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

Pavol Bača **Valentino Vranić**

Institute of Informatics and Software Engineering
Faculty of Informatics and Information Technologies
Slovak University of Technology, Bratislava, Slovakia
vranic@fiit.stuba.sk

ECBS-EERC 2011, September 5–6, 2011, Bratislava, Slovakia

Overview

1 Object-Oriented Design Patterns and Aspect-Oriented Programming

2 Replacing Object-Oriented Patterns

3 Evaluation: Design Pattern Composition

4 Summary

Design Pattern Composition

- Object-oriented design patterns—and especially GoF[1] patterns—are a part of software developers' everyday vocabulary
- Patterns are applied mostly individually (contradictory to the idea of pattern languages), but they are often composed
- The very application of an object-oriented pattern involves it being interleaved with application logic
- Composition makes patterns interleaved with each other

[1] E. Gamma. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

A Pattern as a Concern: How to Separate It

- A pattern may be viewed as a concern
- We'd like to have concerns separated
- But patterns appear to *crosscut* application logic concerns
- Where object-orientation can't help anymore, there comes aspect-orientation: separation of crosscutting concerns
- Aspect-oriented reimplementations of object-oriented patterns resolve pattern crosscutting of application logic concerns[2]
- A qualitative and quantitative assessment showed this is not so with respect to pattern composition[3]

[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.

[3] N. Cacho et al. 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.

Aspect-Oriented Patterns and Composition

- There are patterns intrinsic to aspect-oriented programming: not applicable in object-oriented programming
- These patterns often can be composed by simply including them in the implementation without having to modify it
- We observed that some aspect-oriented reimplementations of object-oriented patterns actually represent intrinsic aspect-oriented patterns

Use Intrinsic Aspect-Oriented Patterns Instead

- Our idea: *use intrinsic aspect-oriented patterns to benefit from their simple and separated composition*
- This involved
  - Finding intrinsic aspect-oriented patterns that correspond to aspect-oriented reimplementations of object-oriented patterns
  - Evaluating them in composition to see whether they behave as the original object-oriented patterns
- We worked with three aspect-oriented patterns
  - Director
  - Worker Object Creation
  - Cuckoo's Egg

The Director Case

- Director was our original inspiration for its ability to replace many object-oriented patterns, but. . .
- It actually addresses the problem of separating the generic reusable behavior from the specific implementation in classes
- This is done by enforcing the corresponding roles of behavior to classes
- Key parts of the GoF patterns "replaced" by Director remain
- Director is actually just oblivious to the problems addressed by these patterns and can't be counted as their substitution

Worker Object Creation (1)

- Worker Object Creation[4] separates the functionality from managing its execution by enveloping it into a worker object
- A worker object is then sent for execution to a different context—usually in another thread

```
void around() : <pointcut> {
    Runnable worker = new Runnable () {
        public void run() {
            proceed();
    }
    invoke.Queue.add(worker); // execution − possibly deferred
}
```

[4] R. Laddad. AspectJ in Action: Enterprise AOP with Spring Applications. Second edition, Manning, 2009.

Worker Object Creation (2)

- A popular application: catch the calls to GUI controls in Swing and have them executed in its event dispatching thread

```
EventQueue.invokeLater(new Runnable() {
    public void run() {
        . . .
    }
});
```

- Proxy shields the functionality of an object by another, proxy object
- This can be done at object creation time by providing the proxy object

Worker Object Creation (3)

- An example: optimizing toolbar creation

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

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

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

Cuckoo's Egg

- Cuckoo's Egg[5] enables to exchange an object of one type with an object of another type
- This is achieved by capturing constructor or factory method calls

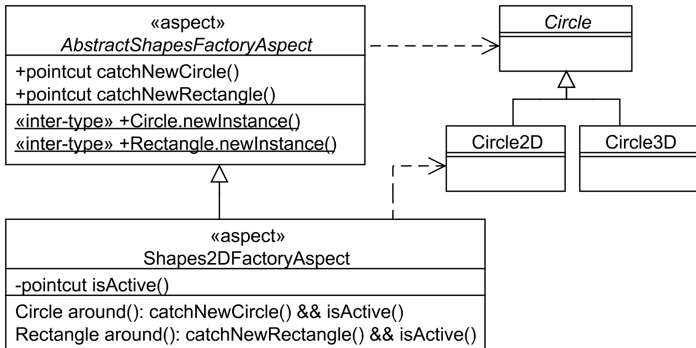
```
public aspect MyClassSwapper {
    public pointcut myConstructors(): call(MyClass.new());

    Object around(): myConstructors() {
        return new AnotherClass();
    }
}
```

- Applied instead of three GoF patterns: Singleton, Abstract Factory, and Flyweight
- Singleton: catch a constructor and look up an existing instance

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

Cuckoo's Egg Instead of Abstract Factory

- Abstract Factory is used to provide an interface for creating families of objects without specifying the classes
- Abstract factory methods can be introduced by an aspect



Cuckoo's Egg Instead of Flyweight

- Flyweight is used to avoid a huge memory consumption by a number of equal objects
- Like Singleton, but involves checking the existence of each instance

The Study

- To check that our pattern replacements are valid beyond their isolated application, we conducted a small study based on the implementation of a toy graphic tool[6]
- The tool supports drawing simple geometric shapes and writing text
- 2D and 3D mode statically configured before the tool is started
- The toolbar, tool buttons, and image gallery created upon a user demand

[6] http://fiit.stuba.sk/~vranic/proj/dp/Baca/aoOoPatterns.zip

The Patterns

- The study provided an opportunity to implement six pairs of pattern compositions
  - Abstract Factory + Singleton
  - Proxy + Abstract Factory
  - Proxy + Singleton
  - Singleton + Flyweight
  - Abstract Factory + Flyweight
  - Proxy + Flyweight
- These have been implemented in three ways by
  1. Object-oriented original pattern implementation
  2. Aspect-oriented pattern reimplementation
  3. Intrinsic aspect-oriented patterns

Aspect-Oriented Reimplementation

- The aspect-oriented reimplementation was developed to demonstrate an aspect-oriented solution is indeed possible
- Hannemann–Kiczales[7] aspect-oriented reimplementations of GoF patterns have been applied

[7] 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.

Implementation with Intrinsic Aspect-Oriented Patterns

- Aforementioned intrinsic aspect-oriented patterns have been successfully applied in the paired compositions of GoF patterns
- A few deviations from the original object-oriented implementation showed up:
  - No explicit Singleton implementation: the only instance of Abstract Factory—i.e., the Cuckoo's Egg aspect—is ensured by an implicit **issingleton**() aspect instantiation modifier
  - A conflict between the pointcuts implemented in Cuckoo's Egg and Worker Object Creation: resolved by a declare precedence statement

Observations

- Intrinsic aspect-oriented patterns are less generic than the aspect-oriented reimplementations of object-oriented patterns (many based on Director)
- But it seems intrinsic aspect-oriented patterns affect other patterns in composition to a lesser degree resulting
  - Simpler composition
  - A pattern is removed by simply excluding it from the build

Quantitative Assessment

- Improved composability reflects significantly in separation of (crosscutting) concerns
- Thus, 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
- We applied metrics used by Garcia et al.[8] and Cacho et al.[9]

---

[8] A. Garcia et al. 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.

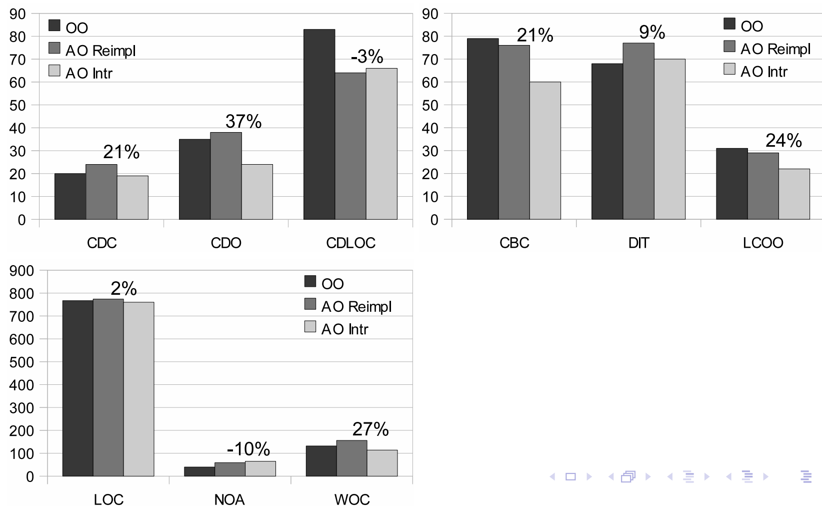
[9] N. Cacho et al. Composing design patterns: A scalability study of aspect-oriented programming. In *Proc. of 5th International Conference on Aspect-Oriented Software Development, AOSD 2006*, Bonn, Germany. ACM, 2006.

Metrics

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

Quantitative Assessment: Results

- In general, intrinsic aspect-oriented patterns had better results



Summary (1)

- Intrinsic aspect-oriented design patterns can be used to implement object-oriented design patterns
- They achieve a better composability compared to both original implementations of object-oriented design patterns and their aspect-oriented reimplementations
- Worker Object Creation can be successfully used instead of Proxy
- Cuckoo's Egg can replace Singleton, Abstract Factory, and Flyweight
- Director is not a substitution
- The validity of these substitutions has been confirmed in a pattern composition study
- Better composability of intrinsic aspect-oriented patterns has been observed and measured

Summary (2)

- However, the study was limited in size
- Only four GoF patterns have been successfully substituted by intrinsic aspect-oriented patterns
- Further work:
  - Extend the study to other patterns
  - Employ other aspect-oriented languages
  - Explore how replacement of object-oriented patterns fits into the context of refactoring