculty of Informatics and Information Technologies
Slovak University of Technology
Bratislava, Slovakia
Email: pavol.baca1@gmail.com, vranic@fiit.stuba.sk

***Abstract*—This paper shows how intrinsic aspect-oriented design patterns can be used to implement object-oriented design patterns in order to achieve better composability compared to both original implementations of object-oriented design patterns and their aspect-oriented reimplementations. Director, an aspect-oriented pattern known for its ability to replace a number of object-oriented patterns, owes its success to its obliviousness to the problems addressed by the patterns it can be used instead of. The paper proposes how two other intrinsic aspect-oriented patterns can be used as an alternative to object-oriented patterns: Worker Object Creation can be successfully used instead of Proxy, while Cuckoo's Egg can replace Singleton, Abstract Factory, and Flyweight. To check that these replacements are valid beyond their isolated application, a small study has been conducted in which a composition of Worker Object Creation and Cuckoo's Egg in place of the composition of the original object-oriented patterns Proxy, Singleton, Abstract Factory, and Flyweight has been used. To assess the implementation of intrinsic aspect-oriented patterns compared to object-oriented patterns they replace and their aspect-oriented reimplementations, software metrics relevant both to object-oriented and aspect-oriented code mainly with respect to separation of concerns have been applied.**

***Keywords*-design patterns; intrinsic aspect-oriented design patterns; design pattern composition**

I. Introduction

Since most aspect-oriented programming languages represent extensions of object-oriented programming languages, object-oriented design patterns can be applied in them directly. As patterns from the catalog by Gamma et al. [1]—known as GoF (Gang of Four) patterns—hold a prominent position being a part of everyday vocabulary of software developers, it comes as no surprise they were the first place to look for patterns valid also for aspect-oriented programming. GoF patterns were reimplemented in AspectJ, the most popular aspect-oriented programming language (based on Java), to separate crosscutting concerns in their application soon after the advent of aspect-oriented programming [2].

Part of these crosscutting concerns arises from pattern composition. As software elements, patterns have to be composed with each other in order to establish a working piece of software. For object-oriented design patterns, this typically means a pattern implementation shares one or more classes with another pattern implementation (to which each pattern adds its own methods) or invokes one or more of its methods. Given that each pattern constitutes a separate concern, such a composition introduces their crosscutting. Apparently, less crosscutting improves composability by making it easier to actually compose patterns, as well as to take a pattern out of a composition.

Since aspect-oriented programming provides mechanisms to modularize crosscutting concerns, aspect-oriented reimplementations of GoF patterns were initially believed to be a priori better composable than the original, object-oriented implementations of the same patterns [2], [3]. However, a qualitative and quantitative assessment [4] showed this is not so in all cases.

On the other hand, there are patterns intrinsic to aspect-oriented programming. These patterns rely on specific aspect-oriented programming constructs and therefore can't be applied in object-oriented programming. The composition of intrinsic aspect-oriented design patterns is often as simple as merely including them in the implementation, i.e. without having to modify other patterns that participate in it [5], [6].

Aspect-oriented reimplementations of object-oriented patterns have been developed with no intention to keep them being patterns in the context of aspect-oriented programming (they become a simple application of aspect-oriented constructs). However, some of such aspect-oriented reimplementations of object-oriented patterns actually represent intrinsic aspect-oriented patterns. For example, the Singleton pattern aspect-oriented reimplementation [2] represents a Cuckoo's Egg aspect-oriented pattern application.

The objective of this paper is to explore to what extent intrinsic aspect-oriented patterns can be used to replace object-oriented patterns in order to benefit from their improved composability. For this, a study has been conducted in AspectJ. Its results are presented in the rest of the paper. Section II discusses the Director aspect-oriented pattern known for its ability to replace a number of object-oriented patterns. Sections III and IV discover how two other intrinsic design patterns, namely Worker Object Creation and

Cuckoo's Egg, can be used to replace several GoF patterns. Section V presents a study that confirms the validity of the replacements in the composition involving two patterns. It also compares the compositions quantitatively. Section VI discusses related work. Section VII concludes the paper and indicates some directions of further work.

II. Director

The Director design pattern enables to define roles of behavior and enforce their implementation to classes. An aspect defines the interaction between these roles by which it actually affects the interaction between the classes. With Director, it is possible to decouple generic and reusable behavior from the implementation classes of a specific application [7].

The Director pattern can be identified in several aspect-oriented reimplementations of GoF patterns [2], [8]. However, it is exactly such a general applicability of Director that raises a question whether it really internally covers such a broad spectrum of problems addressed by object-oriented patterns. With respect to Alexander's definition of a pattern by which it is a rule that connects the context, problem, and solution [9], it is legitimate to ask what problem does Director as a pattern actually address. The problem solved by Director is the separation of the generic reusable behavior from the specific implementation in classes by defining roles of behavior. This problem is unlike anything solved by GoF patterns and their aspect-oriented implementations. In fact, the key parts of the original GoF pattern "replaced" by Director remain and Director actually just ensures their interconnection and enforcement upon the underlying code.

Let's take as an example the Prototype pattern [1], which enables a class to create instances of classes unknown to it by making clones out of the prototype instances of these classes that have been supplied to it previously. In the aspect-oriented reimplementation of Prototype based on Director, Director by itself does not solve the problem of cloning a prototype—it uses Prototype for this [2]:

```
public abstract aspect PrototypeProtocol {
  protected interface Prototype {
  }
  public Object Prototype.clone()
    throws CloneNotSupportedException {
    return super.clone();
  }
  public Object cloneObject(Prototype object) {
    try {
      return object.clone();
    }
    catch (CloneNotSupportedException ex) {
      return createCloneFor(object);
    }
  }
  protected Object createCloneFor(Prototype object) {
    return null;
  }
}
```

The PrototypeProtocol aspect provides a general Prototype AspectJ implementation. It introduces the clone() method implementation into all the classes that implement the Prototype interface defined inside of it using so-called *inter-type declarations*. This just invokes the assumed existing cloning mechanism in these classes. The cloning itself is realized by the cloneObject() method of the aspect that in turn uses the createCloneFor() method to do real cloning for classes that do not implement cloning.

The actual implementation of the createCloneFor() method comes to place in concrete aspects that are aware of the application domain details. Assume we want to clone shapes in a graphic tool. A concrete aspect enforces cloning to the Shape class using another type of inter-type declarations and implements the cloning itself by overriding the createCloneFor() method:

```
public aspect ShapePrototypes extends PrototypeProtocol {
  declare parents: Shape implements Prototype;

  protected Object createCloneFor(Prototype object) {
    ...
  }
}
```

It is sufficient to add these two aspects to the rest of the application code. Upon the first use of the Shape class, a ShapePrototypes aspect instance will be created. It may be accessed by the **aspectOf**() method. The actual cloning is then performed like this:

```
Shape shape = (Shape) ShapePrototypes.aspectOf().
  cloneObject(shapePrototype);
```

It can be said that in the resulting code Director predominates over Prototype, but not that it replaces Prototype as Prototype remains present at the problem solution level.

III. Worker Object Creation

Separating the primary functionality from managing its execution in threads is the problem solved by the Worker Object Creation design pattern. A method call containing the functionality of interest is caught dispatched to another context that might be even in a different thread.

Technically speaking, the functionality is caught by a construct called *pointcut* that enables to express declaratively certain types of points in the execution called *join points*. Another aspect-oriented construct called *advice* can be applied to join points and add some other functionality before, after, or even instead of them. The last kind of advice is called *around advice* and this the one used in Worker Object Creation.

A popular application of Worker Object Creation is to catch all the calls to graphical user interface controls implemented in Swing as they have to be executed in its event dispatching thread to ensure their sequential execution [10]. Apart from reducing the number of lines of code, Worker Object Creation provides an opportunity to manage creating these threads in one place [10]. However, this pattern is much more powerful. Executing a method in a new thread delays its execution, so it can be seen as a special case of a delayed method execution. This will be shown on an example of replacing the Proxy pattern with the Worker Object Creation pattern.

The Proxy pattern is useful when a versatile or sophisticated reference to an object is needed rather than a simple pointer [1]. There are several situations in which the pattern is applicable. One of them is a smart reference which performs loading a persistent object into a memory when it is referenced for the first time. There is a condition that reference class (Proxy) must be of the same type as the referenced class [1].

Worker Object Creation can be used in such situations instead of the Proxy pattern. This is possible because both patterns solve similar problems and, as was mentioned above, the problem solved by Worker Object Creation is more general.

Consider a smart toolbar in an application graphical user interface as an example. In order not to waste the time of a user who doesn't need the toolbar, the toolbar and tool buttons are not created right at the application start, but only on a user demand. Figure 1 shows a class diagram with Worker Object Creation applied to this problem.

Since UML lacks the appropriate syntax that would correspond directly to Java anonymous classes, a static nesting in combination with the «anonymous» stereotype has been used for this purpose [11]. Aspects are modeled as classes with the «aspect» stereotype. Both advices and inter-type declarations (see Section II) are modeled as methods.

The pattern is implemented by the ToolbarProxy aspect. The aspect captures calls to the ToolbarImpl class constructor with an around advice and postpones its execution which, in turn, postpones the toolbar loading. Here is the implementation:

```
public aspect ToolbarProxy {
  public ToolbarImpl.new(ToolbarProxy a) {
  }
  ToolbarImpl around(): call(ToolbarImpl.new()) {
    return new ToolbarImpl(this) {
      private ToolbarImpl toolbarImpl;

      public void setVisible(boolean visible) {
        if (toolbarImpl == null)
          toolbarImpl = proceed();

        toolbarImpl.setVisible(visible);
      }
      ...
      // other methods using the original class instance
    };
  }
}
```

Inside of the around advice, a proxy reference anonymous class is created. It inherits from the ToolbarImpl class (for this reason the ToolbarImpl class can't be final). The anonymous class implements the setVisible() method which makes the toolbar visible. The implementation of the method, which overrides the method in the ToolbarImpl class, invokes the captured constructor with the **proceed**() statement if the toolbar has not been loaded. Thus, the anonymous class plays the role of the Proxy class in the original GoF Proxy pattern.

For the needs of the anonymous class it may be necessary to define a new constructor of the ToolbarImpl class that does nothing beside creating a new toolbar. The constructor with no parameters can be easily added by an inter-type declaration, but there may be a clash with the no parameter constructor if it is already implemented in the class itself. Even if this is not so, it is hard to predict how this class is going to develop in future. This may be avoided by adding a constructor with a parameter type that is hardly going to be used by anyone and the aspect type itself appears to be quite convenient for this.

IV. Cuckoo's Egg

The goal of the Cuckoo's Egg pattern is to control and change class instantiation by capturing constructor or factory method calls. Constructor calls are caught by an around advice (see Section III) followed by returning a new object of the same or derived type [7]. We will see how this pattern can be applied instead of three GoF patterns: Singleton, Abstract Factory, and Flyweight.

The Singleton pattern is used to ensure a class will have only one instance and also to provide a global access point to it [1]. The Singleton aspect-oriented reimplementation is based on catching a constructor call by an around advice and looking up the existing instance in a hash map [2]. The existing instance is returned or the **proceed**() statement is executed in order to produce it. This is actually an application of the Cuckoo's Egg pattern.

The Abstract Factory pattern is used to provide an interface for creating families of objects without specifying the classes [1]. An aspect-oriented equivalent is based on adding inter-type declarations to an interface which is implemented by a factory class. Another solution is to use the Cuckoo's Egg pattern, which will be demonstrated on an example. Consider two families of shape classes: 2D and 3D shapes. The class diagram that shows the application of Cuckoo's Egg to this problem is depicted in Figure 2.

Because an abstract class can't have a constructor, a static factory method is used instead. A **call**() pointcut catches the calls to this static method. Subsequently, the abstract factory aspect and concrete aspects play the roles of factory classes in the Abstract Factory pattern.

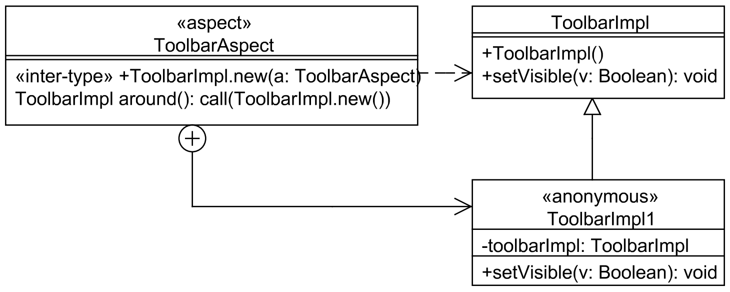
«aspect»
ToolbarAspect
«inter-type» +ToolbarImpl.new(a: ToolbarAspect)
ToolbarImpl around(): call(ToolbarImpl.new())
ToolbarImpl
+ToolbarImpl()
+setVisible(v: Boolean): void
«anonymous»
ToolbarImpl1
-toolbarImpl: ToolbarImpl
+setVisible(v: Boolean): void

Figure 1. Worker Object Creation as a replacement for Proxy.

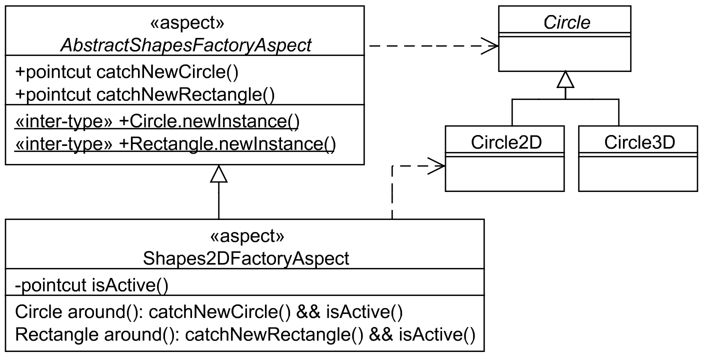
«aspect»
AbstractShapesFactoryAspect
+pointcut catchNewCircle()
+pointcut catchNewRectangle()
«inter-type» +Circle.newInstance()
«inter-type» +Rectangle.newInstance()
Circle
Circle2D
Circle3D
«aspect»
Shapes2DFactoryAspect
-pointcut isActive()
Circle around(): catchNewCircle() && isActive()
Rectangle around(): catchNewRectangle() && isActive()

Figure 2. Cuckoo's Egg as a replacement for Abstract Factory.

In our example, a new instance of a circle is created by a call to the newInstance() factory method inserted into the abstract Circle class by an inter-type declaration. It is more straightforward and natural than calling a factory class to get a new instance. If there is no concrete factory aspect, an IllegalStateException is thrown, which is analogous to the situation when no factory class is defined.

Flyweight is a GoF pattern similar to Singleton. It is used to avoid a huge memory consumption by a number of equal objects [1]. The aspect-oriented reimplementation of Flyweight is based on an aspect that handles a hash table with existing instances [2], which is similar to the aspect-oriented reimplementation of Singleton, but it is not based on an advice [2].

As with Singleton, Cuckoo's Egg can be used instead of Flyweight, too. The difference is that **proceed**() is called every time a new object is needed. A new instance is then compared to existing instances in the aspect hash map and kept only if it is different.

Note that the example from Section IV exhibits some features of the Cuckoo's Egg pattern: a constructor call is caught by an advice and a new object of the same type is created then and returned. However, the dominant issue is that the captured functionality—which happens to be a constructor—is sent to a different context for an unconditional execution. Thus, the original class instance is not dropped as with Cuckoo's Egg.

V. Evaluation: Design Pattern Composition

Previous sections introduced examples of replacing object-oriented design patterns by intrinsic aspect-oriented design patterns. Worker Object Creation has been used instead of Proxy. Cuckoo's Egg has been used to replace three GoF patterns: Singleton, Abstract Factory, and Flyweight.

To check that these replacements are valid beyond their isolated application, we conducted a small study based on the implementation of a toy graphic tool that enables drawing simple geometric shapes and writing text.[1] The tool supports 2D and 3D mode, which is statically configured before the tool is started. The shapes are drawn by clicking the buttons in the toolbar. In order not to waste the time of users who don't need the toolbar, the toolbar and tool buttons are not created right at the application start, but only on a user demand. The graphic tool also offers an image gallery. As with the toolbar, the gallery is created on demand, too.

The tool supports text writing, including arabesque letters. The tool reduces memory costs by reusing an arabesque letter image that has already been created. Only one instance of a gallery is needed, as well as one instance of an arabesque letter storage. Analogously to the 2D and 3D mode configuration, it is possible to configure an arabesque letter font. Another way is to configure the font by the tool interface if it has been created. A font is loaded to the

[1]The complete code is available at http://fiit.stuba.sk/~vranic/proj/dp/Baca/aoOoPatterns.zip.

memory when it becomes necessary.

This simple example offers an opportunity to implement six pairs of pattern compositions. Three implementations have been created: an object-oriented one, with object-oriented design pattern composition, and two aspect-oriented implementations one of which includes aspect-oriented reimplementations of the original object-oriented patterns, while the other one employs the corresponding intrinsic aspect-oriented patterns instead.

A. Object-Oriented Implementation

In the object-oriented implementation, Abstract Factory with Singleton have been applied to enable configuring the application to be used in 2D or 3D mode and to ensure only one its instance is created. Proxy with Abstract Factory have been applied to let the toolbar, for shape painting, be loaded on user demand by using a proxy reference. Proxy with Singleton have been applied to let the single gallery instance be loaded on user demand by using a proxy reference. Singleton with Flyweight have been applied to create a single storage instance for arabesque letters. Abstract Factory with Flyweight have been applied to enable static configuration of an arabesque letter font. Finally, Proxy with Flyweight have been applied to enable user configuration of an arabesque letter font.

The aspect-oriented reimplementation of object-oriented patterns was developed to demonstrate an aspect-oriented solution is indeed possible. The Hannemann–Kiczales [2] aspect-oriented reimplementations of the Proxy, Singleton, Abstract Factory, and Flyweight patterns have been applied.

B. Implementation with Intrinsic Aspect-Oriented Patterns

Finally, the corresponding intrinsic aspect-oriented patterns have been applied instead of the corresponding object-oriented patterns. Their application has actually already been introduced in Sections III–IV. A few deviations from the original object-oriented implementation showed up in the implementation. The first one lies in not having an explicit Singleton implementation since the only instance of Abstract Factory—i.e., the Cuckoo's Egg aspect—is ensured by an implicit **issingleton**() aspect instantiation modifier. The second deviation lies in the conflict between the pointcuts implemented in Cuckoo's Egg and Worker Object Creation. An additional **declare precedence** statement had to be added into both aspects.

Many aspect-oriented reimplementations of object-oriented patterns are based on the Director pattern, which makes a significant part of them reusable (see Section II). Such reusability is not common among intrinsic aspect-oriented patterns, but it seems that they affect the parts of other patterns in composition to a lesser degree than the aspect-oriented reimplementations of the corresponding object-oriented patterns. For example, the Proxy pattern aspect-oriented reimplementation needs a ToolbarProxy class to be added and its instance to be used to access the toolbar object. Thus, adding this pattern affects at least one other class and also a creation of a new class.

On the other hand, no part of any other pattern is affected by the application of the Worker Object Creation pattern, Proxy's aspect-oriented counterpart, nor there is a need to create some other class. Thus, with respect to composability, an intrinsic aspect-oriented pattern is a good alternative. The composition is performed simply by including the corresponding pattern in the application build. A pattern is removed by excluding it from the build. Other intrinsic patterns studied here are also composable by a simple inclusion.

C. Quantitative Assessment

To assess the implementation of intrinsic aspect-oriented patterns compared to object-oriented patterns they replace and their aspect-oriented reimplementations, software metrics relevant both to object-oriented and aspect-oriented code [3], [4] used in several studies about aspect-oriented pattern implementations [3] have been applied:

- separation of concerns: Concern Diffusion over Components (CDC), Concern Diffusion over Operations (CDO), and Concern Diffusion over Lines of Code (CDLOC)
- coupling: Coupling Between Components (CBC) and Depth Inheritance Tree (DIT)
- cohesion: Lack of Cohesion in Operations (LCOO)
- size: Lines of Code (LOC), Number of Attributes (NOA), and Weighted Operations per Components (WOC)

Separation of concerns is measured inverted (interconnection of concerns) as diffusion of concerns over classes and aspects (CDC), methods and advices (CDO), and lines of code (CDLOC) [3]. More separated concerns tend to be less diffuse.

Coupling is observed by a number of classes and aspects to which a class or an aspect is coupled (CBC) and by its depth in the inheritance tree (DIT) [3].

Cohesion is followed inverted (lack of cohesion) as a number of method and/or advice pairs that do not access the same attribute (LCOO).

Size is measured in terms of simple lines of code (LOC), a number of attributes of each class or aspect (NOA), and a number of methods and advices of each class or aspect and the number of its parameters (WOC) [3].

Since improved composability reflects significantly in separation of (crosscutting) concerns (as was discussed in the introduction of the paper), the most relevant metrics for this study are those about separation of concerns. However, coupling and cohesion are also important because they express how modular the implementation is.

Coupling, cohesion, and size metrics have been evaluated automatically using Borland Together 2008 [12] with AJDT plugin for Eclipse [13]. The separation of concerns metrics have been calculated manually line by line.

In aspect-oriented reimplementations of object-oriented patterns, reusable abstract aspects, denoted sometimes as protocols [2], have not been included into the evaluation. The reason for this is that these aspects act as library classes, so they are not included in results along with, for example, the java.util package classes.

As a result of measurements, there have been some additional findings about the Proxy pattern replacement. According to three metrics from the suite, the Proxy pattern substitution by Worker Object Creation seems to be better compared to Proxy's aspect-oriented reimplementation. Improvements of WOC (41%), CBC (28%), and CDO (48%) are a consequence of reducing the number of classes and number of operation parameters.

Figure 3 shows comparison from the separation of concerns perspective. The numbers above the graph indicate percentage by which the substitution by intrinsic patterns is better than the aspect-oriented reimplementation. Using fewer classes in implementation with intrinsic aspect-oriented patterns improved two of three metrics—CDC (21%) and CDO (37%). CDLOC got worse, but only slightly (-3%).

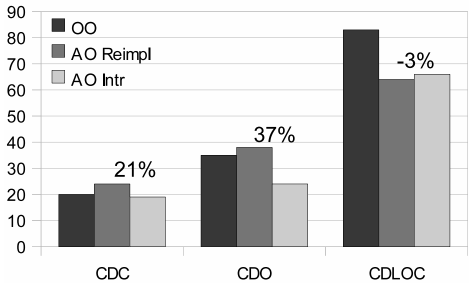

Figure 3. Separation of concerns.

Figure 4 shows comparison from the perspective of coupling and cohesion. Coupling has improved with intrinsic aspect-oriented patterns as can be seen from both coupling metrics that have been used: CBC (21%) and DIT (9%). LCOO metrics for cohesion improved too (24%).

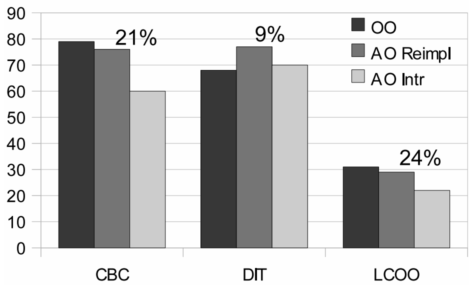

Figure 4. Coupling and cohesion.

Figure 5 shows comparison from the perspective of size. Except for the improvement of WOC (27%) no other improvement has been achieved. NOA (-10%) is worse for intrinsic aspect-oriented patterns than for reimplementations. With respect to this result, it was detected that NOA metrics as evaluated by Borland Together 2008 tool counts both pointcuts and advices as attributes. Thus, using more aspects inside intrinsic aspect-oriented pattern compositions increases the NOA value.

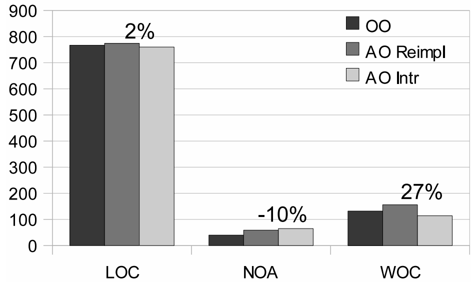

Figure 5. Code size.

The assessment demonstrated that substitutions by intrinsic aspect-oriented patterns don't become less convenient than aspect-oriented reimplementation of object-oriented patterns. Moreover, they exhibited better results with respect to separation of concerns and coupling in composition.

VI. Related Work

Much of the effort related to object-oriented design patterns in aspect-oriented programming has been focused on a direct aspect-oriented reimplementation of GoF patterns in AspectJ. Some reimplementations have been reported in other aspect-oriented languages, e.g. CeasarJ [14] or ComposeJ [15], but the most systematic work was an aspect-oriented reimplementation of all 23 GoF patterns in AspectJ by Hannemann and Kiczales which demonstrated improvement in code locality, reusability, composability, and pluggability of 17 of them [2].

An independent qualitative assessment of GoF pattern reimplementations discovered an improvement of the separation of concerns related to pattern realization mechanism, but without a significant advance in reuse [3].

Further evaluation of (mostly pair) compositions of aspect-oriented GoF pattern reimplementations for separation of concerns, coupling, cohesion, and conciseness revealed that the quality of aspect-oriented reimplementations depends on the patterns involved, the composition intricacies, and the application requirements [4]. The results were not always in favor of aspect-oriented reimplementations.

The idea of using intrinsic aspect-oriented patterns to replace object-oriented patterns comes from the reported ability of the Director pattern to act in place of some GoF patterns [7], [8] that has been shown here to be overestimated. However, a hope was given to other intrinsic aspect-oriented patterns.

The use of the Cuckoo's Egg pattern proposed in this paper is similar to the Advised Factory Method (Advised Creation Method)^2 idiom [16].

VII. Conclusions and Further Work

Intrinsic aspect-oriented design patterns can be used to implement object-oriented design patterns. They are expected to achieve a better composability compared to both original implementations of object-oriented design patterns and their aspect-oriented reimplementations. Even though, they seem to be an alternative, not a preference.

The Director pattern is known for its ability to replace a number of object-oriented patterns. This paper pointed out that part of Director's success lies in its obliviousness to the problems addressed by the patterns it can be used instead of. Since Director does not replace the original pattern at the problem solution level, it should not be counted as its substitution.

On the other hand, two other intrinsic aspect-oriented patterns have been discovered that can be used as substitutions of object-oriented patterns. Worker Object Creation can be successfully used instead of Proxy. Cuckoo's Egg can replace Singleton, Abstract Factory, and Flyweight. The validity of these substitutions has been confirmed by the composition of Worker Object Creation and Cuckoo's Egg in place of the composition of the original object-oriented patterns Proxy, Singleton, and Abstract Factory.

In a quantitative assessment, better results have been observed with intrinsic aspect-oriented patterns than with aspect-oriented reimplementations. However, it must be admitted that the results were obtained for a quite limited number of patterns: out of 23 GoF patterns only four of them have been successfully substituted by intrinsic aspect-oriented patterns.

Possible directions of further work include exploring the potential of other intrinsic aspect-oriented patterns to replace object-oriented patterns. Other aspect-oriented programming languages should be engaged, too, especially those that support symmetric aspect-oriented programming (e.g., HyperJ or CaesarJ). Yet another possible direction is to investigate how replacement of object-oriented patterns fits into the context of refactoring, both at the implementation and modeling level [17].

Acknowledgments

The work was supported by the Scientific Grant Agency of Slovak Republic (VEGA) grant No. VG 1/0508/09.

This contribution/publication is also a partial result of the Research & Development Operational Programme for the project Research of Methods for Acquisition, Analysis and Personalized Conveying of Information and Knowledge, ITMS 26240220039, co-funded by the ERDF.

^{2}originally denoted as "adviced"

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[2] J. Hannemann and G. Kiczales, "Design pattern implementation in Java and AspectJ," in *Proc. of 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2002*. Seattle, Washington, USA: ACM, 2002, pp. 161–173.

[3] A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena, and A. von Staa, "Modularizing design patterns with aspects: A quantitative study," in *Proc. of 4th International Conference on Aspect-Oriented Software Development, AOSD 2005*. Chicago, Illinois, USA: ACM, 2005, pp. 3–14.

[4] N. Cacho, C. Sant'Anna, E. Figueiredo, A. Garcia, T. Batista, and C. Lucena, "Composing design patterns: A scalability study of aspect-oriented programming," in *Proc. of 5th International Conference on Aspect-Oriented Software Development, AOSD 2006*. Bonn, Germany: ACM, 2006, pp. 109–121.

[5] R. Menkyna, V. Vranić, and I. Polášek, "Composition and categorization of aspect-oriented design patterns," in *In Proc. of 8th International Symposium on Applied Machine Intelligence and Informatics, SAMI 2010*. Herl'any, Slovakia: IEEE, Jan. 2010.

[6] R. Menkyna, "Towards combining aspect-oriented design patterns," in *Proc. Informatics and Information Technologies Student Research Conference, IIT.SRC 2007*, M. Bieliková, Ed., Bratislava, Slovakia, Apr. 2007, pp. 1–8.

[7] R. Miles, *AspectJ Cookbook*. O'Reilly, 2004.

[8] R. Menkyna, "The Director as a connection between object-oriented and aspect-oriented design," in *Proc. of Informatics and Information Technologies Student Research Conference, IIT.SRC 2009*, M. Bieliková, Ed., Bratislava, Slovakia, Apr. 2008.

[9] C. Alexander, *The Timeless Way of Building*. Oxford University Press, 1979.

[10] R. Laddad, *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning, 2003.

[11] R. C. Martin, *UML for Java Programmers*. Prentice Hall, 2003.

[12] Borland Software Corporation, "Borland Together 2008," 2010, http://www.borland.com/us/products/together/.

[13] The Eclipse Foundation, "AJDT: AspectJ Development Tools," 2010, http://www.eclipse.org/ajdt/.

[14] E. Sousa and M. P. Monteiro, "Implementing design patterns in CaesarJ: An exploratory study," in *Proc. of the Workshop on Software engineering Properties of Languages and Aspect Technologies, SPLAT 2008 (at AOSD 2008)*. Brussels, Belgium: ACM, 2008, pp. 1–6.

[15] J. C. Wichman, “ComposeJ: The development of a preprocessor to facilitate composition filters in the java language,” Master’s thesis, University of Twente, 1999, http://trese.cs.utwente.nl/oldhtml/publications/msc_theses/wichman.thesis.pdf.

[16] S. Hanenberg, A. Schmidmeier, and R. Unland, “AspectJ idioms for aspect-oriented software construction,” in *Proc. of 8th European Conference on Pattern Languages of Programs, EuroPLoP 2003*, Irsee, Germany, Jun. 2003, pp. 617–644.

[17] M. Štolc and I. Polášek, “A visual based framework for the model refactoring techniques,” in *Proc. of 8th International Symposium on Applied Machine Intelligence and Informatics, SAMI 2010*. Herl’any, Slovakia: IEEE, Jan. 2010.