Design Pattern Implementation in Object Teams

João L. Gomes
CITI, Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa
2829-516 Caparica
PORTUGAL
clean_you@hotmail.com

Miguel P. Monteiro
CITI, Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa
2829-516 Caparica
PORTUGAL
mmonteiro@di.fct.unl.pt

ABSTRACT

Implementing the 23 Gang-of-Four design patterns in the aspect-oriented programming language Object Teams/Java (OT/J) yields modularity and reusability results roughly comparable to those obtained in a similar study of AspectJ, though not in the same exact set of patterns. Due to differences in composition mechanisms, the two languages seem complementary rather than overlapping. AspectJ is clearly superior to OT/J in terms of quantification capability while OT/J is clearly superior to AspectJ as regards extensibility of pattern modules.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**] Software Architectures – *information hiding, languages and patterns*.
D.3.3 [**Programming Languages**]: Language Contructs and Features – *classes and objects, patterns and polymorphism*.

General Terms

Design, Languages.

Keywords

Aspect-oriented programming, modularity, design patterns.

1. INTRODUCTION

Aspect-oriented Software Development is an emerging paradigm characterized by a systematic approach to the modularization of crosscutting concerns [6]. Many aspect-oriented languages (AOLs) were proposed in recent years, providing a significant variety of mechanisms for module composition. Even in languages that are backwards-compatible extensions to Java, as is the case of OT/J and AspectJ (the most popular AOL) we witness marked differences. Such variety provides a motivation for comparative analyses of AOLs, namely as regards actual support for modularity and composability. However, for most AOLs there is a dearth of studies by independent authors, particularly those involving dual implementations of common functionalities. This work contributes to filling this gap by presenting a complete collection of implementations in OT/J of all 23 Gang-of-Four (GoF) design patterns [1], as well as a preliminary analysis of the results obtained, in terms of the modularizations attained.

2. OBJECT TEAMS

This section assumes familiarity with AspectJ and is focused on presenting OT/J [4]. OT/J adds *teams* to Java, i.e., modules capable of enclosing a special kind of inner classes – *roles* – that represent the internal concepts of a collaboration of objects. Role classes are *virtual* [5], i.e., roles can be overridden and subject to dynamic dispatch, the same way as methods in mainstream object-oriented languages. The type system of OT/J supports *family polymorphism* [1], i.e., it ensures consistency between role instances, preventing the mixing of role instances from different teams. Each role can be bound to a specific class from an application through a *playedBy* relation that mimics inheritance.

Most AOLs support to some extent the *quantification* property, i.e., the ability to specify assertions over execution events of a program, so that the intended behavior of *aspect modules* – teams, in the case of OT/J – is implicitly called upon reaching any of the specified events. This way, AOLs such as OT/J and AspectJ compose additional behavior to existing programs without the need for invasive changes to the program's source code. However, OT/J restricts quantifiable events to those of a class bound to a specific role. Intended role behavior is expressed by the role, which can specify that its methods be implicitly called whenever specified events from the bound class occur.

3. APPROACH

Two different existing repositories of pattern implementations by independent authors were used as basis for this study[1], though it is primarily based on the study by Hannemann and Kiczales (HK), which comprises dual implementations in Java and AspectJ of the 23 GoF patterns. We reimplemented the HK Java examples in OT/J. To certify that pattern implementations are reusable, a second Java repository of GoF patterns was developed in OT/J as well. For each pattern, only modules we suceeded in using in the examples from both repositories are classified as reusable. In addition, we required modules to have non-abstract members to be taken into account in the analysis of reusability.

4. RESULTS

The HK study distinguishes between two kinds of pattern role, while acknowledging that the distinction is not always clear-cut:

- *Superimposed roles* are assigned to classes that have functionality and responsibility outside the pattern and contain code pertaining to other sets of responsibilities.
- *Defining roles* are completely defined by the pattern, with no functionality outside the context of the pattern.

[1] The material used for this study is available at:
http://ctp.di.fct.unl.pt/~mpm/AOLA/

Results obtained for OT/J are presented next, organized into the various degrees of success in yielding clean pattern modules and what compositions can be performed with those modules. We also comment of language support for specific patterns.

Zero failures to modularize. In HK study, 6 AspectJ pattern implementations (*Façade*, *Abstract Factory*, *Bridge*, *Builder*, *Factory Method* and *Interpreter*) failed to attain code locality, which we consider the minimum requisite for deeming a modularization successful. By contrast, all OT/J implementations achieved code locality and thus all 23 patterns can be considered successful modularizations. This marked superiority of OT/J is primarily due to the ability to package together participant classes as roles within a common team. So even when all else fails, this packaging capability can still yield a successful modularization within a team. Crucially, such team modules can always be further extended non-invasively, through the addition of sub-teams.

Identical to Java. *Singleton* is the sole pattern whose OT/J implementations resulted identical to those in Java. In the case of AspectJ it was *Façade*. The intent of *Singleton* is to ensure a class only has one instance, and provide a global point of access to it [1]. The usual way to implement it in Java is to block access to constructors through non-public visibility and provide an accessor method that always returns the same class instance whenever it is called. The AspectJ implementation of *Singleton* intercepts calls to the constructor and makes it return the same class instance in all cases. In OT/J, roles cannot intercept base constructors and therefore do not provide the means to emulate a similar outcome. However, it can be argued that the singleton property is inherent to a given module, and its Java implementation is already localized within a single class. Thus, the sole pattern in relation to which OT/J does not "improve" on Java can, by coincidence, be still reasonably considered a successful modularization.

Non-reusable modularizations. For OT/J, this group includes 10 patterns: *Adapter*, *Bridge*, *Builder*, *Decorator*, *Façade*, *Interpreter*, *Iterator*, *Proxy*, *State* and *Template Method*. For AspectJ, this group includes 6 patterns: *Adapter*, *Composite*, *Decorator*, *Proxy*, *State* and *Template Method*. In OT/J, failure to reuse these patterns is mainly due to pattern instances being very scenario-specific. In the case of *Adapter*, *Bridge*, *Decorator* and *Proxy*, this is due to their common purpose of adapting a given class. Adaptations of case-specific classes (i.e., *glue code*) are non-reusable by their very nature. *State* is about keeping track of the state of a given object, which is again case-specific.

Reusable modularizations. This group includes 10 patterns: *Chain of Responsibility*, *Command*, *Composite*, *Flyweight*, *Mediator*, *Memento*, *Observer*, *Prototype*, *Strategy* and *Visitor*. AspectJ attains reuse for these 10 patterns, plus *Iterator* and *Singleton*. The general implementation approach, similarly to AspectJ, was to place parts common to different pattern instances in abstract reusable teams, which were concretized by sub-teams for the instance-specific part. For OT/J, this group can in turn be subdivided into (1) those that have only super-imposed roles (*Chain of Responsibility*, *Mediator, Observer* and *Prototype),* and (2) those that include defining roles (*Command, Composite, Flyweight, Memento, Strategy* and *Visitor*). As in the AspectJ study, benefits brought by the new language mechanisms are primarily felt when dealing with superimposed roles. Code associated to such roles can be extracted to roles within teams. However, it is tricky or even non-sensical to attempt a separation of defining roles. The OT/J approach for the second subgroup was to either make the team itself represent the defining role, or to use unbound roles to represent defining pattern roles within a team module. This way, participant classes became clean of pattern-specific code.

Direct Language Support. The HK study includes a group of patterns (*Adapter*, *Decorator*, *Proxy*, *Strategy*, *and Visitor*) whose AspectJ implementations "disappear", because language constructs implement them directly, though with some inherent limitations. Using OT/J, the purposes of *Factory method* and *Abstract Factory* are directly supported by language constructs. The purpose of *Factory Method* is to emulate polymorphic constructors, which is exactly what is obtained from virtual classes [5]. The purpose of *Abstract Factory* is to provide an interface for creating families of related objects and ensure that instances of a given family are created consistently, avoiding undesirable mixing between families. That is exactly the purpose of family polymorphism [1].

5. SUMMARY

In terms of the modularization, reuse and direct language support, there are advantages on both sides and no language emerges as a clear winner overall. However, OT/J has a clear advantage in terms of extensibility and, in general, of what can be done with the resulting modules. In AspectJ, concrete aspects cannot be extended, while OT/J teams are always extensible, though in some specific scenarios the option of extending the team is not applicable.

To sum up the differences between the two languages, AspectJ and OT/J seem geared for different purposes. AspectJ is known to yield very good results when used for applications that perform "highly crosscutting" tasks of the kind provided by profilers, monitoring and instrumentation tools. The fine-grained joinpoint model of AspectJ is suitable for such tasks, which often do not even yield a product to be shipped to clients. However, AspectJ seems unsuitable for the support of large architectures and long-term evolvability. OT/J is the opposite: it seems unsuited for the former but seems very promising for the latter.

6. ACKNOWLEDGEMENTS

This work was partially supported by project AMADEUS (POCTI, PTDC/EIA/ 70271/2006), funded by Portuguese *Fundação para a Ciência e Tecnologia*.

7. REFERENCES

[1] Ernst E. Family polymorphism. ECOOP 2001, Heidelberg, Germany, 2001.

[2] Gamma, E., Helm, R., Johnson, R., Vlissides, J., Design Patterns – Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994.

[3] Hannemann, J., Kiczales, G., Design Pattern implementation in Java and in AspectJ, OOPSLA 2002, Seattle, USA, 2002.

[4] Herrmann S., Hundt C., Mosconi, M. ObjectTeams/Java Language Definition version 1.3 (OTJLD). Technical Report 2009/08, Technische Universität Berlin, 2009.

[5] Madsen O. L., Moller-Pedersen B., Virtual classes: a powerful mechanism in object-oriented programming. OOPSLA'89, New Orleans, Louisiana, USA, 1989.

[6] Rashid, A. and Moreira, A. Domain Models Are NOT Aspect Free. MoDELS 2006, Denver, USA, 2006.