Parametric Aspects: A Proposal

Jordi Alvarez

Universitat Oberta de Catalunya
Avinguda Tibidabo 39–43, 08035 Barcelona, Spain
jalvarezc@uoc.edu

Abstract. Aspect-Oriented Software Development (AOSD) provides better design solutions to problems where Object Oriented Development produces tangled and scattered designs. Nevertheless, there are still several problems for which AOSD is not helpful. An example is the implementation of some design patterns. While it has been shown that some of them can be implemented in a domain-independent way by the use of AOSD [1], there is still a group of them for which current AOSD techniques are of little use. This paper proposes to extend Aspect Oriented Languages with parametric aspects. This extension can integrate seamlessly into Aspect Oriented Languages like AspectJ, and allows to provide a better design solution for problems for which current AOSD techniques are of little help, improving software reuse and reducing its complexity, thus facilitating the software evolution process. Two representative examples are used in order to expose the proposed extension: the implementation of the abstract factory pattern in a domain-independent way, and that of a simple Enterprise Java Bean.

1 Introduction

AOSD solves some of the problems that arise in OO Development [2, 3]. In problems where OO produces a tangled and scattered design, Aspect Orientation can achieve a cleaner and more compact design. Nevertheless, the ideas behind AOSD are in continuous evolution and have not yet been completely developed. Software engineering properties promoted by the use of AOSD can still be further improved in several directions. This paper proposes an extension that allows to increase software reuse and to decrease software complexity. Software complexity has been identified as one of the main drawbacks for a successful software evolution process [4]. Thus, we are concerned here with the capability to produce software systems more suitable for software evolution.

The implementation of GoF design patterns [5] with AO techniques (using the AspectJ language [6]) has been boarded in [1]. A subset of 12 design patterns from [5] has been implemented in an abstract way (completely independent of the domain). This means that pattern implementation can be reused for different applications of the design pattern to different domains. The rest of the patterns in [5] cannot be completely implemented in a domain independent way with current AOSD techniques. It is argued that they are "too abstract" in order to provide a domain-independent implementation for them [7]. Nevertheless, these problems for which current AO Languages are not able to provide qualitatively cleaner and more compact design, still show some regularities and common behaviour. When software needs to be evolved, these regularities should be taken into account in the evolution process; and their complexity has an affect on the resulting complexity of the evolution process.

This paper explores how ideas in parametric types (or genericity) can be extended to AO languages. A proper combination of both can be powerful enough to capture some of the regularities that AO languages alone were not able to capture. Our AO extension allows to decouple these regularities from the part of the software system that corresponds to the application domain, coding them only once in an abstract enough way. As a result, the complexity of the latter can be significantly reduced. As this is mainly the part of software systems that is subject to software evolution, also the complexity that the software evolution process shall deal with, can be importantly reduced.

Parametric types (or generics) were introduced to the main stream of the software industry with the C++ language [8]. Also, several proposals to add parametric types to the Java language exist [9–13], and the next Java release: Java SDK 1.5, will allow them [14, 15]. The main idea behind parametric types is to introduce parameters into type definitions. In this way, regular types can be defined in two phases: (1) a "generic" type is defined, in which one or more formal parameters take part, and (2) a regular type can be created as an instance of the generic type just by replacing (at compile time) the formal parameters by other types or constants.

Our approach borrows this two-phase definition mechanism for aspects. This does not only adds genericity 'a la C++' to AO Languages, but also takes benefit from AO ideas in order to provide a more powerful genericity that perfectly fits into the AO approach. The paper also uses two very representative examples in order to show how this extension helps indeed to produce much more compact designs.

2 Parametric Aspects Definition

This section describes which constructs do we propose to add to aspect languages in order to be able to define parametric aspects. Although the proposal is valid for the aspect oriented approach in general, an extension of AspectJ syntax has been developed and is used in the provided examples.

Throughout the text, the *AbstractFactory* GoF design pattern [5] will be used as an example. This design pattern is one of the patterns for which current AOSD techniques cannot provide a domain-independent implementation. It will be shown how the parametric aspect extension proposed here allows to provide a domain-independent implementation of a restricted version of the *AbstractFactory* pattern, allowing to reuse it from domain to domain. In this way, if the application domain changes and the software needs to be evolved, the design pattern implementation will not take part in the performed changes, remaining unchanged.

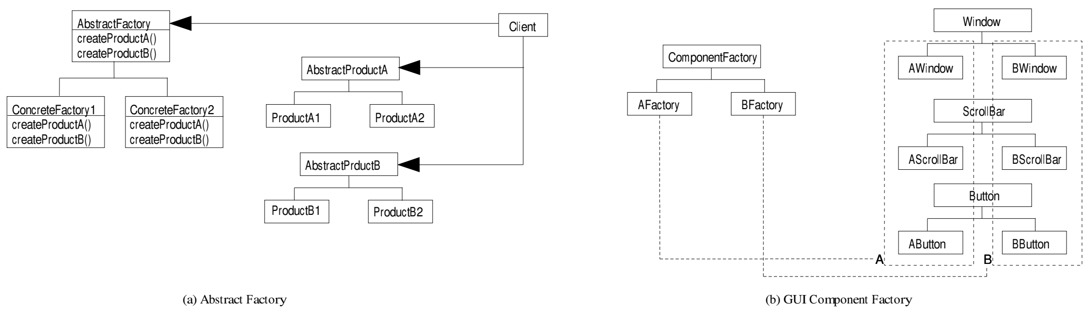

Fig. 1. Representation of the *AbstractFactory* pattern and a simplified implementation of it for the domain of GUI components.

Figure 1 part (a) shows a representation of the *AbstractFactory* pattern. The goal of this pattern is to detach the creational behaviour for classes in a hierarchy (abstract products) from the hierarchy itself. This allows to provide different creational strategies (factories) without the need to modify the hierarchy. Part (b) corresponds to the application of the *AbstractFactory* pattern to the domain of GUI components.

The same as parametric types, parametric aspects have a set of formal parameters attached to its definition. In what follows, we refer to these parameters as *roles*, in a similar way to the roles taking part in design patterns. Roles in parametric aspects may correspond to types (classes), methods, attributes, and constants. Note that this is different to parametric types, where parameters correspond only to types or constants.

The piece of code in next page shows the implementation of a restricted version of the *AbstractFactory* pattern through a parametric aspect. The following constructs are part of the syntax extension to AspectJ:

- The **roles** keyword is used to specify the roles (parameters) taking part in the parametric aspect. Roles may correspond to classes/types, methods, attributes, constants, and class sets. From now on we will refer to these elements with the common name of language objects.
- The **generic** keyword allows to partially define classes, methods, and other language objects. These "generic" definitions contain references to roles, and are not properly defined as regular language objects until the parametric aspect is instantiated, and the roles are replaced by regular language objects (see next section). So, these "generic" definitions are indeed skeletons that will be completed when the aspect is instantiated.
- One skeleton might correspond to multiple definitions when the parametric aspect is instantiated. When this is possible, the keyword **multiple** is used. `AbstractFactoryAspect` (see next page) contains one class skeleton for `<ProductSet>Factory`. When the parametric aspect is instantiated, this only definition produces one class definition for every class-set playing the role `ProductSet` (see also next section).
- Class sets are defined through the **class-set** keyword. Class sets will not have a counterpart in the final program code. Nevertheless, the possibility to have this kind of roles, and to use them in skeleton definitions will be shown to be very useful.
- A special expression language is also provided in order to use roles within generic definitions (skeletons). The expressions are evaluated at weave time and enclosed between '<' and '>' symbols. Simple expressions allow just to refer to roles; and more complex expressions can conveniently traverse the class hierarchy (see `create<AbstractProduct>` method definition, and next section for the explanation). These expressions will be referred from now on as *parametric expressions*. Class, method, and attribute names can be a combination of parametric expressions and text prefixes and suffixes. This allows to define very powerful skeletons.

```
abstract aspect AbstractFactoryAspect {
  roles AbstractFactory, AbstractProduct, ProductSet;

  generic class <AbstractFactory> {
    multiple public abstract <AbstractProduct> create<AbstractProduct>();
  }

  multiple generic class <AbstractProduct>;

  multiple generic class-set <ProductSet>;

  multiple generic class <ProductSet>Factory extends <AbstractFactory> {
    multiple public <AbstractProduct> create<AbstractProduct>() {
      return new <subclasses(AbstractProduct) & members(ProductSet)>();
    }
  }
```

The definition of `AbstractFactoryAspect` contains several class-skeleton definitions: `AbstractFactory`, `AbstractProduct`, and `<ProductSet>Factory`. Additionally, `<AbstractFactory>`, and `<ProductSet>Factory` contain also method-skeleton definitions for the factory creational methods. All these definitions may have a total or partial definition inside the parametric aspect. These definitions can be further completed when the aspect is instantiated. This allows language objects playing a given role to be forced to respect a given structure.

3 Aspect Instantiation

Parametric aspects will always be abstract aspects. They can be instantiated using the **extends** keyword, resulting in regular aspects that can be used by developers as any other concrete aspect in the system. In order to achieve this, the aspect that instantiates the parametric aspect must provide replacement for all the roles appearing in it.

All roles must be accordingly replaced by language objects (regular types, methods, attributes, constants, or class sets). For each one of these language objects, we say that the language object *plays* the corresponding role.

The piece of code below shows how `AbstractFactoryAspect` can be instantiated for the GUI component domain. The resulting `ComponentFactoryAspect` only needs to specify the replacement objects for the roles declared in `AbstractFactoryAspect`. In other situations, additional behaviour could be provided by the concrete aspect in the usual way.

```
aspect ComponentFactoryAspect extends AbstractFactoryAspect {
    class ComponentFactory plays AbstractFactory;
    class Window plays AbstractProduct;
    class ScrollBar plays AbstractProduct;
    class Button plays AbstractProduct;
    class-set A plays ProductSet { AWindow, AScrollBar, AButton; }
    class-set B plays ProductSet { BWindow, BScrollBar, BButton; }
}
```

`ComponentFactoryAspect` shows how, once the *AbstractFactory design pattern* is defined as a parametric aspect, applying the pattern to a concrete domain can easily be done by defining another aspect that fills the gaps corresponding to the roles for the application domain.

`ComponentFactoryAspect` replaces the role `AbstractFactory` by the `ComponentFactory` class. We say that `ComponentFactory` class plays the `AbstractFactory` role. There are three different classes (`Window`, `ScrollBar`, and `Button`) that play the `AbstractProduct` role. Two class-sets:`A` (containing `AWindow`, `AScrollBar`, and `AButton`) and `B` (containing `BWindow`, `BScrollBar`, and `BButton`) play the `ProductSet` role.

Then, if the application domain changes (for example, a new factory or a new set of products is needed), and our software system needs to be adapted to it, only new roles and their specific behaviour must be defined. The abstract factory design pattern implementation has been completely decoupled from the application domain code, and can remain unchanged, favouring the software evolution process requested by the domain change.

Skeleton definitions in the abstract parametric aspect correspond to language objects. Every one of these language objects has a *unique identifier*. The unique identifier of a class is its name and package. The unique identifier of a method is its name plus the type of its parameters plus the unique identifier of the class that contains it. The unique identifier of a skeleton definition in the parametric aspect can contain parametric expressions. In order to produce regular language object definitions from the skeletons, the parametric expressions that take part in their unique identifier are evaluated according to the role assignments established in the concrete aspect. When several language objects play the same role, all the possible combinations are generated, and one (class/method/...) definition is generated for every combination.

Then, as `ComponentFactoryAspect` declares two `ProductSet`, two class definitions would be generated for `<ProductSet>Factory`: `AFactory`, and `BFactory`. The same happens with method definitions: as three classes play the role `AbstractProduct`, three method definitions would be generated for the `create<AbstractProduct>` method skeletons in `AbstractFactoryAspect`. This would result in three methods for each one of the classes: `ComponentFactory`, `AFactory`, and `BFactory`. The set of classes and method definitions that would be generated at weave time is shown below:

```
class ComponentFactory {
  public abstract Window createWindow();
  public abstract ScrollBar createScrollBar();
  public abstract Button createButton();
}
class AFactory extends ComponentFactory {
  public Window createWindow()       { return new AWindow(); }
  public ScrollBar createScrollBar() { return new AScrollBar(); }
  public Button createButton()       { return new AButton(); }
}
class BFactory extends ComponentFactory {
  public Window createWindow()       { return new BWindow(); }
  public ScrollBar createScrollBar() { return new BScrollBar(); }
  public Button createButton()       { return new BButton(); }
}
```

In order to be able to generate the bodies of the methods in `AFactory` and `BFactory` from just one method skeleton in `AbstractFactoryPattern`, the *parametric expression language* must be powerful enough. With this goal, a language is provided that allows to deal with class sets, obtain the superclasses and the subclasses of a given class, and make intersections and unions of different class sets. This behaviour is obtained with the use of a few functors (**members**, **subclasses**, **superclasses**, **subclasses***, and **superclasses***) in combination with set-related operators (`&` and `|`). The complete syntax is not shown here for lack of space. This reduced set of features allows to conviniently traverse the class hierarchy at weave time.

```
<subclasses(AbstractProduct) &
             members(ProductSet)>
```

The above parametric expression (extracted from `create<AbstractProduct>` method skeleton in `AbstractFactoryAspect`) computes the intersection of the subclasses of the class playing the `AbstractProduct` role and the classes in the class-set playing the `ProductSet` role. That is, it computes the concrete product to be created for a given factory and an abstract product.

As parametric expressions are evaluated at weave time, they cannot deal with modifications introduced into the class hierarchy through the use of runtime reflection. Additionally, the weaving process itself may update also the hierarchy, which poses a recursion problem as the evaluation of parametric expressions before the weaving process might differ from their evaluation after it. The simplest solution is taken and only the hierarchy before the weaving process is taken into account.

4 Related Work

The introduction of parametric type ideas into AOSD has also been boarded recently by other researchers. [16] proposes parametric introductions, which allow to add the classes to which introductions apply as parameters to the introduction definitions themselves. This allows, for example, to easily implement the *singleton* design pattern. The main difference with our approach is that [16] has as parameters the classes being instrumentalized, whereas in our approach parameters are one more element in the aspect itself, and might refer to newly created language objects (classes, methods . . .). On the other hand, the use of the parametric expression language provides a very powerful way to define behavior.

Family Polimorphism [17] addresses a similar problem, although it is not related to the AOSD approach. Also, [18] combines aspects and frame technology, with the goal of adding parametrisation and also deeper generalisation capabilities to AOSD systems.

5 Conclussion and Further Work

This paper proposes to extend AOP languages in order to support parametric aspects. The extension allows to define behaviour in a domain-independent way for problems for which the existing AO languages are only able to provide an overall domain-dependent solution. As a consequence, the complexity of the part of software systems that are dependent on the domain, and the complexity that the software evolution process must deal with, can be significantly reduced.

An extension of AspectJ with the syntax shown in this paper is currently under development. Additionally, the scope of application of the proposed extension is being evaluated for other design patterns and also other different problems.

A Appendix. A Second Example: EJB Development

EJB development is a tedious task. Many interfaces with redundant code need to be developed. This task can be lightened by development environments and tools. Nevertheless, no solution is provided from the programming language point of view. The use of parametric aspects could help in avoiding the development of redundant code, as the code below shows.

```
abstract aspect EJB {
    roles Bean, beanService;
    generic class <Bean> {
        generic multiple public <beanService>;
    }
    interface <Bean>Remote extends javax.ejb.EJBObject {
        public <beanService> throws java.rmi.RemoteException;
    }
    interface <Bean>RemoteHome extends javax.ejb.EJBHome {
        <Bean>Remote create() throws java.rmi.RemoteException, javax.ejb.CreateException;
    }
    interface <Bean>Local extends javax.ejb.EJBLocalObject {
        public <beanService>;
    }
    interface <Bean>LocalHome extends javax.ejb.EJBLocalHome {
        <Bean>Local create() throws javax.ejb.CreateException;
    }
}

aspect HelloEJB extends EJB {
    class Hello plays Bean {
        String hello(String name) plays beanService {
            return "Hello "+name+"!";
        }
    }
}
```

The `EJB` aspect definition declares two roles and provides skeletons for all the interfaces needed to define an EJB. Then, in order to define a simple EJB, we only need to extend the `EJB` aspect with a concrete aspect (in the example above, the `HelloEJB` aspect), and define the class that plays the `Bean` role, and the method that plays the `beanService` role.

References

1. Hannemann, J., Kiczales, G.: Design pattern implementation in Java and AspectJ. In: Proceedings of OOPSLA'02. (2002) 161–173
2. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J., Irwin, J.: Aspect-oriented programming. In: Proceedings of ECOOP'97. Number 1241 in Lecture Notes in Computer Science, Springer Verlag (1997) 220–242
3. AOSA: Aspect Oriented Software Development (2003) `http://www.aosd.net`.
4. Lehman, M.M., Ramil, J.F.: Rules and tools for software evolution planning and management. Annals of Software Engeneering **11** (2001) 15–44
5. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Elements of Reusable Object-Oriented Software. Addison Wesley (1995)
6. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, M.: An overview of AspectJ. In: Proceedings of ECOOP'01. Volume 2072 of Lecture Notes in Computer Science., Springer Verlag (2001) 327–353
7. Noda, N., Kishi, T.: Implementing design patterns using advanced separation of concerns. In: Workshop on Advanced Separation of Concerns in Object Oriented Systems - OOPSLA'01. (2001)
8. Stroustrup, B.: The C++ Programming Language. second edn. Addison Wesley (1991)

9. Bracha, G., Odersky, M., Stoutamire, D., Wadler, P.: Making the future safe for the past: Adding genericity to the Java programming language. In Chambers, C., ed.: Proceedings of OOPSLA'98, Vancouver, BC, ACM SIGPLAN Notices (1998) 183–200
10. Odersky, M., Wadler, P.: Pizza into Java: Treanslating theory into practice. In Jordan, N.D., ed.: Proceedings of POPL'97, Paris, France, ACM, ACM Press (1997) 146–159
11. Myers, A., Bank, J., Liskov, B.: Parameterized types for Java. In Jordan, N.D., ed.: Proceedings of POPL'97, Paris, France, ACM, ACM Press (1997)
12. Agesen, O., Freund, S.N., Mitchell, J.C.: Adding type parameterization to the Java programming language. In Bloom, T., ed.: Proceedings OOPSLA'97, Atlanta, Georgia, ACM Press (1997)
13. Cartwright, R., Steele, G.L.: Compatible genericity with runtime-types for the Java programming language. In Chambers, C., ed.: Proceedings of OOPSLA'98, Vancouver, BC, ACM SIGPLAN Notices (1998)
14. Bracha, S.L.G.: JSR 14: Add generic types to the Java programming language (2001) `http://jcp.org/en/jsr/detail?id=14`.
15. Torgensen, M., Hansen, C.P., Ernst, E., von der Ah, P., Bracha, G., Gafer, N.: Adding wildcards to the Java programming language. In: Proceedings of the 19th ACM Symposium on Applied Computing. (2004)
16. Hanenberg, S., Unland, R.: Parametric introductions. In Akşit, M., ed.: Proc. 2nd Int' Conf. on Aspect-Oriented Software Development (AOSD-2003), ACM Press (2003) 80–89
17. Ernst, E.: Family polymorphism. In Knudsen, L., ed.: Proceedings of ECOOP'01. Volume 2072 of LNCS., Springer Verlag (2001) 303–326
18. Loughran, N., Rashid, A., Zhang, W., Jarzabek, S.: Support product line evolution with framed aspects. In: Proceedings of the Third AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS), Lancaster, England (2004)