Experiences in modularizing business rules into aspects

Andy Kellens, Kris De Schutter, Theo D'Hondt
Programming Technology Lab
Vrije Universiteit Brussel
{ akellens | kdeschut | tjdhondt}@vub.ac.be

Viviane Jonckers
System and Software Engineering Lab
Vrije Universiteit Brussel
vejoncke@ssel.vub.ac.be

Hans Doggen
inno.com
Beerzel, Belgium
hans.doggen@inno.com

Abstract

This paper provides an experience report on the use of aspect-oriented technology as a means to modularize the implementation of business rules in an object-oriented, large scale case study. The goal of this refactoring of the system was to provide a proof-of-concept implementation of how such an aspect-oriented solution can improve the modularity and the extensibility of the business rule implementation. This paper focusses on the approach taken in refactoring the system and the difficulties of integrating the aspect solution into the build process.

1 Introduction

Aspect-oriented programming has been advanced as a novel technique to modularize crosscutting concerns, i.e. concerns that are spread across the implementation of multiple modules. Over the recent years, we have seen an adoption of this technique by industry [4, 5] as a means to separate mostly non-functional concerns (such as logging, monitoring, ...) from the base code of a system, with the aim of increasing evolvability and maintainability by providing a better separation of concerns.

In this paper we relate our experiences in applying aspect technology to an industrial system as a means to modularize the implementation of business rules (a *functional* concern) in an object-oriented customer information system. While — as discussed by Cibran et al. [1] — the modularization of business rules can benefit from the use of aspects, the goal of this work is to demonstrate the feasibility of migrating business rules in an existing, large-scale system.

This experience report focusses on two aspects of the refactoring process. First, we relate how the business rules were implemented in the original system and how, by using aspect technology, we were able to refactor them in a separate module. Second, we discuss a major technical challenge we had to tackle in order to perform the refactoring, namely the integration of the aspect-oriented solution into the build process of the system.

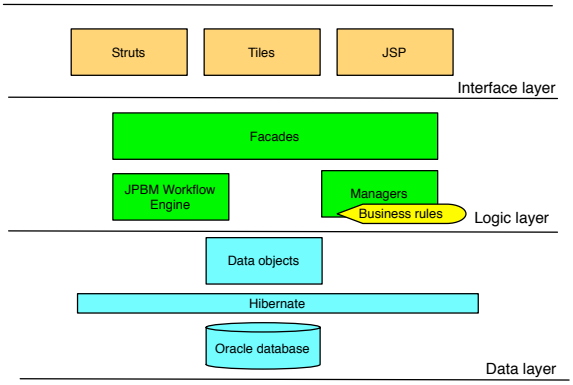

Figure 1. Overview of the architecture of the system

2 Business rules in the case study

The system under consideration is a customer information system developed by a Belgian company (+ 30.000 employees) and assisted by the Flemish company inno.com. It is a fairly large J2EE application (305 KLOC) consisting of 2602 classes with a total of 26968 methods. The system is designed following a 3-tier architecture as depicted in Figure 1 and makes use of off-the-shelf technology such as Struts, Tiles and JSP to render the user interface, the JBPM workflow engine in the logic layer to steer the business flow

```
class DAOManager {
  public void saveDAO(DAO dao) {
    validateDAO(dao);
    dao.save(); }
  public void updateDAO(DAO dao) {
    validateDAO(dao);
    dao.update(); }

  public void validateDAO(DAO dao)
  throws BusinessException{
    try {
      validateBR1(dao);
      validateBR2(dao);
    } catch (UncaughtBusinessException exception) {
      /* ... */
    }}
  private void validateBR1(DAO dao)
  throws BusinessException {
    /* check rule 1 */ }
  private void validateBR2(DAO dao)
  throws BusinessException {
    /* check rule 2 */ }}
```

Figure 2. Implementation of the business rules

in the application, and Hibernate as a means to enable persistence of the data objects in the data layer. Aside from the use of a workflow engine in the logic layer, the business logic of the system is implemented by a number of Facade and Manager classes that coordinate the various operations on data objects.

As mentioned above, it is our intent to refactor a part of the business rules present in the system. These business rules are situated in the Manager classes, together with that part of the core logic that implements operations on data objects. In order to implement the business rules, the original developers of the system consistently followed a particular implementation idiom. We will illustrate this idiom by means of an example[1] (see Figure 2). For a particular data access object (DAO), a separate Manager class (`DAOManager`) is provided that implements the operations for this DAO. For example, in Figure 2 two operations `saveDAO` and `updateDAO` are shown that apply to a DAO. Aside from performing the operation, these methods also invoke a `validateDAO` method that verifies the business rules applicable to the data object. For each business rule, a separate method is provided implementing that business rule. These methods then get invoked from within the `validateDAO` method, together with code that performs the exception handling in case a rule is violated.

Although the current implementation rigourously follows the above idiom, the lack of a clear modularization of the business rules does introduce a number of disadvantages with respect to evolvability and extensibility of the set of business rules. Since the business rules are spread throughout the system and *tangled* with the Manager classes, adapting or extending the set of business rules cannot be performed in isolation of the other concerns in the system. Moreover, since the business rules are tangled with the operations on DAOs, this needlessly complicates the source code of the Manager classes. Furthermore, we encountered some cases of *code duplication* that resulted from the lack of an explicit modularization of the business rules: in some situations a particular business rule was applicable to multiple DAOs, which resulted in duplication of that rule in all affected Manager classes. A second source of code duplication was the exception handling mechanism to deal with violations of business rules, which is almost identical across all Manager classes. Finally, the Manager classes exhibit some degree of *scattering*: the `validateDAO` method has to be invoked from within multiple operations on the DAO.

[1] Due to a non-disclosure agreement, we are not allowed to show an actual example from the source code. Therefore, we present an abstract example of the applied idiom.

```
aspect DAOBusinessRules {
  pointcut businessOperation(DAO dao):
    (execution(* DAOManager.saveDAO(DAO)) ||
     execution(* DAOManager.updateDAO(DAO)))
    && args(dao);

    before(DAO dao) throws UncaughtBusinessException:
      businessOperation(dao) {
        /* check rule 1 */    }
    before(DAO dao) throws UncaughtBusinessException:
      businessOperation(dao){
      /* check rule 2 */    }}
```

Figure 3. Implementation of the business rules using aspects

3 Modularizing the business rules with aspects

As a proof-of-concept to illustrate the use of aspects to express the business rules in the customer information system, we refactored a subset of the business rules using the AspectJ language [2]. To this end, we implemented a hierarchy of aspects — parallel to the hierarchy of data access objects — in which we separated the business rules from the core logic of the system. An illustration of this refactoring, applied to the example above can be found in Figure 3. For each DAO we create a separate aspect implementing that DAO's business rules. In this aspect, a pointcut `businessOperation` is specified that enumerates all the places in the execution of the core logic at which the business rules need to be verified. In our example, this pointcut should intercept the execution of the methods `saveDAO` and `updateDAO`. Each business rule applicable to the DAO is implemented by a separate *before* advice that is triggered before the actual execution of an operation. Aside from modularizing the business rules, we also extracted the exception handling mechanism that is associated with the verification of business rules from the Manager classes and implemented this mechanism using aspects.

After this refactoring, we were able to completely remove the source code pertaining to the business rules from the Manager classes. The source code for the `DAOManager` class in our example now becomes:

```
class DAOManager {
  public void saveDAO(DAO dao) { dao.save(); }
  public void updateDAO(DAO dao) { dao.update(); }}
```

From this class, all methods implementing business rules as well as the exception handling mechanism associated with the business rules are removed, leaving only the operations belonging to the core logic present in the Manager class.

4 Evaluation of the solution

Our refactored solution offers the advantage that a separation of core logic and business rules is obtained. Consequently, the tangling of the business rules with the core logic is eliminated from the system. Moreover, rather than being scattered throughout the Manager classes, the business rules implementation is made explicit and located in a single module. As a result, adding or manipulating business rules is limited to investigating and editing the aspects that implement these business rules, instead of manipulating the Manager classes. For example, the addition of a new business rule for a DAO can be achieved by adding a new *before* advice to the corresponding business rule aspect. Furthermore, the use of the pointcut mechanism to capture the invocations of the business rules removes the scattered calls to the validation methods from within the Manager classes.

Our refactoring also removed the code duplication resulting from the business rule implementation. First, the exception handling code — which was similar for all business rules — was extracted as a separate aspect. Second, business rules that are applicable to multiple DAOs, and which resulted in code duplication in the original application, are now only present in one aspect. Remember that we represent the business rules as a *hierarchy* of aspects: if a business rule is applicable to multiple DAOs, it is implemented higher in the hierarchy.

We were greatly aided in performing the above refactoring by the fact that the discussed implementation idiom is used *rigourously* throughout the implementation. Consequently, this makes translating the original semantics of the system relatively straight-forward into an aspect-oriented solution without having to restructure the code significantly. As future work, we have yet to investigate to which degree this refactoring process may even be automated.

5 Integration within the build process

Aside from performing the refactoring of the source code of the system, we also had to incorporate the business rule aspects in the build process of the otherwise standard object-oriented system. In doing so we want to retain as many of the qualities of the original build system while adding one of our own: the weaving of aspects should introduce as little runtime overhead as possible. This means tackling the weaving within the build system itself, rather than relegating it until runtime.

In the case study presented here, the build system consists of an ANT script making use of the basic ANT tasks, and augmented with several macros which handle recurring patterns. The structure within intimately reflects that of the application's architecture: all modules are compiled separately, each placed in a jar of its own, all of which are then placed inside the final web-archive along with any resources they need. This allows for separate compilation of modules, reducing developer overhead as they are consigned to a single module. It also makes for a very straightforward and very clean build structure, which helps maintenance and evolution thereof.

We will now discuss the three different possible scenarios for integrating aspects into this build system we investigated, arguing their advantages and disadvantages.

5.1 Aspects as just another module

The first approach was to take the existing structure and simply continue in the same vein. As the build system is based around separate compilation of modules, we chose to treat our aspects as just another module. In order to get this working we needed to make the following changes:

- Replace the default compiler with the AspectJ compiler. AspectJ provides an ANT task for doing this. As the AspectJ compiler accepts all legal Java programs this in itself has no further consequences on the final result.
- Compile the aspects separately, and place them in a jar of their own. AspectJ allows separate compilation of aspects without having to instantly weave them in the application.
- Compile all other modules, telling the AspectJ weaver to apply the previously compiled aspects. Each module will then get the aspects applied. Further processing for each module remains as before.
- Add the aspects jar to the web-archive, together with the AspectJ runtime library.

This approach has the advantage of having only a small impact on the structure of the build system, maintaining its modular design and retaining the advantage of separate compilation of modules. Of course, modification of the aspects will entail a complete recompilation of all modules, but for the module developers this should happen only infrequently.

However due to strict ordering needed, *first* compile the aspects, *then* compile/weave the modules, this meant that we were not able to compile the system. The business rule aspects explicitly reference classes present in the modules. This means that they can *not* be compiled *before* these modules. The modules should already be compiled, or be compiled along with the aspects. As our aspects modify the modules, the first is no option either, and so the latter approach becomes necessary. Consequently, we had to investigate other means to incorporate our aspects into the build.

5.2 All-in-one compilation/weaving

The second approach was to take all modules along with the aspects and compile/weave them in one go. While solving the inter-dependency problem between our aspects and the modules, this comes at a great cost:

- We need to rewrite the entire build system in order to obtain a single compilation step. This means that all modules/aspects are copied to a single location where they get compiled, thus breaking the modular structure of the build file.
- A minor side effect of this single compilation step is that it becomes hard to separate the different modules again after compilation. Whereas before we ended up with a single jar per module, now we have one monolithic jar instead.
- The major side effect is that we lose separate compilation. Any modification to any part of the system now entails a recompilation of the entire system, and this quickly becomes tedious and costly, resulting in a significant overhead on the development process.

5.3 Middle ground

In the end, we preferred to trade some of the aspects' modularity in for separate compilation of the modules. We chose to consider the aspects to be part of the modules they affect rather than belonging to a module of their own. Note that this also limits the scope of the aspects to the module in which they are placed. By applying these restrictions we obtained a situation where each module can again be compiled separately, where compilation of a module is an all-in-one compilation of all aspects and classes in that module.

The obvious disadvantage here is that we can no longer have aspects which cross the boundaries of modules. In the case of the business rule aspects in this paper, however, this poses no problem, as these rules are indeed tied to a single module. If module boundaries have to be crossed, we imagine the following two workarounds:

- If the aspects are truly generic —that is, they are not tied to any of the application's types or classes— then we can apply them using the solution from section 5.1.
- If they do not exhibit this genericity then the workaround is to duplicate the aspects in each module. This may be done through a simple copy operation during the build so as to circumvent source code duplication. Communication among the different copies will then have to be implemented manually, e.g. through a mediating class. This is similar to what Nordberg proposes in [3] as a way to mix components and aspects.

Since we had to make a trade-off between build system modularity and build time, this is again not an ideal solution. The problem we encountered seems to indicate that in the presence of aspects, current build systems no longer can express and check the source code dependencies between various modules, and might indicate the need for more advanced, aspect-aware build systems.

6 Conclusion

In this paper we have discussed the refactoring of the business rules implementation in an object-oriented system to an aspect-oriented solution. At the level of the aspect-oriented solution itself we were successful in migrating the crosscutting business rules into aspects, facilitated by the rigourous use of a particular pattern for implementing business rules in the original system. However, the integration of the aspects into the build process of the system proved to be less trivial due to the lack of proper support for aspect modules in current-day build systems. We discussed three alternative solutions to this problem and their impact on the development process.

Acknowledgments — Andy Kellens is funded by a research grant provided by the "Institute for the Promotion of Innovation through Science and Technology in Flanders" (IWT Vlaanderen). Kris De Schutter receives support from the research project AspectLab, also sponsored by the IWT Vlaanderen.

References

[1] M. Cibran and M. D'Hondt. A slice of mde with aop: Transforming high-level business rules to aspects. In *9th International Conference on MoDELS/UML*, pages 170–184, 2006.

[2] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proc. ECOOP 2001, LNCS 2072*, pages 327–353. Springer-Verlag, June 2001.

[3] M. Nordberg. Aspect-oriented dependency inversion. In *Workshop on Advanced Separation of Concerns in Object-Oriented Systems (OOPSLA 2001)*, Oct. 2001.

[4] T. Pijpops and J. Van Reusel. Improving the design of a large jave EE application with AOP. In *Proceedings of the 7th International Conference on Aspect-Oriented Software Development (Industry track)*, pages 45–47, 2008.

[5] D. Shepherd, T. Roper, and L. Pollock. Using AOP to ease evolution. In *ICSM (Industrial and Tool Volume)*, pages 16–25, 2005.