Reducing Static Dependences Exploiting a Declarative Design Patterns Framework

Mario Luca Bernardi[1] and Marta Cimitile[2]
[1]*Giustino Fortunato University, Benevento, Italy*
[2]*Unitelma Sapienza University, Rome, Italy*

Keywords: Software Engineering, Design Patterns, Aspect Oriented Software Development, Software Metrics.

Abstract: Object Oriented Design Patterns (DPs) are recurring solutions to common problems in software design aiming to improve code reusability, maintainability and comprehensibility. Despite such advantages, the adoption of DPs causes the presence of crosscutting code increasing, significantly, the code duplication and the dependencies between systems. The main idea of this research is that code crosscutting can be reduced by the integration of Model Driven Development (MDD) techniques with Aspect Oriented Programming (AOP). According to this, an approach based on a Domain Specification Language (DSL) to to define declaratively the structure of DPs and their adoption on classes to declarative, is proposed. The approach aims to support aspects derivation to compose, at run time, AOP-based version of the specified DPs. The approach has been applied in a case study where the developed supporting framework was used in a concrete refactoring scenario, and a subsequent maintenance task. The results from the case study are presented and discussed.

1 INTRODUCTION

DPs are general repeatable solutions to recurring problems of software implementation improving effectiveness and efficiency in software development. Despite such advantages, the usage of DPs have also some disadvantages like an increasing number of static dependencies among components, an higher code crosscutting (this should introduce some defects in the system) and in some case greater code duplication (Hannemann and Kiczales, 2002). More in general, the adoption of multiple DPs may have an impact on the system modularity affecting its maintainability, comprehensibility and testability (Hannemann and Kiczales, 2002; Bernardi et al., 2016). Aspect-Oriented Software Development (AOSD) is a software development technology that achieves to improve the modularity of the developed software (Hannemann, 2001) giving more emphasis to the separation of crosscutting concerns (CCs). Consequently, AOSD represents a valid solution to reducing crosscutting resulting from DPs adoption. It performs an advanced separation of concerns and encapsulates CCs in separate modules, called aspects. Aspects are composed with other software artifacts through powerful weaving mechanisms enforcing transparency, optionality, and unpluggability modularity (Hannemann and Kiczales, 2002). Moreover, software development can be also improved by the adoption of Model Driven techniques achieving to increase the overall system modularity of design models, source code, structure and behaviour of run-time objects (Bernardi et al., 2013a). Model Driven Software Development (MDSD) models all the relevant system knowledge and the obtained models are used in formal transformations to produce all needed artifacts (i.e., source code and documentation). These models are also synchronized, combined, and transformed across different levels of abstraction introducing some meta models that characterize a set of models as their instances and are instantiated using a DSL to describe their abstract syntax and to generate a model. This paper proposes an approach based on a MDSD techniques and an Aspect Oriented DSL-based framework to define declaratively the structure of DPs improving modularity, dynamic behaviour and obliviousness. Moreover, the proposed approach allows to improve flexibility permitting to adopt different pattern variants with limited impact on system source code and achieves a greater internal cohesion leaving concrete system classes as decoupled as possible from the components of the DPs code. Another advantage of the proposed approach is that it is completely declarative. This permits to apply any change

Bernardi, M. and Cimitile, M.
Reducing Static Dependences Exploiting a Declarative Design Patterns Framework.
DOI: 10.5220/0006009401540160
In *Proceedings of the 11th International Joint Conference on Software Technologies (ICSOFT 2016) - Volume 2: ICSOFT-PT*, pages 154-160
ISBN: 978-989-758-194-6

to the DP implementations and their corresponding concrete classes changing only the DSL specifications without any direct impact on the system source code. The approach was implemented using a framework, called Declarative Design Pattern Framework (DDPF), built on the Eclipse Modeling tools using Xpand as Model to Text (M2T) transformation engine and AspectJ as AOP language. The generation step is performed starting from a model written by the DSL and dynamically generates aspects applying idioms and DPs on system classes with null (or very reduced) impact on them. A template-language engine is used to parse the DSL DSL code generating some Java and AspectJ resources to implement flexible and modular DPs using the concrete classes of the system. To validate the effectiveness of the proposed approach a real world system, the Korsakow editor system, is refactored using DDPF and both DDPF and OOP versions are analyzed using modualrity metrics (DOS and DOF defined in (Eaddy et al., 2007)). The comparison was performed evaluating the quality of modularization and the impact of required changes on system classes involving DPs. This in order to assess the the framework adoption in terms of: (i) DSL effectiveness to express (and change at run-time) design choices, and (ii) internal structure quality of the resulting system source code.

The remaining of the paper is structured as follows. Section 2 reports some related work. Section 3 presents and discusses the proposed approach and the architecture of the AspectJ-based framework adopting it. The section 4 reports a description of the the case study and discusses the quantitative evaluation of the proposed framework using an adequate set of AOP-aware source code metrics. Conclusive remarks and future works are finally presented in section 5.

2 RELATED WORK

Several DPs description languages and DPs implementation approaches are proposed. In (Eden, 2001), a formal language for DPs representation is reported. It allows the designer to refine and customize the descriptions of canonical DPs or to define new DPs from scratch. Another pattern language is also proposed in (Zdun, 2004). It allows to trace and manipulate software components and their dependencies. In (Elaasar et al., 2006), the DPs are applied to concrete classes using pattern languages or models. Moreover, in (Boussaidi and Mili, 2007) authors propose an explicit DPs representation as well as the transformation embodied in their application.

Several studies are also focused on the adoption of aspect oriented techniques to implement object-oriented DPs: in (Baca and Vranic, 2011) intrinsic aspect-oriented DPs are used to improve composability with respect to the original implementations of object-oriented DPs and their aspect-oriented re-implementations. In (Soundarajan and Hallstrom, 2004), aspect oriented design language is used to support software reuse deriving the specific requirements for the Aspect Oriented Software Development Design Language architecture by examining the AspectJ extensions for a distributed computing environment.

Respect to the above discussed approaches (that apply DPs to the existing design or code), in this paper we propose a DSL approach aiming to reduce crosscutting introduced by DPs in Object Oriented code (Bernardi et al., 2013c; Bernardi et al., 2013b; Bernardi et al., 2014). Moreover, an approach for support application that uses DPs into an aspect-based version refactoring is proposed in (Giunta et al., 2012). Here, with respect to our declarative approach, a lower flexibility in modifying patterns application is obtained thought an explicit refactoring of existing source code. Differently from this, in the proposed approach the system is developed from scratch using a declarative language that is more suitable in a forward engineering context. Finally, the proposed approach differs from the other ones for the definition of a dynamic DSL that involves, in a completely dynamic manner, the existing source code in a pattern logic. According to this, the proposed engine in the place to modify or generate code, generates aspects that inject pattern logic at run-time while system classes are oblivious. Since only aspects perform interception and a run-time bytecode manipulation is performed to apply pattern logic, an higher flexibility, a lower invasivity and a reduction of the maintenance effort is obtained.

3 PROPOSED APPROACH

The proposed approach is focused on the definition of a DSL based on an OO system meta-model and allowing to map DPs (along with their variants, roles and default implementations) on the system classes. The DSL is composed by two main parts: i) the representation of the source code elements and ii) the representation of all the elements (pattern, role definition and implementation) aiming to perform dynamic interception of a wide set of events that are used to involve, at run-time, the concrete classes in pattern collaboration. The source code elements are modeled using the programming language syntax of the implemented system. In particular, the proposed DSL is

implemented considering Java as the target programming language and it is defined according to the abstract syntax of the Java language specification. The system elements are represented by 113 meta-classes and the relationships among them. The system element meta model also include generic types, statements, blocks, exceptions and expressions.

In (Bernardi et al., 2014) the core excerpt of the second part of the DSL meta-model is shown. It describes the overall structure of the DSL and focuses on the main concepts used to implement idioms and DPs (leaving base system classes oblivious and decoupled from pattern logic). For each idiom, or DP, the framework performs a mix of crosscutting injection, alterations and introductions on the system classes. In the following, there are some simple code examples describing the most important elements in the meta-model using the proposed DSL.

The Pattern Declaration Element. Each pattern declaration can be seen as a layer containing only the logic related to its goals and responsibilities. Patterns are merged with the base system (and to other layers) using AOP injection and interception features. The ModelRoot (that is the root element of any DSL instance) contains an ordered list of patterns. In the following example two patterns (a Decorator and an Observer) are defined:

```
order SimpleObserver, *
pattern DrawablesDecorator { ...
pattern DefaultSubject { ...
pattern SimpleObserver { ...
```

The ordering is fixed (using the composition order statement) so that Identification is always merged before any other pattern. This is important when some alterations to the base system are not optional (i.e., the case of dependencies that require a role to be granted in order to apply a pattern definition).

Define, Implement Pattern and Role(s) Elements. The DSL statement *define role* allows a developer to introduce a role in the system, while the statement *implement pattern* is used to assign a pattern to a set of concrete classes. The block *implement role as* allows to define a named implementation of a role that can be used for subsequent registration using the framework static entry point. In the following example, a CacheStrategy pattern, made of two roles (i.e. CacheKeyStrategy and CachingContext), is declared:

```
1  pattern CacheStrategy<T> {
2   define role CacheKeyStrategy {
3      public String getCacheKey(
4      ResourceURI uri,
5      Map<String,Object> params);
6   }
7   define role  CachingContext<T> {
8     CacheStrategy<T> cacheKeyStrategy
       ;
9    <T> request(ResourceURI uri);
10  }
11  implement role CacheKeyStrategy<
       Figure>
12     As FigureCacheStrategy { ... }
13  implement role CachingContext<
       Figure>
14     As FigureCachingContext { ... }
15  ...
16  }
17  FigureManager fm  = new
       FigureManger();
18  DDPF.grant(FigureCacheStrategy.
       class, fm,Figure.class);
19  DDPF.revoke(FigureCacheStrategy.
       class,fm);
```

There are two ways to apply a pattern. The first one exploits a static entry point of DDPF framework. It is shown in line 18 of the above DSL excerpt: in this case the DDPF framework is requested to bind the pattern to a single instance. This approach allows to apply the pattern directly to a set of objects in order to involve them, dynamically, in a patterns collaboration. When such collaboration is completed and the instances do not need anymore to be linked together, it can be destroyed by revoking pattern adoption as shown in line 19. The other way requires the usage of dynamic inheritance by implementing, behind the scenes, the ExtensionObjectsPattern (Gamma et al., 1995) at class load time. As a consequence, all the classes inheriting or implementing them will be forced to provide this additional behavior (the strategy context and the actual caching key strategy). The DSL language construct allowing to do this is the *implement role on* statement. It allows to specify the implementation of a role that must be supplied to a set of existing classes (the role implementation is hence anonymous). The syntax is based upon the InjectionRule nested element that must be followed by the source code of the role implementation. This can be an existing concrete type (in this case partial override of the concrete type is allowed) or a definition from scratch. In both cases, the provided members should create no conflict with the target types members (including the ones that are weaved from other pattern elements). In the next example, the *inject* statement is used to provide an implementation (defined from

scratch) of the CachingContext role introduced in the previous example and to apply it to the a FigureManager (existing) system class:

```
implement pattern CacheStrategy<
    Figure> On FigureManager {
    inject role CachingContext {
      Figure request(ResourceURI
    uri){ ... }
  }
}
```

These classes have no references to the pattern CachingContext implementation and have no imperative dependencies on Identifiable interface (since in this example the Identification pattern is completely orthogonal). However, studies that try to quantify the crosscutting present in real systems, show that most of the patterns are crosscutting and hence they depend on each other. The *implement role* statement just allows the definition of shared fields and methods to express such an interleaving.

Shared Field and Shared Method Elements. When a role is implemented for a set of concrete classes, one or more methods/fields can be provided to link the logic of concrete classes to the new behaviour. In the following DSL example, the UUID can be used to build the name of AbstractFigure, as shown in the following example, where a shared method is nested in the role implementation statement (a similar way can be used for a shared field):

```
inject role Identifiable
    on AbstractFigure+ {
  UUID _uuid;
    public UUID getUUID() { ... }
    public void setUUID(UUID u) { ...
      }
    @Override
    public String getName() {
     return super.getName() + "with
    UUID:"+_uuid.toString();
    }
  }
```

This DSL excerpt allows to inject the getName(void) method in the complete hierarchy rooted in the class AbstractFigure as a part of an Identifiable role, making it dependent on a Named role providing the getName() behavior. They could be both provided by a single pattern definition, but this is not required. Moreover, different methods can be specified for different sub-classes of AbstractFigure, thus reusing the common behaviors as much as possible without loosing flexibility.

3.1 Specifying DPs by the DSL

DPs can be defined by using the DSL elements shown in the previous examples. The following example shows an excerpt of the defined DSL to implement a modular Observer over Figure and View classes:

```
pattern Observer {
  define role Subject {
    void addSubject(Listener l);
    void removeSubject(Listener l);
    void notify();
  }
  define role Listener {
    void update(Subject s);
  }

  implement role Subject On Figure+
    {
    List<Listener> list = new
    ArrayList();
    void addSubject(Listener l){
      list.add(l);
    }
    void removeSubject(Listener l){
      list.remove(l);
    }
    void notify(){
      for (Listener l : list) l.
    update(this);
    }
  }
  implement role Listener On View+
    {
    void update(Subject s){
      target.refresh();
    }
  }
  @Override
  void Figure.setName(String name){
    target.setName(name);
    notify();
  }
}
```

In this example the two observer roles (Subject and Listener) are defined and injected respectively in all Figure and View hierarchies using the *implement role on* statement. The methods defined within role implementation can exploits the **target** reference in order to access the public and protected interface of the type the role is bound to (Figure and View in the example). To support and evaluate the approach, the DDPF was developed. In the prototype, aspects are generated in order to inject members implementing the pattern logic into marker interfaces nested in the aspect itself. The pattern roles are often associated to concrete classes by means of the declare parent construct using such marker interfaces. Each pattern element can

be seen as the intermediate mapping layer of a three layers structure in which concrete system classes are involved in pattern relationships by an aspect that acts as “pattern mapper”. This layer is responsible of implementing a dynamic mapping of DPs and idioms to concrete classes intercepting object creation and enforcing the pattern protocol for instances that need it. Concrete classes, belonging to the ”base system” layer, are oblivious of being involved in a pattern and the pattern relationships can be removed simply acting on the mapping layer. Commonalities among different pattern instances can be factorized in the pattern logic pattern, while multiple relationships can be easily resolved in the pattern mapping layer by associating two aspects to the same concrete class.

4 CASE STUDY

To assess the effectiveness and the validity of the approach, a real world Java system, the Korsakow editor [1], was redesigned by adopting the proposed framework. The assessment of the DDPF approach takes into account the quality of the modularization in terms of pattern logic decoupling from concrete classes. To evaluate the improvement in the quality of the modularization, a crosscutting comparison between two versions of the system (original and refactored with ddpf), has been performed. In order to quantify scattering and tangling of DPs code within system classes, DOF and DOS metrics (Eaddy et al., 2007) are used; this allowed to compare crosscutting across two different systems. Moreover, in order to evaluate the impact of a requirement change, a composite Macro Command was introduced mixing Command pattern hierarchy implementation with a Composite pattern. Several metrics are evaluated (the number of #changed LOC and modules and DOS/DOF variation).

4.1 Command Pattern Analysis

The analyzed system is editor to build layout supporting several languages. It has a big command hierarchy interleaved with several other concerns (persistence, parsing, text manipulation just to name few).

The Korsakow command hierarchy is rooted in a ICommand interface. This interface can be refactored easily as the DDPF role and this allow to remove several LOC from the entire hierarchy as shown in the quantitative analyses. Each editor instance is associated to a set of Commands interleaved by a Factory and exposed to the IDE by means of an action framework. As shown in the figure, the command processors are created by the main instance and follows the request/response model. Each factory is also responsible to inject its context object to the set of its commands. Most commands work on document model (not shown in the figure). Document model is centered on composites, as can be simple items (of several kind of items, e.g. Keyword, Media, Pattern, Predicate, Resource, Text, Sound) associated to a single file. The OOP version of the Command pattern is heavily interleaved with several secondary concerns as stated by requirements. In each Command there are explicit fragments linking it to security, logging and tracing concerns (increasing scattering and tangling). In the DDPF version the entire command hierarchy has been implemented using role assignment and role implementation constructs. This implements the ExtensionObjectsPattern inside an abstract aspect and adds the ”Command” role extension to all the class hierarchy. Analyzing the system code, we found that, with respect to the plain OOP version, DDPF with the adoption of ExtensionObjectsPattern allowed to modularize several concerns (making system classes not aware of being linked to the pattern) with comparable level to the marker interface implementation having the additional advantage of being totally dynamic. In the refactored system every command class can become a command associating it a specific logic that exploits its internal state.

4.2 Discussion

The modulariry assessment for DDPF and OOP versions has been perfomed by calculating: (*i*) the ratio of lines of source code related to DP code in each class or module with respect to the total LOC of the same class or module, and (*ii*) the Degree of Scattering (DOS) and Degree of Focus (DOF) metrics (Eaddy et al., 2007), for each module and pattern.The section also provides quantitative information about: (i) size and dimension ; (ii) code clones and (iii) pattern code presence and distribution.

Results are reported in figure 1. In particular, in Figure 1 the LOC ratio (%LOC) of each design pattern concern over the cloned LOC ratio are compared for each module in both DDPF (AOP) and OOP versions for the command hierarchy. The results show that the ratio of the average cloned LOCs in the OOP implementation is significantly greater than the DDPF system. In this case the DDPF usage improved the maintainability decreasing the probability of introducing bugs during maintenance. This is confirmed by looking at the Degree of Focus (DOF) of the modules reported (only for Items implementing Compos-

[1] https://github.com/korsakow/korsakow-editor

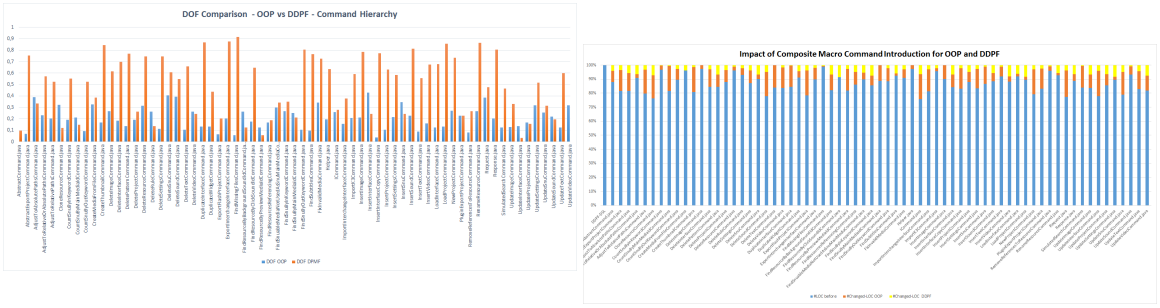

Figure 1: LOC/cloned-LOC ratio by module for OOP and DDPF (left) - Affected LOC/modules for Commands Hierarchy and Macro Command (on the right).

ite and Command hierarchy) in Figure 1. Modules of the OOP version have a worse DOF since they explicitly implement design patterns protocols in addition to other secondary concerns while in DPMF system the DP logic is better modularized in the aspects that are derived by the framework from the DSL statements.

4.3 Macro Commands

This section studies how changes to a pattern propagates to base system classes and requires a source code modification. Such requirement change impacts very differently on OOP and DDPF versions. In the OOP version, the needed changes is much higher than the DDPF version. In the DDPF system concrete commands were not impacted by the command aggregation. The only required changes are related to the DSL statements and to the addition of new modules for additional commands. DOS and DOF were evaluated after the maintenance task and provide evidence that the resulting modularity for the DDPF system increases over the OOP version.

4.4 Performance Evaluation

The case study also verified that the AOP-based architecture does not have a negative run-time performance impact (due to aspect runtime interception overhead). With this aim, the AOP system was instrumented in order to gather execution times of the aspect overheads. The worst average times in several different categories of pointcut expressions, as automatically generated by DSL statements (i.e. object creation/destruction, interception of pattern operations), were selected and the time spent in the aspect runtime to jump to pattern logic was collected. We found that time needed to handle creation/destruction of object is usually greater than the time required to intercept operations. In pointcut expressions related to the design patterns operations the worst overheads due to aspect interception mechanism are resulted always less than 5% of the pattern collaboration times.This shows the suitability of performance overhead introduced by the aspect declarations generated from DSL statements.

5 CONCLUSIONS AND FUTURE WORKS

AOSD and MDSD features have been used to develop a design pattern adoption approach and a supporting framework allowing:

- the declarative specification of DPs by a DSL;
- an AOP based implementation of the specified DPs.

The goal is to improve the modularity, the internal code quality, and the flexibility of DPs. DPs are specified by a DSL based on a meta-model where a DP is seen and structured as an (ordered) sequence of named design pattern elements. An improved prototype of the framework, adopting an AOP version of the ExtensionObjectsPattern, was used in a case study to assess its effectiveness and efficiency. The results from the case study showed that the AOP implementation of DPs significantly improved the modularity of the system with respect to traditional OO version. Future work will consider improvements of both the prototype framework and DSL.

REFERENCES

Baca, P. and Vranic, V. (2011). Replacing object-oriented design patterns with intrinsic aspect-oriented design patterns. In *Eastern European Regional Conference on the Engineering of Computer Based Systems*, ECBS-EERC '11, pages 19–26, Washington, DC, USA. IEEE Computer Society.

Bernardi, M. L., Cimitile, M., and Distante, D. (2013a). Web applications design recovery and evolution with RE-UWA. *Journal of Software: Evolution and Process*, 25(8):789–814.

Bernardi, M. L., Cimitile, M., and Lucca, G. A. D. (2013b). An aspect oriented framework for flexible design pattern-based development. In *ICSOFT 2013 - International Joint Conference on Software Technologies, Reykjavík, Iceland, 29-31 July, 2013*, pages 528–535.

Bernardi, M. L., Cimitile, M., and Lucca, G. A. D. (2013c). A model-driven graph-matching approach for design pattern detection. In *20th Working Conference on Reverse Engineering, WCRE 2013, Koblenz, Germany, October 14-17, 2013*, pages 172–181.

Bernardi, M. L., Cimitile, M., and Lucca, G. A. D. (2014). Declarative design pattern-based development using aspect oriented programming. In *Symposium on Applied Computing, SAC 2014, Gyeongju, Republic of Korea - March 24 - 28, 2014*, pages 1625–1630.

Bernardi, M. L., Cimitile, M., and Lucca, G. A. D. (2016). Mining static and dynamic crosscutting concerns: a role-based approach. *Journal of Software: Evolution and Process*, 28(5):306–339.

Boussaidi, G. E. and Mili, H. (2007). A model-driven framework for representing and applying design patterns. In *Computer Software and Applications Conference, 2007.*, volume 1, pages 97–100.

Eaddy, M., Aho, A., and Murphy, G. C. (2007). Identifying, assigning, and quantifying crosscutting concerns. In *International Workshop on Assessment of Contemporary Modularization Techniques*, ACoM '07, pages 2–, Washington, DC, USA. IEEE Computer Society.

Eden, A. H. (2001). Formal specification of object-oriented design. In *International Conference on Multidisciplinary Design in Engineering CSME-MDE 2001*, pages 21–22.

Elaasar, M., Briand, L. C., and Labiche, Y. (2006). A metamodeling approach to pattern specification. In *International Conference on Model Driven Engineering Languages and Systems*, MoDELS'06, pages 484–498, Berlin, Heidelberg. Springer-Verlag.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Giunta, R., Pappalardo, G., and Tramontana, E. (2012). Aodp: Refactoring code to provide advanced aspect-oriented modularization of design patterns. In *ACM Symposium on Applied Computing*, SAC '12, pages 1243–1250, New York, NY, USA. ACM.

Hannemann, J. and Kiczales, G. (2002). Design pattern implementation in java and aspectj. In *ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, pages 161–173, New York, NY, USA. ACM.

Hannemann, J., K. G. (2001). Overcoming the prevalent decomposition of legacy code. In *Workshop on Advanced Separation of Concerns at the International Conference on Software Engineering*, ICSE.

Soundarajan, N. and Hallstrom, J. O. (2004). Responsibilities and rewards: specifying design patterns. In *International Conference on Software Engineering, 2004.*, pages 666–675.

Zdun, U. (2004). Pattern language for the design of aspect languages and aspect composition frameworks. *IEE Proceedings - Software*, 151(2):67–83.