Declarative Design Pattern-based Development using Aspect Oriented Programming

Mario L. Bernardi
Department of Engineering
University of Sannio
Benevento, Italy
mlbernar@unisannio.it

Marta Cimitile
Unitelma Sapienza University
Rome, Italy
marta.cimitile@unitelma.it

Giuseppe A. Di Lucca
Department of Engineering
University of Sannio
Benevento, Italy
dilucca@unisannio.it

ABSTRACT

Aspect Oriented Programming (AOP) can help to reduce crosscutting in the implementation of Design Patterns (DPs), due to typical deficiencies of Object Oriented languages that can affect negatively the quality of the overall software system. The implementation of DPs may be further improved by using Model Driven Development techniques together with AOP. We have defined an approach to specify and to apply, declaratively, DPs to the classes of the base system. A Domain Specification Language (DSL) has been defined to specify declaratively the structure of DPs and their adoption on classes. From the DSL specifications, aspects are derived to compose, at run-time, an AOP-based version of the specified DPs. The approach was validated by a case study where the developed supporting framework was used in a concrete development scenario, and subsequent maintenance task. The results from the case study are presented and discussed.

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

1. INTRODUCTION

Design patterns (DPs) provide well-known solutions to recurring problems that developers have to face during software implementation. There are many advantages of using DPs in software implementation [8]. Despite such advantages, the way DPs are implemented may impact the modularity of a system, affecting its comprehensibility, maintainability and testability. In fact, the usage of DPs implies the presence of crosscutting with the possibility to introduce defects. Aspect Oriented Programming (AOP) and Aspect Oriented Software development (AOSD) [11] can be used to improve the way software is structured, decomposed and implemented reducing crosscutting concerns (CCCs) resulting from DPs adoption. Using AOP constructs, CCCs are encapsulated into a different kind of module (the Aspect) and powerful weaving mechanisms support their subsequent composition with other software artifacts, enforcing modularity (i.e. composition transparency, optionality, and unpluggability) [12, 10]. AOSD can be effectively combined with Model-Driven Software Development (MDSD) improving the overall system modularity of design models, source code, structure and behaviour of run-time objects [3]. As a matter of fact, MDSD supports software development by capturing all relevant system knowledge in models. These models are used in formal transformations to generate all needed artifacts including source code and documentation. Moreover, they are synchronized, combined, and transformed across different levels of abstraction and every model is an instance of a meta model. The meta model defines the DSL describing the abstract syntax to instantiate the meta model when generating a model.

This paper describes and validates an Aspect Oriented DSL-based framework, called DPMF. The framework aims to specify and implement DPs declaratively improving (i) modularity, (ii) dynamic behaviour and (iii) obliviousness. It also aims at improving flexibility of adopting different pattern variants with limited impact on system source code. A greater internal cohesion is obtained leaving concrete system classes as decoupled as possible from DP code components.

With respect to implementing DPs using AOP source code components, the proposed DSL-based approach improves flexibility and maintainability. Indeed, it has the advantage of being completely declarative: changes to the DP implementations and their application to concrete classes do not impact directly on the system source code. This allows developers to change which DPs involve which classes by changing only the DSL specifications.

The framework was built on top of the Eclipse Modeling tools (using Xpand as Model to Text - M2T - transformation engine) and AspectJ as AOP language. It includes a generation step that starts from a model written by the proposed DSL and emits aspects dynamically applying idioms and DPs on system classes with reduced (but often with no) impact on them. The DSL code is parsed by a template-language engine that generates Java and AspectJ resources to implement flexible and modular DPs involving concrete classes of the OO base system.

The case study compares two different implementations of a Java system: one implemented using the proposed framework, against the other implemented using a traditional pattern-based approach. Two teams of expert software engineers was required to develop a Java backend system serving multimedia items for a web/mobile online store. The development, entirely design-pattern based, was carried out by the two teams at the same time. The first group developed the system without using the DPMF framework (i.e., implementing the patterns from scratch) while the second group used DPMF. The comparison was performed evaluating the quality of modularity and the impact of required changes on system classes involving DPs. This allowed to validate the results of applying the framework in terms of: (i) DSL effectiveness and flexibility
to express (and change) design choices, and (ii) internal quality of the resulting system source code.

The remaining of the paper is structured as follows. Section 2 discusses some related work. Section 3 presents and discusses the proposed DSL for Design Pattern implementation and briefly depicts 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 approaches to DPs representation, transformation and applying were proposed in literature. In [7] code generation is used in order to apply DPs to concrete classes using pattern languages or models. In [14], a pattern language aiming to trace and manipulate software structures and dependencies (giving an explanation of existing aspect composition frameworks), is presented. Moreover, an explicit DP representation as well as the transformation embodied in its application is proposed in [6].

There are also several studies proposing aspect-oriented DPs to implement object-oriented DPs. In [1] intrinsic aspect-oriented DPs are used to achieve better composability compared to both original implementations of object-oriented DPs and their aspect-oriented re-implementations. [13] discusses software reuse with aspect-oriented design language and derives the specific requirements for the AOSDDL (Aspect Oriented Software Development Design Language) architecture by examining the AspectJ extensions for a distributed computing environment. All these approaches, are usually focused to apply DPs to the existing design (by models transformations) or code (by applying code generation). Our DSL approach resulted from the analysis conducted on design patterns and crosscutting introduced by them in Object Oriented code proposed in [4, 2]. It differs from the cited approaches for the definition of a dynamic DSL involving existing source code in pattern logic in a completely dynamic fashion. Thus instead of modifying or generating code, our engine generates aspects that inject pattern logic at run-time, keeping system classes oblivious. The advantages are a higher flexibility, a lower invasivity and a lower maintenance effort since only aspects performs interception and run-time bytecode manipulation to apply pattern logic. Finally, in [9] an approach for assisting the refactoring of an application that uses DPs into an aspect-based version, is proposed. With respect to our declarative approach, it provides lower flexibility in modifying patterns application being centered on an explicit refactoring of existing source code. Our approach is instead focused on developing the system from scratch using a declarative language and hence is more suitable in a forward engineering context.

3. DP SPECIFICATION DSL

The DSL is based on a meta-model of an OO system allowing to map DPs (along with their roles, variants and default implementations) on system classes. The DSL is made up of two main parts. The first one models the source code elements, according to the programming language syntax used to implement the system. The current DSL implementation considers Java as the target programming language. The complete metamodel contains 113 meta-classes representing the system elements (including generic types, blocks, statements, expressions and exceptions) and the relationships among them. The DSL is defined according to the abstract syntax of the Java language specification. The description of this first part of the model is out of the scope of this paper.

The second part of the DSL introduces the main elements (concern, role definition and role implementation) and the constructs to perform dynamic interception of a wide range of events (used to involve concrete classes in DP collaboration at run-time).

Figure 1 shows the core excerpt of the second part of the DSL meta-model. 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 Concern element. Each Concern can be seen as a layer containing only the logic related to its goals and responsibilities. Concerns 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 Concerns. In the following example two concerns (FiguresListening and Identification) are defined:

```java
order Identification, ...
concern FiguresListening { ... }
concern Identification { ... }
```

The ordering is fixed (using the composition order statement) so that Identification is always merged before any other concern. This is important when some alterations to the base system are not optional, e.g., when injected members are needed by all other concerns.

Define, Assign and Implement Role(s) elements. The DSL statement define role allows a developer to introduce a role in the system, while the statement assign roles is used to assign defined roles to existing interfaces and classes. In the following example, the DSL specifies that the defined roles Identifiable and Named are assigned in the roles Figure and View.:

```java
concern Identification { 
  define role Identifiable { 
    void setUUID(UUID u); 
    UUID getUUID();
  } 
  define role Named { 
    void setName(String u); 
    String getName();
  } 
  assign roles Identifiable.Named to Figure, View;
... 
```

As a consequence, all the classes inheriting or implementing them will be forced to provide this additional behaviour (the UUIDs and Name elements along with getters/setters in the example). The implement role statement allows to specify the implementation of a role that must be supplied to a set of existing classes. 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 concern elements). In the next example, the inject statement is used to provide an implementation (defined from scratch) of the Identifiable role introduced in the previous example and to apply it to the AbstractFigure and DefaultView (existing) system classes:

```java
implement role Identifiable on AbstractFigure, DefaultView {
  inject {
    UUID _uuid;
    public UUID getUUID() {
      return _uuid;
    }
  } 
... 
```
These classes have no references to the concern “Identifiable” implementation and have no imperative dependencies on Identifiable interface (since in this example the Identification concern is completely orthogonal). However, studies that try to quantify the crosscutting present in real systems, show that most of the concerns 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. Referring to the previous DSL examples, 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):

```java
implment role Identifiable on AbstractFigure+ {
  inject [UUID _uuid ;
  public UUID getUUID() . . .
  public void setUUID(UUID u ) . . .
} String getName() {
  return super.getName() + ", _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 the Identifiable concern, making it dependent on Named role. They are both provided by the same Identification concern, but this is not required. Moreover, different methods can be specified for different subclasses of AbstractFigure, thus reusing the same behaviour as much as possible without losing flexibility.

### 3.1 Specifying Design Patterns 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 and pluggable Composite DP applied to all subclasses of Figure:

```java
concern FigureComposite {
  assign role Component to Figure ;
  assign role Figure to system class AbstractFigure ;
  implement role Component on Figure+ {
    inject ConcreteComponent ;
  }
  implement role Component on AlternativeFigure {
    inject AlternativeConcreteComponent ;
  }
  assign role Composite to PanelFigure , GroupFigure ,
  ZOrderedGroupFigure ;
  implement role Composite on PanelFigure , GroupFigure ,
  ZOrderedGroupFigure {
    inject ConcreteComposite {
      @Override
    boolean hasChildren ( void ) {
      return true ;
    }
} Image (Panel | Group)Figure . getRaster () {
  Image i = Image . build ( ) ;
  for (Component c : getChildren ( ) ) {
    i . merge ( ( Figure ) c ) . getRaster ( ) ;
  }
  return i ;
}
```

Figure 1: The DSL Core Meta-Model: an excerpt of the main structure and elements.
The Component role is assigned to figures that are leaves. Conversely, a Composite role is assigned to the figures that may have internal sub-figures (like GroupFigure or ZOrderedGroupFigure). With reference to the Component role, it’s interesting to note that the class ConcreteComponent is used as the default Component implementation on all Figure hierarchy (the plus notation is used to indicate that). However, for AlternativeFigure, that needs a different kind of implementation, the Component role is provided by another class (namely AlternativeConcreteComponent). This kind allows to inject, by a flexible way, a reasonable default implementation for a complete hierarchy and to change default behaviour when requested. The Composite example uses both shared methods and override injection. The method hasChildren(...) is overridden to return always true, whereas the method getRaster() (defined on Figure interface) is implemented by wrapping the default one. Without the @Runtime annotation, such wrapping is static (the method are injected at compile time by the generated aspects). When using @Runtime the generated aspects use interception to obtain the same results. There is a significant trade-off between time and space, since the dynamic version produces smaller objects but there are less optimization chances and, usually, it is slower. The injected getRaster() uses the getChildren() method (of the CompositeComposite just injected) to obtain all the sub images in order to return an image collating them.

Using the statements already discussed, standard DPs have been implemented and included, as built-in in the framework. Patterns requiring collaborations between several roles (like the Observer, Builder etc.) can be easily constructed using the statements and the predefined (but extensible) interfaces/classes provided by the framework. For instance consider the following excerpt involving the DAO design pattern:

```java
class DAOPattern {
    inject dao for Item {
        @key id,
        @index name, @index title, description, ...
    }
}
```

This example injects the logic that saves/restores the Figure state into a DBMS. The framework, injects add/modify/remove methods, dynamically serializes the listed fields of the target object and writes them into (or retrieves from) a DBMS. The inject dao statement uses a mix of static role implementation and dynamic interception in order to inject a core Data Access Object interface into target classes and dynamic accessor/modifiers based on object structure and state.

To support and evaluate the approach, the DPs Modeling Framework (DPMF) 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 Concern 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 "concern 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 concern, while multiple relationships can be easily resolved in the mapping layer concern by associating two pattern aspects to the same concrete class.

4. CASE STUDY

To assess the effectiveness and the validity of the approach, a real Java system was designed and developed by two different ex-
pert groups; one group adopted the proposed framework whereas the other used a classic design patterns based development. The assessment of the DPMF approach takes into account two key aspects: (i) the quality of the modularization and (ii) the impact on system source code with respect to requirement changes involving adopted patterns.

To evaluate the improvement in the quality of the modularization, a crosscutting comparison between two versions of the system, has been performed. In order to quantify scattering and tangling of DPs code within system classes, DOF and DOS metrics\[5\] are used; this allowed to compare crosscutting across two different systems.

To evaluate the impact of a requirement change in the Command pattern hierarchy implementation (the largest part of the system), several metrics are evaluated (the number of #changed LOC and modules and DOS/DOF variation).

The system, a highly-available back-end serving multimedia items for a web/mobile online store, is written in Java and is comprised of 78684 LOC, 454 classes and 33 interfaces. An excerpt of the most relevant architecture elements is shown in Figure 2. The core component of the server is a multi-context Command-based executor. Each context is associated to a set of Commands by a Factory and exposed through a service endpoint by a Java servlet (using industrial application servers). As shown in the figure, the command processors are created by the main server, implemented by a Singleton DP, and they are associated to a large Command hierarchy, by means of the Factory DPs. Each factory is also responsible to inject its context object to the set of its commands. Most commands work on Items (not shown in the figure). Items are Composites, as can be simple items (of several kind) associated to a single download package or collections of items (called Bundles). The items are garbage collected using a recycling Strategy as implemented by the ItemsCollector. When the reference count becomes zero an item or a bundle is sent to the ItemsCollector that select the right cleanup task for the item kind.

The OOP version required the coding of the Command pattern integrated with several secondary concerns as stated by requirements. In particular in each Command there are explicit fragments linking it to security, logging and tracing concerns (increasing scattering and tangling).

In the DPMF version the entire command hierarchy has been implemented using role assignment and role implementation constructs. This generated a marker interface inside an abstract aspect to remove indirection introduced by the “Command” role. Analyzing the system code, we found that, with respect to the plain OOP version, DPMF effectively allowed to modularize several concerns (making system classes not aware of being linked to the pattern).

4.1 Modularity quantitative assessment

The modularity was assessed for both the DPMF and OOP versions by evaluating (i) the percentage of lines of source code related to Design Pattern logic present in each module with respect to the total Lines of Code (LOC) of the same module, and (ii) the Degree of Scattering (DOS) and Degree of Focus (DOF) metrics\[5\], for each module and concern. This analysis provided quantitative information about (i) size and dimension; (ii) code duplication and (iii) crosscutting concerns presence and distribution.

Results are highlighted in figures 3 and 4. In particular, in Figure 3 the LOC ratio (%LOC) of each design pattern concern over the cloned LOC ratio are compared for each module in both DPMF
(AOP) and OOP versions. The results show that the ratio of the average cloned LOCs in the OOP implementation is significantly greater w.r.t. the DPMF system. In this case the DPMF 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 Composite and Command hierarchy) in Figure 4. 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.

4.2 Introducing Composite on Commands

In order to evaluate how changes to pattern adoption propagates to system classes requiring source code modifications an impact analysis w.r.t. a change in requirements was performed.

In particular the two teams, after the development cycle, were asked to implement commands pipelining and transactions in the Java server to improve performance. This required the refactoring of the Command hierarchy as a composite hierarchy. In particular two composite commands having an inner sequence of simple commands (the PipelineCommand and the TransactionalCommand) were introduced.

As Figure 3 shows on right side, such requirement change had a quite different impact on OOP and DPMF systems. In the OOP case, the number of source code changes (for both the number of affected LOC and the number of affected modules) is much greater than the DPMF case. In the DPMF system almost all concrete commands were not impacted by Composite introduction and most of the changes is related to the DSL statements and to the addition of modules for the new commands (also in these modules the number of affected LOC is reduced with respect to the OOP case).

DOS and DOF, evaluated after the maintenance task, show that the resulting modularity quality for the DPMF system is better than in the OOP case.

4.3 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. In pointcut expressions related to Command, Composite, Strategy and Singleton 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 exploited to develop an approach and a supporting framework allowing: (i) the declarative specification of DPs by a DSL; (ii) an AOP based implementation of the specified DPs. The aim 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 Concern elements. A prototype of the framework 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.

6. REFERENCES


