Toward Disposable Domain-Specific Aspect Languages [*]

Arik Hadas[1] David H. Lorenz[1,2 †]

[1]Open University, Raanana 43107, Israel
[2]Technion–Israel Institute of Technology
Haifa 32000, Israel
arik.hadas@openu.ac.il, dhlorenz@cs.technion.ac.il

Abstract

Consider the task of auditing an application whose main functionality is to execute commands received from clients. One could audit command executions with AspectJ. Alternatively, one could design, implement, and use a domain-specific aspect language for auditing, and then throw the language away. In this paper we argue that such disposable aspect languages are useful and that developing them may overall be as cost-effective as using general-purpose aspect languages.

Categories and Subject Descriptors D.2.11 [*Software Engineering*]: Software Architectures—Domain-specific architectures

General Terms Language, Design.

Keywords Aspect-oriented programming (AOP), Domain-specific aspect language (DSAL), General-purpose aspect language (GPAL).

1. Introduction

Aspect-oriented languages can be classified into two categories: *general-purpose aspect languages (GPALs)* and *domain-specific aspect languages (DSALs)*. GPALs are languages that provide as general as possible constructs in order to express crosscutting concerns across problem domains. The most prominent example of a GPAL is AspectJ, which has a large ecosystem and supportive tools. In contrast, DSALs limit the generality of their constructs in order to provide better separation of crosscutting concerns for a particular problem domain. A survey of DSALs can be found in Fabry *et al.* [1].

The viability of the DSALs category seems to have an "upper bound" in terms of generality and a "lower bound" in terms of specificity. On the one hand, DSALs that are too general are usually avoided because their applicability overlaps existing GPALs. On the other hand, DSALs that are too specific are avoided too because, unless they are reusable across applications, they may not justify their development cost.

[*] This research was supported in part by the *Israel Science Foundation (ISF)* under grant No. 1440/14.

[†] Work done in part while visiting the Faculty of Computer Science, Technion—Israel Institute of Technology.

MODULARITY Companion'16, March 14–17, 2016, Málaga, Spain
ACM. 978-1-4503-4033-5/16/03...
http://dx.doi.org/10.1145/2892664.2892676

In this work we look at a special kind of *ad-hock crosscutting concern*s that in principle could be expressed in a GPAL but their implementation typically results in "aspect spaghetti code," i.e., the aspect itself suffers from code scattering and tangling. In order to handle such concerns we identify a subcategory of DSALs called *disposable aspect languages (*DISPAL*s)*. DISPALs are atypical DSALs, challenging both boundaries of the DSAL spectrum. On the one hand, DISPALs can be translated into GPALs, potentially rendering them redundant. On the other hand, DISPALs are extremely specific with a slim expectation for reuse.

Nevertheless, DISPALs present new possibilities. The fact that reuse is not an objective makes DISPALs easier to design than DSALs. The fact that their constructs can be translated into constructs in some GPAL makes them also simpler to implement, sometimes to the extent that they can be built ad-hock and on-demand for a particular application and then disposed of. Considering DSALs as disposable promotes an agile-like Language Oriented Modularity (LOM) [6] process, that may fundamentally change the way we use aspect languages in software projects.

2. Example

Consider a Java application whose main functionality is to receive commands from clients and execute them (the commands, not the clients). A natural design choice for such an application is to use the COMMAND design pattern [2]. Listing 1 shows the `Command` class, root of all commands. The method `execute` is responsible for the core logic of the command. The method `end` is called when asynchronous operations that were initiated in the `execute` method are completed. The method `isSucceeded` returns true if the command succeeded and false otherwise.

A concrete command called `CopyCommand` for copying a resource asynchronously is shown in Listing 2. It overrides the three abstract methods defined in `Command`. The `execute` method locks the resource on the source so that it will not be deleted while it is being copied, and initiates the copy operation. The `end` method unlocks the resource on the source, and if the copy operation succeeds, it adds the resource on the destination to the system. The `isSucceeded` method returns true when the copy operation completes successfully and false when any of the operations in the `execute` method or the copy operation fails.

2.1 Basic Auditing

Now, suppose we are initially asked to audit commands in two places as they are being executed:

- After the execution of the `execute` method, indicating whether the asynchronous operations have started or the command has failed.

Listing 1: Command class

```
public abstract class Command<T extends Parameters> {
 protected T params;
 Command(T params) { this.params = params; }
 abstract void execute();
 abstract void end();
 abstract boolean isSucceeded();
 public T getParameters() { return params; }
}
```

Listing 2: Copy resource command

```
class CopyCommand extends Command<CopyParameters> {
 public CopyCommand(CopyParameters p) { super(p); }
 @Override
 void execute() {
  lock(params.getResource(), params.getSource());
  copy(params.getResource(), params.getSource(), params.
      getDestination());
 }
 @Override
 void end() {
  unlock(params.getResource(), params.getSource());
  if (isSucceeded())
   add(params.getResource(), params.getDestination());
 }
 @Override
 boolean isSucceeded() { ... }
}
```

- After the execution of the end method, indicating whether the asynchronous operations succeeded or failed.

2.2 Enhanced Auditing

As the application evolves additional requests for auditing may be made:

- Ability to audit commands that can be either synchronous or asynchronous.
- Ability to translate audit messages to other languages.
- Ability to include specific information in audit messages. For example, for `CopyCommand` including the name of the copied resource.
- Ability to customize audit messages according to the command's parameters. For example, indicating for `CopyCommand` whether or not the resource was encrypted before it was copied.

3. Disposable DSAL vs. AspectJ

One can implement the auditing in AspectJ. Initially, the aspect is fairly simple. It defines two pieces of advice. The first advice is executed after the `execute` method, fetches the command's type, and produces a message according to the return value of `isSucceeded`. The second advice is executed after the `end` method, fetches the command's type and, again, produces a message according to the return value of `isSucceeded`.

However, with AspectJ the code within the aspect becomes more and more tangled as more requirements are added. Listing 3 sketches an implementation in AspectJ for enhanced auditing. The code becomes cumbersome because more logic is required in order to figure out which message should be produced in each phase. The aspect needs to check whether the command is asynchronous or not (and optionally check its parameters) in order to determine the relevant message. It is also more complicated to produce the message,

Listing 3: Enhanced Auditing in AspectJ

```
public privileged aspect EnhancedAudit {
 after(Command c): execution(* execute()) && this(c) {
  if (c instanceof CopyCommand) {
   CopyCommand copyCmd = (CopyCommand) c;
   CopyParameters params = copyCmd.getParameters();
   if (!copyCmd.isSucceeded()) {
    audit(resolve(AuditMessages.COPY_FAILED, params.
         getResource()));
   } else {
    if (copyCmd.isAsync()) {
     String msg = resolve(copyCmd.encrypt ?
       AuditMessages.COPY_ENCRYPT_STARTED : AuditMessages.
            COPY_STARTED,
       params.getResource(), params.getSource(), params.
            getDestination());
     audit(msg);
    } else {
     audit(resolve(AuditMessages.COPY_SUCCEEDED,
       params.getResource(), params.getSource(), params.
            getDestination()));
 }}}}
 after(Command c): execution(* end()) && this(c) {
  if (c instanceof CopyCommand) {
   CopyCommand copyCmd = (CopyCommand) c;
   CopyParameters params = copyCmd.getParameters();
   if (!copyCmd.isSucceeded()) {
    audit(resolve(AuditMessages.COPY_FAILED, params.
         getResource()));
   } else {
    audit(resolve(AuditMessages.COPY_SUCCEEDED,
      params.getResource(), params.getSource(), params.
           getDestination()));
 }}}
...
}
```

since we need to use translationable enumeration that represents the messages, and to retrieve information from the command that should be included in the message.

Alternatively, one can design an ad-hock DISPAL for the required auditing. Listing 4 depicts a grammar definition for such a DISPAL written in the Xtext language workbench. In this DISPAL the audit messages for each command are declared by a section that begins with the name of the command. The messages that should be produced in different conditions are defined using a case statement. Listing 5 depicts the implementation of enhanced auditing for `CopyCommand` in the produced DISPAL.

The semantics for the DISPAL is provided via transformation of the auditing language into AspectJ. This allows us to leverage the weaving capabilities of the AspectJ compiler instead of developing either a custom standalone compiler or a weaver plugin, thus avoiding relatively complex low-level programming.

4. Discussion

Obviously, the need to define and implement a new language imposes additional cost for using the DISPAL. However, the simplicity of a DISPAL reduces the cost of implementing the language and programming in it. In this section we discuss when one might use DISPALs rather than GPALs or ordinary DSALs.

On the one hand, there are crosscutting concerns that do not rely on the structure or business-logic entities in a particular application. These crosscutting concerns are likely to appear in various applications and therefore their solution can be reused *across applications*. When the solution is easy to express with a general-

Listing 4: DAL grammar definition in Xtext

```
Model: commands+=Command(',' commands+=Command)*;
Command:
  type=[types::JvmDeclaredType|QualifiedName] ':'
    (cases+=Case(',' cases+=Case)*)?
  ';'
;
Case:
  'case' actionState=ActionState ('&' (fields+=[types::
        JvmField]))*
  'log' '(' msg=[types::JvmEnumerationLiteral] (',' ops+=[types::
        JvmOperation])* ')'
;
enum ActionState: started|success|failure;
QualifiedName: ID ("." ID)*;
```

Listing 5: Auditing aspect in the DAL

```
CopyCommand:
  case failure
    log(COPY_FAILED, getResource),
  case started & encrypt
    log(COPY_ENCRYPT_STARTED, getResource, getSource,
          getDestination),
  case started
    log(COPY_STARTED, getResource, getSource,
          getDestination),
  case success
    log(COPY_SUCCEEDED, getResource, getSource,
          getDestination)
;
```

purpose language, GPAL would be the best choice. For example, concerns like measuring the execution time of selected methods, or executing selected methods in a transaction, could be solved by GPALs, since generally one could find a third-party aspect that does the job or implement an aspect that is not likely to change as the application evolves. When the solution is difficult to express with a general-purpose language, it might be better to develop a DSAL for it. Hopefully, the reusability of the language across applications would justify the cost of developing that DSAL. An example of this is synchronization of method executions, which is easier to express using a DSAL like COOL.

On the other hand, there are crosscutting concerns that are highly related to the structure and business logic entities in a particular application. These concerns are likely to get more and more complicated as the application evolves, and are less likely to be reused across applications. An example of such a concern is the auditing concern described in Section 2. The requirements are highly affected by the structure of the commands and their use. As we have demonstrated, these requirements are likely to get more complicated as the application evolves. The benefit of using DISPALs for such concerns is that they provide a simpler and more restrictive grammar for the given problem and therefore are easier to program with compared to GPALs. In addition, the cost of developing a DISPAL is less than that of a DSAL, since DISPALs are more specific and thus easier to define, implement, and maintain for the particular problem.

One may argue that the development and maintenance of a transformation from a DISPAL to a GPAL is as hard as (and even harder than) programming in that GPAL. Indeed, in order to define the transformation of the DISPAL to a GPAL we used the following method. First, we wrote an aspect in the GPAL that solves a special case of the crosscutting concern in question. Then, we generalized that aspect by defining a transformation from the DISPAL to that GPAL for the general case. Thus, this argument may be true for the language design effort. However, it is not necessarily true for the overall development cost, taking into account that in a typical project DISPAL design is done by few developers, while all other developers enjoy the benefit of programming in these DISPALs.

The cost of developing DISPALs can still seem to be too high. Our experience with the oVirt project, however, is that the development of DISPALs was relatively simple. First, the fact that the language is tailored to a specific problem means that a limited number of constructs needs to be defined. Second, it means that we can rely on the structure of the application in question, which generally simplifies pointcut definitions within aspects. Third, the availability of language workbenches that facilitate both domain-specific grammar definition and its transformation into a general-purpose language greatly reduces the development cost. The transformations of the DISPALs that were developed for the crosscutting concerns identified in the oVirt project are available at `github.com/OpenUniversity`. The application of the approach to crosscutting concerns found in other open source projects is left for future work.

5. Conclusions

While GPALs typically improve the modularization of the software, the logic within the aspect code remains complex and costly to develop and maintain. Ad-hock DISPALs optimized for the problem at hand offer a declarative and more effective alternative. The relative ease of implementing DISPALs with modern development tools like aspect language workbenches [3–5] makes even one-time use of DISPALs cost-effective.

More broadly, thinking of aspect languages as disposable brings about a more agile-like process in designing and using DSALs. It might also suggest that future research should focus on making GPALs more expressive (by exposing more join points, for example) rather than making DSALs more reusable.

References

[1] J. Fabry, T. Dinkelaker, J. Noyé, and E. Tanter. A taxonomy of domain-specific aspect languages. *ACM Computing Surveys (CSUR)*, 47(3), Apr. 2015.

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

[3] A. Hadas and D. H. Lorenz. Demanding first-class equality for domain specific aspect languages. In *Modularity'15 Comp.*, pages 35–38, Fort Collins, CO, USA, Mar. 2015. ACM Press. Position paper.

[4] A. Hadas and D. H. Lorenz. First class domain specific aspect languages. In *Modularity'15 Comp.*, pages 29–30, Fort Collins, CO, USA, Mar. 2015. ACM Press. Poster Session.

[5] A. Hadas and D. H. Lorenz. A language workbench for implementing your favorite extension to AspectJ. In *Modularity'15 Comp.*, pages 19–20, Fort Collins, CO, USA, Mar. 2015. ACM Press. Demo Session.

[6] D. H. Lorenz. Language-oriented modularity through Awesome DSALs: summary of invited talk. In *Proceedings of the 7th AOSD Workshop on Domain-Specific Aspects Languages (DSAL'12)*, pages 1–2, Potsdam, Germany, Mar. 2012. ACM Press.