# An Aspect Oriented Framework for Flexible Design Pattern-based Development

Mario L. Bernardi[1], Marta Cimitile[2] and Giuseppe A. Di Lucca[1]
[1] *Department of Engineering, University of Sannio, Benevento, Italy*
[2] *Unitelma Sapienza University, Rome, Italy*

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

Abstract: The implementation of a Design Pattern (DP) may be affected by some problems due to typical deficiencies of Object Oriented languages that may worsen the modularity of a software system, and thus its comprehensibility, maintainability, and testability. Aspect Oriented Programming allows to implement DPs by its powerful quantification constructs that can handle better modularity and composition, helping to overcome some of the OO design trade-offs in current DP implementations. In Model Driven Development system models, defined by a Design Specification Language (DSL), are transformed between different levels of abstraction to get system implementation. In this paper we propose an Aspect Oriented DSL-based framework to specify and to apply, declaratively, Design Patterns to the system classes. The main aim driving the definition of the proposed framework is to improve the modularity, the internal code quality, and the flexibility, by allowing software designers to specify DP models with an extensive modifiability thus reducing the impact of changes related to DP adoption.

## 1 INTRODUCTION

The way a DP is usually implemented may heavily impact the overall system structure, and in particular it impacts the modularity of the system, thus affecting its comprehensibility, maintainability, testability too. This is mainly due to some typical deficiencies of Object Oriented languages that may worsen the overall modularity of a software system. Moreover, the invasive nature of DP implementations may make it hard to distinguish between the code of DP instances and the code of the 'base' system. In (Hannemann and Kiczales, 2002; Nordberg, 2002) the authors showed how several DPs from GoF catalog (Gamma et al., 1995) introduce crosscutting that OO abstractions are often unable to well modularize.

Aspect Oriented Programming (AOP) and Aspect Oriented Software development (AOSD) (Hannemann and Kiczales, 2001) improves the way software is structured, decomposed and implemented by providing means for modularizing crosscutting concerns (CCCs). CCCs are encapsulated into a new kind of module, the Aspect, and powerful weaving mechanisms support their subsequent composition with other software artifacts. The AOP constructs can help to overcome some of the OO design trade-offs and indirection characterizing current DP implementations. Composition transparency, optionality, and unpluggability are example of modularity properties that can be enforced by AOP implementation of DPs and that have to be considered when assessing the quality of AO designs (Hannemann and Kiczales, 2002; Hachani and Bardou, 2003). Model-Driven Software Development (MDSD) improves the way software is developed by capturing key features of a system in models which are developed and refined as the system is developed. During the system's life-cycle, models are synchronized, combined, and transformed between different levels of abstraction. Thus models have to be formal. Every model is an instance of a meta model. The meta model defines the Domain Specific Language (DSL) describing the abstract syntax to be used to instantiate the meta model when generating a model. While AOSD and MDSD are different in many ways they both help the developer to reason about one concern at a time. From this point of view AOSD and MDSD complement each other and can be used together to improve both the overall modularity of development of artifacts at design models down to source code, and the structure and behaviour of run-time objects .

In this paper we exploit AOSD and MDSD fea-


L. Bernardi M., Cimitile M. and A. Di Lucca G..
An Aspect Oriented Framework for Flexible Design Pattern-based Development.
DOI: 10.5220/0004510305280535
In *Proceedings of the 8th International Joint Conference on Software Technologies* (ICSOFT-PT-2013), pages 528-535
ISBN: 978-989-8565-68-6

tures to propose an Aspect Oriented DSL-based framework used to specify and to implement DPs declaratively. The main goals behind the framework design are: (*i*) modularity, to improve code internal quality avoiding bad duplicated code, (*ii*) dynamic behaviour, to improve flexibility of adopting different pattern variants with limited impact on system source code; and (*iii*) obliviousness, to increase internal cohesion by leaving concrete system classes as much decoupled as possible from design pattern protocols, abstract classes and interfaces.

The framework is centered on a generation step that, starting from a model written by the proposed DSL, emits aspects dynamically applying idioms and DPs on system classes with reduced (and often 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 overall design is more modular than a pure Object Oriented one and benefits from interesting dynamic properties like optionality, pluggability (the ability to enable/disable o change DP involvement at runtime). Moreover the performance hit, due to dynamic aspect introspection, is greatly reduced.

To validate the approach, a prototype framework was built on top of the Eclipse Modeling tools (using Xpand as Model to Text - M2T - transformation engine) and AspectJ as AOP language. Quantitative assessment is done comparing the AO versions of the applied DPs against reference Object Oriented DPs' implementations. The comparison is performed by means of a selected set of AOP-aware internal-quality metrics. A qualitative discussion highlighting the properties of the generative approach is also provided. This allows 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. Sections 3 and 4 present and discuss the proposed DSL for Design Pattern implementation and the architecture of the AspectJ-based framework adopting it. The section 5 reports a description of the the case study and discusses the quantitative evalutation of the proposed framework using an adequate set of AOP-aware source code metrics. Conclusive remarks and future works are finally presented in section 6.

## 2 RELATED WORK

In (Hannemann and Kiczales, 2002) authors provide a study on crosscutting introduced by design patterns adoption providing AOP-based implementation that have influenced requirements for our DSL definition and design goals.

The topic has largely discussed in the last years in literature and several approaches to represent, transform and apply design patterns were proposed (Alencar et al., 1997; Elaasar et al., 2006). These approaches are mainly interested in applying a code generation approach to apply design patterns to concrete cases using pattern languages or models.

In (El Boussaidi and Mili, 2007) an explicit representation of a pattern as well as the transformation inherent to its application is proposed. (Baca, 2011) shows how intrinsic aspect-oriented design patterns can be used to implement object-oriented design patterns in order to achieve better composability compared to both original implementations of object-oriented design patterns and their aspect-oriented re implementations. In (Zdun, 2004) a pattern language is presented for tracing and manipulating software structures and dependencies, with an explanation of different, existing aspect composition frameworks as sequences through this pattern language. Alternative designs, common design trade-offs, and design decisions for implementing aspect composition frameworks, are also evaluated. (Soundarajan and Hallstrom, 2004) discusses the software reuse in the design and implementation for aspect oriented design language in general and derives the specific requirements for the AOSDDL (Aspect Oriented Software Development Design Language) design language architecture by examining the Aspect J 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 crosscutting introduced by DPs in Object Oriented code proposed in previous works (Arpaia et al., 2010; Bernardi et al., 2005; Bernardi et al., 2012) . The main difference of our approach w.r.t. cited approaches is 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 generate aspects that inject pattern logic at run-time, keeping system classes oblivious. For this reason, the proposed approach can be more flexible, less invasive and requires less maintenance effort since only aspects performs interception and run-time bytecode manipulation to apply pattern logic.

# 3 THE PATTERNS SPECIFICATION LANGUAGE

The framework is based on a meta-model used to define a DSL suitable to map design patterns (along with their roles, variants and default implementations) onto system classes. A DSL instance is structured as an (ordered) sequence of named Concern elements. Each Concern has in turn an inner structure specifying its contributions to the base system:

- provided roles (i.e. interfaces along with a concrete implementation)
- injection of built-in roles (in particular DP roles) into existing types (including hierarchies)
- aggregation/composition rules among existing classes or interfaces
- injection of behaviour into (and interception at run-time of) existing classes

The DSL is made up of two main parts: (i) a language part (Java in the current prototype) allowing the representation of code elements (including generic types, blocks, statements, expressions and exceptions); (ii) a second part introducing the main concepts of the DSL language (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). The complete Java metamodel contains 113 meta-classes representing the elements, and the relationships among them, of a Java software system according to the abstract syntax of the Java language specification as specified in (Gosling et al., 1996) The description of this part of the model is out of the scope of this paper; it has however a key role since it is used whenever referencing or introducing new source code elements.

The Figure 1 shows the core excerpt of the second part of the DSL meta-model. This part describes the overall structure of the DSL and focuses on the main concepts used to implement idioms and design patterns (leaving base system classes oblivious and decoupled from pattern logic). For each idiom or DP, the framework performs a mix of injection, alterations and introductions of the system classes. We describe the most important ones in the remaining of the section by means of small code examples written in the proposed DSL.

## 3.1 The Concern Element

The crosscutting model represented by the proposed DSL can be seen as the weaving of an ordered sequence of Concerns: each Concern can be seen as a layer containing only the logic that is 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: this order can be changed using the composition order statement like in:

```
order Identification,*
concern FiguresListening { ...
concern Identification { ...
```

In this case two concerns (FiguresListening and Identification) are defined; the ordering is fixed so that Identification is always merged before any other concerns. This is important when some alteration to the base system are not optional: injected members needed by all other concerns (or even by the base system itself).

## 3.2 Define, Assign and Implement Role(s) statements

These statements are the groundings for all other constructs that follows. They allow developer to introduce a role in the system (using *define role*) and to assign them to existing interfaces and classes (using *assign roles*). Let us consider the following *define role* statements:

```
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;
        ...
```

This fragment introduces the roles Identifiable and Named into two existing roles (Figure and View). As a consequence all the classes inheriting or implementing them will be forced to provide this additional behaviour (the UUIDs and Name fields along with getters/setters in the example). This is useful only if we have a means to inject (or not) such behaviour in a modular way: i.e. providing Identification concern default implementation for classes of the base system but having the flexibility to specify different implementations for different classes. This is exactly what the *implement role* statement does. It allows to specify the implementation of a role that must be supplied for a set of existing classes. The syntax is based upon the InjectionRule nested element that must be

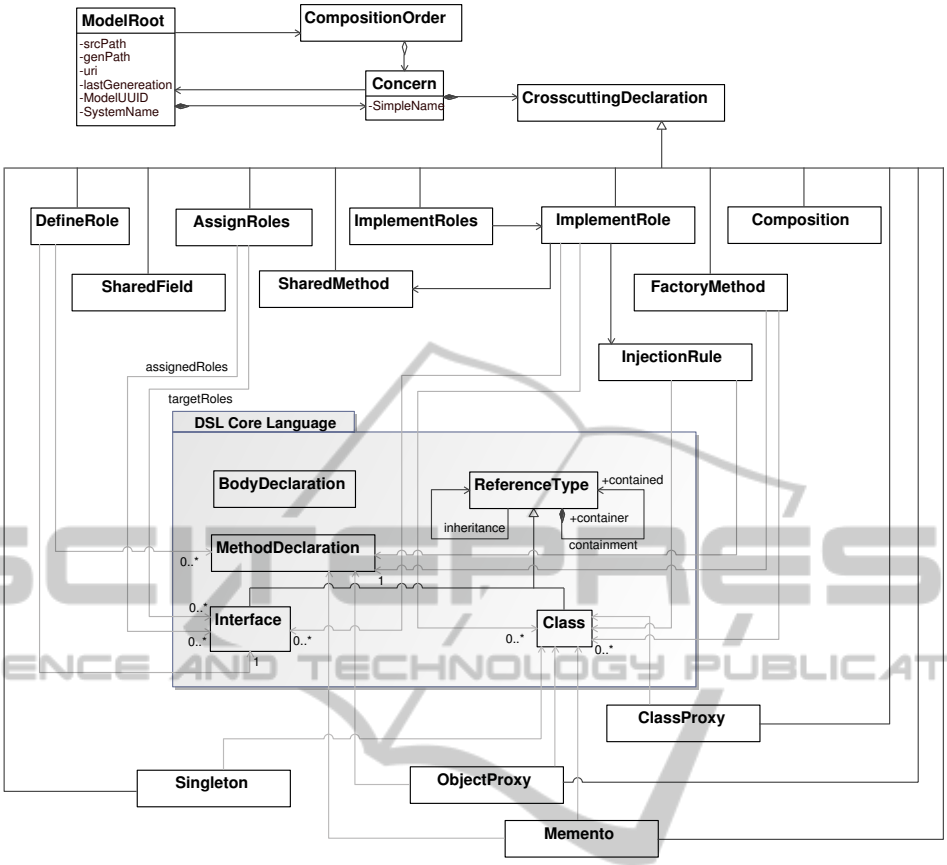


Figure 1: The DSL Core Meta-Model : an excerpt of the main stucture and elements.

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 target types members (including the ones that are weaved from other concern elements). For instance the example:

```
implement role Identifiable
               on AbstractFigure, DefaultView {
 inject {
  UUID _uuid;
   public UUID getUUID() {
    return _uuid;
   }
   public void setUUID(UUID u) {
    _uuid=u;
   }
  }
}
```

uses the *inject* statement to provide an implementation (defined from scratch) for the Identifiable role seen before and apply it to the AbstractFigure and DefaultView (existing) system classes. These classes are by no means aware that they are "Identifiable" by UUID and have no imperative dependencies on Identifiable interface (since in this trivial example the Identification concern is completely orthogonal). This however is not the "real world" common case. Studies that tries to quantify the crosscutting present in real systems reveal that most of the concerns are crosscutting and hence they depend on each other. To express such interleaving the *implement role* statement allows the definition of shared fields and methods.

## 3.3 Shared Fields and Methods

When implementing a role for a set of concrete classes one or more methods could be provided in order to link, in a modular way, the concrete classes logic to the new provided behaviour. An example will make this more clear. Referring to the previous DSL snippet, the UUID could be used to build the AbstractFigure name. In this case a shared method (the same concept applies to shared fields) can be used. It should be nested in the role implementation statement as follows:

```
implement role Identifiable
```

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

This excerpt injects the getName(void) method in the complete hierarchy rooted in the AbstractFigure class as a part of Identifiable concern. This means that Identifiable role depends on Named role. They are both provided (in this simple case) by the same Identification concern but this is not required. Moreover different method can be specified for different subclasses of AbstractFigure thus reusing the same behaviour as much as possible without loosing flexibility.

### 3.4 Composite Example

Using the statements seen so far, we could easily implement a modular and pluggable Composite implementation. Consider the following excerpts in which a Composite DP is applied to all subclasses of Figure:

```
\renewcommand{\baselinestretch}{0.96}
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;
 }
 }
}
```

Some figures are leafs and hence Component role is assigned to them. Conversely to figures that may have internal sub-figures (like GroupFigure or ZOrderedGroupFigure) a Composite role is assigned. Referring to the Component role, it's interesting to observe that the class ConcreteComponent is used as the default Component implementation on all Figure hierarchy (using the plus notation). However for AlterativeFigure, that needs a different kind of implementation, the Component role is provided by another class (namely AlternativeConcreteComponent). This kind of flexibility allows to inject a reasonable default implementation for a complete hierarchy and to change default behaviour when requested. The Composite case 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. Since no @Runtime annotation is provided such wrapping is static (the method are injected at compile time by the generated aspects). When using @Runtime the generated aspects uses interception to obtain the same results. There is a tradeoff between time and space since the dynamic version produces smaller objects but there are less optimization chances and it is, usually, slower. The injected getRaster() uses the getChildren() method (of the ConcreteComposite just injected) to obtain all the sub images returning an image containing all of them.

### 3.5 Other Statements

Using the statements already discussed all DPs have been implemented and included, as built-in in the framework (and as additional DSL statements in some cases). 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 with the framework.

Interesting cases are patterns based on single role (like Singleton, Memento, or Factory Method) for which a concise DSL statement is implemented. For instance consider the following excerpt:

```
concern FigureMemento {
        memento on Figure+
           for {title,center};
        memento on CircleFigure
           for {title,radius,center};
}
```

This example injects the logic that saves/restores the Figure state into an opaque object. The framework, when the getMemento() is called, dinamically serializes the listed fields of the target object

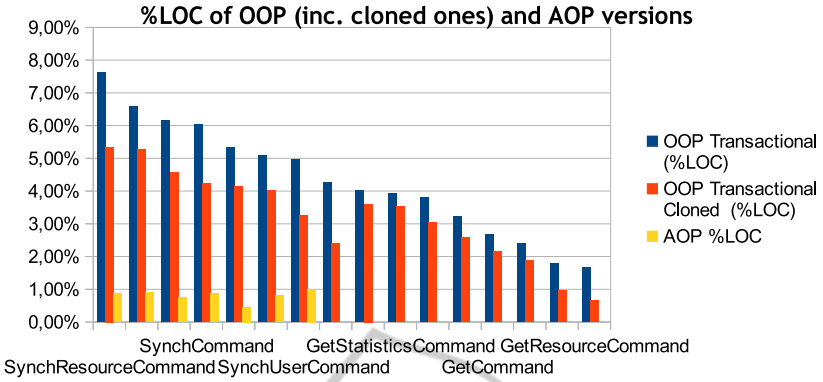


Figure 2: Transactional Commands: Cloned LOC Ratio comparison.

and write them into a memento object. When setMemento(Memento m) is called the state fields are restored. This statements uses static role definition and implementation to inject memento interface into target classes.

# 4 THE DESIGN PATTERNS MODELING FRAMEWORK (DPMF)

In our prototype framework, 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 aspect layer is responsible of implementing a modular mapping of DPs and idioms to concrete classes intercepting object creation and enforcing the observer/observable 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.

# 5 CASE STUDY OVERVIEW

To validate and assess the approach, the same system was designed and developed by two different expert groups; one group adopted the proposed framework whereas the other used a classic design patterns based development. The system, a high performance REST server providing dynamic content for a family of mobile applications, is written in Java and is comprised of 24824 LOC , 364 classes and 28 interfaces. The original Java system was pattern-based. The core component is a Command-based executor initialized by means of a Command Factory. Executors are mapped to different endpoints into industrial application servers (Jetty in this case). The Server is a Singleton and is responsible of managing an executor for each endpoint. We performed a metric-based comparison between the old pure-java system and the re-engineered version using the proposed framework. The restructuring was performed by removing patterns adoption from all system classes using the declarative DSL to implement them.

The restructuring of the Command hierarchy was the largest and most difficult one. The command pattern was implemented using a role assignment and implementation. This generated a marker interface inside an abstract aspect to remove indirection introduced by the ”Command” role. Looking at the system code, we found that the original design failed to effectively modularize several concerns. Command and Composite pattern were tangled in an explicit way, making difficult to change one aspect without impacting the other ones. The refactored system enforces a clean separation between the two providing a default role implementation for the all simple Commands and a MacroCommand role implementation that, transparently, aggregates sequence of plain commands with no impact on the Command pattern struc-

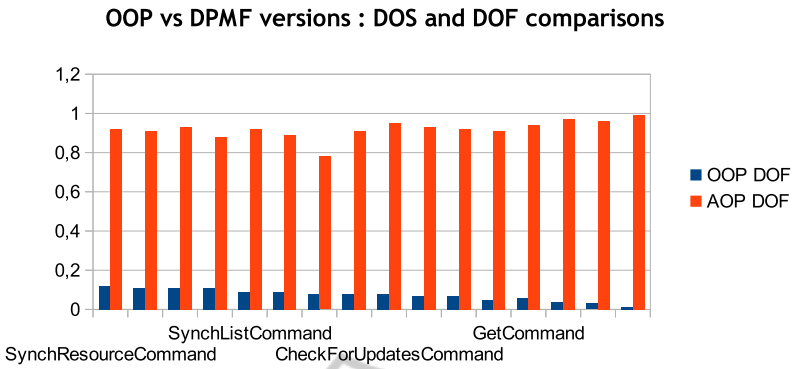


Figure 3: Transactional Commands: DOF comparison.

ture itself. Using shared methods the composite operations were, in some cases, overridden to provide custom behaviour (like in the case of TransactionalMacroCommand that needed a @Runtime interception to wrap the sequence execution in a transactional context). This was a big improvement over the original design in which transactions were not modularized. Moreover in each command there were authentication, logging, tracing and business logic concerns with high level of scattering and tangling.

## 5.1 Quantitative Assessment

The software quality attribute of modularity was assessed for both the AOP and OOP versions by evaluating (i) the percentage of lines of source code related to fault detection 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 (Eaddy et al., 2007), for each module and concern. This analysis provides quantitative information about (i) size and dimension ; (ii) cohesion and coupling and (iii) crosscutting concerns presence and distribution. The DPMF-based software quality was evaluated in comparison with the corresponding OOP version.

## 5.2 AOP Metrics Results

Results are highlighted in Figures 2 and 3. In particular, in Figure 2 the percentage LOC (%LOC) of the Transactional concern all over the Command are compared for both AOP and OOP versions. In the figure, the ratio of the cloned LOCs in the OOP implementation, completely removed in the AOP version, is reported. Of course, the cloned code makes worst the maintainability and increases the probability of introducing bugs in the code.

In Figure 4, the level of DOS (a) and DOF (b) for each Command module with respect to Base System

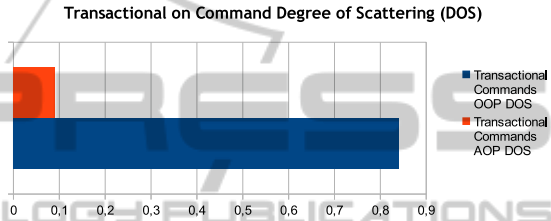


Figure 4: Transactional Commands: DOS comparison.

and Transactional concerns is reported. The results show a radically increased modularity for the AOP version, because each Command module is much more focused with respect to the OOP version. Moreover, the Transactional concern is highly scattered in the OOP version (high values of DOS), while it is very focused in the AOP implementation (this is verified by the low values obtained for DOS).

## 5.3 Performance Evaluation

The case study was aimed also at verifying experimentally that the AOP-based generative architecture would not have a negative impact on run-time performance of the overall system (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 main attention was paid to evaluate the overheads added by AOP interception mechanism to the injected pattern logic time in order to assess the effectiveness of the AOP architecture, i.e. that the AOP response times are not worst than the OOP version. The above described analysis was carried out by running the two versions of the software in the same conditions. The worst average times in several different categories of pointcut expressions related to Transactional Macro command and that were 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 routines were collected.

Times needed to handle creation/destruction of object are usually greater than those required to intercept operations. Command Factories must be set up for Command executors to being created and operative. This requires more time than the other kind of pointcuts expressions, that have only to capture the context of an operation, issuing error or executing business logic if necessary.

In pointcut expressions related to Command design pattern operations the worst overheads due to aspect interception mechanism are always less than 1.5% of the pattern collaboration times. Therefore, the suitability of performance overhead was assessed, where all the timing constraints were satisfied flatly.

## 6 CONCLUSIONS AND FUTURE WORKS

An Aspect Oriented DSL-based framework to specify and to apply, declaratively, Design Patterns to the system classes has been proposed in this paper. AOSD and MDSD features are exploited to improve the modularity, the internal code quality, and the flexibility of DPs. The framework allows software designers to specify DP models with a more extensive modifiability thus limiting the impact of changes, related to DP adoption, on the code of the base system. 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 version of DPs dramatically improved the modularity of the system with respect to the 'traditional' OO version. Future work will consider improvements of the prototype framework and DSL.

## REFERENCES


Gamma, E., Helm, R., Johnson, R., Vlissides, J., 'Design Patterns: Elements of Reusable Object-Oriented Software', Addison-Wesley (1995).

Hannemann, J., Kiczales, G., 'Design Patterns Implementation in Java and AspectJ', Proc. of Object Oriented Programming Systems Languages and Applications 2002 (OOPSLA '02).

Hachani, O. , Bardou D., 'On Aspect-Oriented Technology and Object-Oriented Design Patterns' , Proc. of European Conference on Object Oriented Programming 2003 (ECOOP 2003).

Nordberg Martin E., 'Aspect Oriented Indirection - Beyond Object Oriented Design Patterns' , Proc. of Workshop. Beyond Design: Patterns (mis)used, Proc. of Object Oriented Programming Systems Languages and Applications 2002 (OOPSLA '02).

AspectJ web site - http://www.eclipse.org/aspectj

Hannemann, J., Kiczales, G.,'Overcoming the Prevalent Decomposition of Legacy Code', Workshop on Advanced Separation of Concerns at the International Conference on Software Engineering 2001 (ICSE'01).

Eaddy, M., Aho,A , Gail C. Murphy., 'Identifying, Assigning, and Quantifying Crosscutting Concerns', Proc. of the First International Workshop on Assessment of Contemporary Modularization Techniques (ACoM '07), IEEE Computer Society, Washington, 2007.

Arpaia, P., Bernardi, M.L., Di Lucca, G., Inglese, V., Spiezia, G.,'An Aspect-Oriented Programming-based approach to software development for fault detection in measurement systems', Comput. Stand. Interfaces 32, 2010.

El Boussaidi, G., Mili, H. , 'A model-driven framework for representing and applying design patterns', Proc. of the 31st Annual International Computer Software and Applications Conference (COMPSAC '07), Vol. 1. IEEE Computer Society, Washington, DC, USA, 2007.

Alencar, P.S.C., Cowan, D.D., Dong, J., Lucena, C.J.P., 'A transformational Process-Based Formal Approach to Object-Oriented Design', Formal Methods Europe, 1997

Elaasar, M., Briand, L.C., Labiche, Y., 'A metamodeling approach to pattern specification', In Proc. of the 9th international conference on Model Driven Engineering Languages and Systems (MoDELS'06), Springer-Verlag, Berlin, Heidelberg, 484-498, 2006.

Baca, P. Vranic, V., 'Replacing Object-Oriented Design Patterns with Intrinsic Aspect-Oriented Design Patterns', Proc. of the 2nd Eastern European Regional Conference on theEngineering of Computer Based Systems (ECBS-EERC), pp.19,26, 5-6 Sept. 2011.

Zdun, U., 'Pattern language for the design of aspect languages and aspect composition frameworks', Software, IEE Proc. , vol.151, no.2, pp.67,83, 5 April 2004.

Soundarajan, N., Hallstrom, J.O., 'Responsibilities and rewards: specifying design patterns', Proc. of 26th International Conference on Software Engineering 2004 (ICSE 2004).

Gosling, J., Joy, B., Steele, G.L., 'The Java Language Specification (1st ed.)', Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

Bernardi, M.L., Di Lucca, G.A., 'Improving Design Pattern Quality Using Aspect Orientation', Proc. of the 13th IEEE International Workshop on Software Technology and Engineering Practice, 2005.

Bernardi, M.L., Cimitile, M., Maggi, F. M., 'Model Driven Development of Process-centric Web Applications', Proc. of the 7th International Conference on Software Paradigm Trends 2012 (ICSOFT 2012).