Pattern Transformation for Two-Dimensional Separation of Concerns

Xiaoqing Wu, Barrett R. Bryant and Jeff Gray
Department of Computer and Information Sciences
The University of Alabama at Birmingham
Birmingham, AL 35294-1170, USA
{wuxi, bryant, gray}@cis.uab.edu

Marjan Mernik
Faculty of Electrical Engineering and Computer Science
University of Maribor
2000 Maribor, Slovenia
marjan.mernik@uni-mb.si

ABSTRACT
Design patterns are utilized in software development to decouple individual concerns, so that a change in a design decision is isolated to one location of the code base. However, multi-dimensional concerns exist in software development and therefore no single design pattern offers a panacea toward addressing problems of change evolution. By analyzing the matrix of concerns during the software development process and utilizing transferable aspect-orientation and object-orientation, a pattern transformation based two-dimensional separation of concerns is described, which integrates the benefits derived from the Inheritance pattern and several GoF patterns. An example implementation is shown using Java and AspectJ.

1. INTRODUCTION
One general intention of design patterns is to decouple individual concerns, so that a change in a design decision is isolated to one location of the code base. Each design pattern is designed to facilitate one kind of change, i.e. changes in one dimension. However, software evolution can happen in multiple dimensions [1] and each dimension has its own best-fit modularization requirements. Therefore, none of the design patterns is a panacea to fulfill the multi-dimensional evolution needed during software development.

Aspect-Oriented Programming (AOP) [2] provides special language constructs that modularize concerns which crosscut conventional program structures (e.g., class hierarchies of object-oriented programs). This offers a second dimension for software modularization besides object-orientation. Except for the Inheritance pattern, most object-oriented design patterns (e.g., Visitor, Mediator, Abstract Factory) are generally defined as collaborations between several objects, which emerge as crosscutting concerns. Applicability of AOP toward modularizing object-oriented design patterns has been heavily researched [3, 4].

Workshop on Modeling and Analysis of Concerns in Software (MACS 2005)
16 May 2005, St. Louis, MO, USA

This paper explores two-dimensional separation of concerns in software development and demonstrates the interchangeability between the object-oriented Inheritance pattern [5] and the aspect-oriented implementation of several Gang-of-Four (GoF) patterns [6] based on the technique of pluggable aspects. The pattern transformation approach combines object-oriented design pattern principles with aspect orientation and leads to a two-dimensional approach toward software evolution, which highlights the benefits derived from the Inheritance pattern and GoF patterns, while reducing their limitations.

The next section explores the usage and limitations of the object-oriented Inheritance pattern and aspect-oriented Visitor pattern. Section 3 explores the idea of two-dimensional separation of concerns using a concern matrix. The pattern transformation approach based on pluggable aspects is detailed in Section 4. A system development example is used in these sections to demonstrate the contribution of this approach. Section 5 generalizes the idea by studying the interchangeability between the Inheritance pattern and other functional patterns. The current status and future work of the approach are discussed in Section 6. Section 7 cites related work, followed by a conclusion in Section 8.

2. DESIGN PATTERN IMPLEMENTATION
In this section, the usage and limitations of the Inheritance and Visitor patterns are explored in building a simple payroll system of a company. Initially, there are three kinds of employees in the system: regulars are paid by weekly wages; executives have a bonus in addition to regular wages; and contractors are paid by hours of work. The system is desired to have basic functionalities to calculate the amount of an employee's wages and export all related employee information. The system is desired to be extended and modified easily.

2.1 Inheritance Pattern Implementation
A straightforward way to build the system in an object-oriented fashion is to create a super class named *Employee* with abstract or concrete operations such as *name* (get the name of a employee), and *wage* (calculate the salary). Afterwards, a subclass for each kind of employee can be defined, which inherits from *Employee* and implements all of its defined virtual methods (as illustrated in Figure 1). This approach is named as the Inheritance pattern in [5], which is a variation of the Interpreter pattern in [2]. An advantage of this approach is that during software evolution, any new kind of employee can be added to the system by creating a new type of

the *Employee* class, without extensive changes to the existing class hierarchy. This type of generalization is a key characteristic of the benefit of using object-oriented design principles. However, an

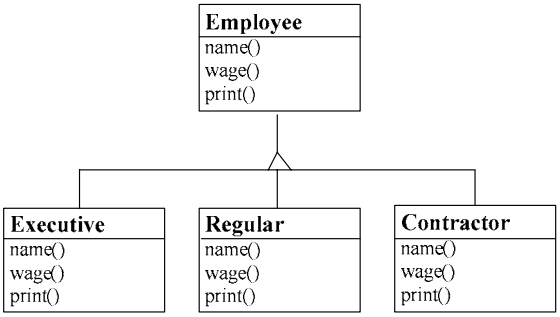

Figure 1. Inheritance pattern implementation

inherent problem in this approach is that each functional operation (defined as a method within each class) crosscuts the various other class boundaries, thereby leading to a system that is hard to comprehend and maintain in terms of system functionality (especially in the presence of deep or broad inheritance hierarchies). Moreover, adding a new operation common to all subclasses requires an invasive change throughout the existing class hierarchy. For example, there could be a need to add a new kind of financial activity. It would be better if each new function could be added separately, and the classes were independent of the operations that apply to them.

2.2 Visitor Pattern Implementation

The Visitor pattern is used to resolve the problem that occurs with the Inheritance pattern. In the Visitor pattern, all the methods pertaining to one functional operation of the element classes are encapsulated into a single visitor class, which can be freely added or deleted from the system. Conventionally, the implementation of the Visitor pattern uses object-oriented principles, as illustrated in Figure 2. The desired operations of the system are produced by invoking iteratively the accept methods within every element class (*Employee* types) throughout the class hierarchy.

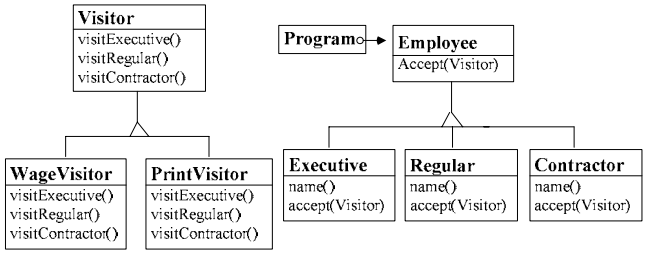

Figure 2. Object-oriented Visitor pattern implementation

Because object-orientation describes a system by a collection of objects rather than a collection of operations, it is clear that object-orientation is not a natural specification of programs based on the Visitor pattern. The complicated implementation of this design pattern introduces a lot of extra code in the element classes and makes the code hard to understand and maintain [4]. Alternatively, observations have indicated that the introduced visitor class has basic AOP characteristics: without them the structure and behavior characteristics are scattered throughout the code base. Aspect-orientation when applied to the Visitor pattern can isolate crosscutting behavior in a more explicit way. For example, using AspectJ [7], each visitor is implemented by an aspect instead of a class and the inter-type declarations in AspectJ allow declaring methods and fields of multiple classes inside one aspect. Therefore, visit operations can be plugged into existing element classes directly and the accept methods originally defined in them are no longer needed (as will be shown later in Listing 2).

However, an important observation for the Visitor pattern is that, since each operation crosscuts each visitor class/aspect, adding a new visitable type to the existing class/aspect hierarchy will cause an invasive change to all of the visitors resulting in a maintenance nightmare. Therefore, no matter if it is an aspect-oriented or a pure object-oriented implementation, the Visitor pattern is applicable only under conditions when the class structure is static and does not change frequently. However, in the case of building the payroll system, it is very likely that new employee types could be added, such as volunteers who have no stipend at all or sales people whose wage is determined by some percentage (e.g. 70% ~200%) of the base wage (depending on his/her quota completion).

3. CLASS-FUNCTION CONCERN MATRIX

It is clear that both Inheritance and Visitor patterns have benefits and limitations in the implementation of the payroll system. The Inheritance pattern assists in flexibly adding new types of employees, but is unsuitable for adding new functionality to the system; the Visitor pattern is useful for adding operations, but inappropriate for adding new employee types. The ideal solution is to combine the synergistic usefulness of the Visitor pattern with the Inheritance pattern while addressing their limitations.

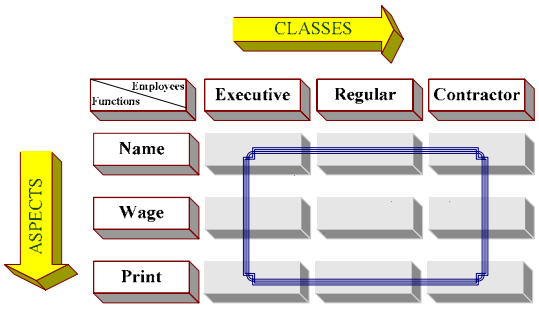

Figure 3. The 2D class-function concern matrix

Reflected by the payroll example, the abstraction of all the necessary constructs in the system can be considered as a two-dimensional (2D) class-function concern matrix [1], shown in Figure 3. Each column represents an employee type, and each row represents the same functionality on all kinds of employees. Each employee has several operations and each operation crosscuts every other employee type. From an orientation point of view, each column represents a class and each row represents an aspect. If all of the artifacts are modularized vertically, an instance of the Inheritance pattern emerges, which could be realized using object-orientation. Correspondingly, if the matrix artifacts are modularized horizontally, an instance of the Visitor pattern emerges, which can be implemented using aspect-orientation.

4. PATTERN TRANSFORMATION

The 2D class-function concern matrix of the payroll system expresses the essence of the development problem, which reflects that an ideal solution should provide two-dimensional separation

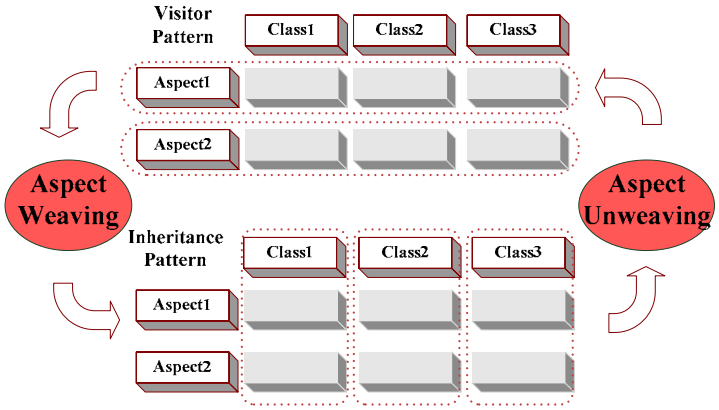

Figure 4. Pattern transformation overview

of concerns [1] and a facility to make the two dimensions transferable. As a result, the pattern transformation based software construction approach is developed in which the Inheritance pattern is implemented using pure Java and the Visitor pattern is implemented using Java and AspectJ. These two patterns are transferable in the development process and only one pattern exists at a time.

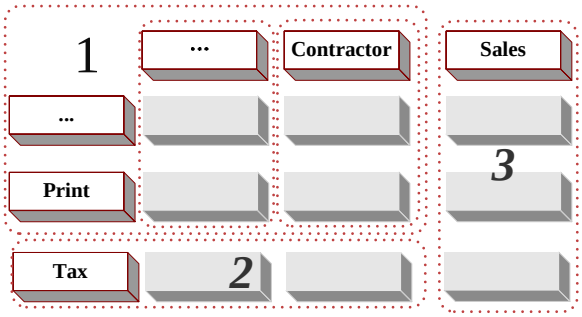

Figure 5. Two-dimensional extension for the payroll system

Because the implementations of the two patterns have the same set of operations and both use Java code in method implementation, the transformation between two patterns is achieved by relocation of all the methods, i.e. from AspectJ aspects to Java classes (aspect weaving) or from Java classes to AspectJ aspects (aspect unweaving). The whole software development paradigm is outlined by the following steps (illustrated in Figure 4) and the aspect weaving and unweaving implementation is described in Section 6.

1. Initially use the Inheritance pattern to implement the system as in Figure 1.
2. Once new functional behaviors need to be added or old functions need to be changed in the system, transform the Inheritance pattern to the Visitor pattern by unweaving the operation methods of each class into individual aspect specifications (the transformation result is shown in the upper part of Figure 4), and then change the operations in a visitor aspect or add new visitor aspects.
3. Once new element classes need to be added, transform the Visitor pattern to the Inheritance pattern by weaving the operations in each aspect into the corresponding class. After weaving, since no more aspects are needed, all the aspect specifications become empty (the transformation result is shown in the lower part of Figure 4), and add the new element classes using the Inheritance pattern.

```
a)   class Regular extends Employee{
b)     public Regular(String name, String ssn, double wage){
c)       super(name, ssn);
d)       this.wage = wage;
e)     }
f)     public double wage(){
g)       return wage;
h)     }
i)     public void print(){
j)       System.out.print ("Regular: ");
k)       super.print();
l)     }
m)   }
```

Listing 1. Java class for *Regular*

The whole approach is illustrated by using the payroll system example. Initially, the class-function matrix for this particular system is composed of 4 classes (including super class *Employee*) and 3 functions (i.e., *name*, *wage* and *print*) that can be applied to those classes. The system is created by an Inheritance pattern as shown in Part 1 of Figure 5. A Java implementation for the *Regular* class is shown in Listing 1. Suppose a new operation needs to be added to the system for calculating each employee's tax payment. For illustration purposes, we assume that tax paid by executives, regulars and contractors are 30%, 25%, 20%, respectively, of their total salary. To update this change, the implementation will be changed to the Visitor pattern to modularize the matrix horizontally such that a new visitor aspect *Tax* (Part 2 of Figure 5) can be easily added to implement the new financial operation. Because the new aspect is cleanly separated

```
1   aspect Tax {
2     public abstract void Employee.tax();
3     public double Executive.tax(){
4       return wage() * 30%;
5     }
6     public double Regular.tax(){
7       return wage() * 25%;
8     }
9     public double Contractor.tax(){
10      return wage() * 20%;
11    }
12  }
```

Listing 2. AspectJ specification for *Tax*

from the generated node classes, there is no single manual change required inside each class. An AspectJ implementation for the *Tax* aspect is shown in Listing 2 (the wage returned by the method *wage()* is pre-taxed).

Suppose that the system needs to take in new kinds of employees, such as a sales person whose wage is determined by the base wage × (his/her quota completion percentage + 40%). If the sales person completes his/her full quota, he/she will be paid by 140% of the base wage. In order to modularize the class-function concern matrix in a vertical way to facilitate adding a new element class, each operation of a specific aspect is weaved into the class it belongs to and implements an instance of the Inheritance pattern. For example, after the weaving process, the new *Regular* class is shown in Listing 3, where the code in bold represents the new method weaved from the aspect *Tax*. To update the change, new class *Sales* is generated and functional operations are added manually to the class without changing the existing class structure. As in part 3 of Figure 5, the new class is written in the same format as the existing ones in order to enable the possible weaving and unweaving process in the later phases.

```
1.    class Regular extends Employee{
2.      public Regular(String name, String ssn, double wage){
3.        super(name, ssn);
4.        this.wage = wage;
5.      }
6.      public double wage(){
7.        return wage;
8.      }
9.      public void print(){
10.       System.out.print ("Regular: ");
11.       super.print();
12.     }
13.     public double tax(){
14.       return wage() * 25%;
15.     }
16.   }
```

Listing 3. The class *Regular* after weaving

5. INHERITANCE PATTERN VS. OTHER PATTERNS

The Visitor pattern is designed to facilitate the changes that the traditional Inheritance pattern is not able to address. Other examples of GoF patterns that exhibit this same property include the Abstract Factory pattern, the Observer pattern, and the Mediator pattern. The common property for these patterns is that they are all used in the case that there are multiple behaviors that crosscut multiple subject classes. The purpose of these patterns is to extract the same kind of functional behavior from the different classes and encapsulate the behavior as an isolated class (visitor, abstract factory, observer, and mediator). However, due to the same reason as seen in the Visitor pattern, the Abstract Factory, Observer, and Mediator patterns all have drawbacks in adding new kinds of subject classes. For example, the Abstract Factory pattern facilitates adding new kinds of factories, but has difficulty in supporting new products; i.e., once the new products are added, invasive change will crosscut all the affected factory classes. Likewise, the Observer pattern facilitates adding observers, but is unsuitable for adding subjects; Mediator facilitates adding mediators, but hampers adding colleagues when multiple mediators exist. Alternatively, using the Inheritance pattern, which encapsulates all the related operations of a subject class inside the class, can solve the drawbacks of these patterns.

The 2D class-function concern matrix shown in Figure 3 can also be used in analyzing the relationship between the Inheritance pattern and the Abstract Factory, Observer, or Mediator patterns. Each column of the matrix represents a different subject class, and each row represents a factory, an observer or a mediator. The transformation ability between all these patterns is illustrated in Figure 6.

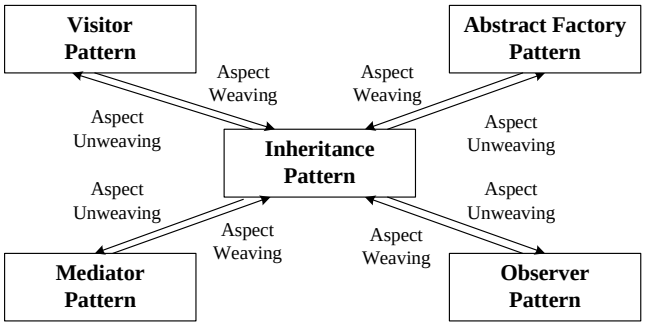

Figure 6. Transformation relationship between Inheritance pattern and other patterns

6. CURRENT STATUS AND FUTURE WORK

AspectJ is used for the aspect specification in this pattern transformation approach. The Java-based syntax of AspectJ enables each operation method to be pluggable between Java classes and AspectJ aspects. However, aspect weaving in AspectJ occurs at the byte code level without the availability of the transformed Java source code. Moreover, there is no way in AspectJ to unweave the operations from Java classes (e.g. Lines 13-15 of Listing 3) to their corresponding aspects (e.g. Lines 7-9 of Listing 2). To overcome these constraints, a program transformation system was used additionally (i.e., the Design Maintenance System (DMS) [8]) to perform source-to-source aspect weaving and unweaving when pattern transformation is needed. The implementation is detailed in [9] and is not repeated here due to space limitation.

The aspect unweaving is currently based on matching the exact operation names (i.e., the operations from different classes), but sharing the same function name will be weaved together as an aspect. More complex software development based on this approach raises several practical problems. For example, one visit operation of a single element class may not be well captured by one visitor function, and sometimes new attributes besides functions need to be introduced in an aspect. After weaving into classes, these additional functions and attributes can not be unweaved easily as aspects because their name could be different, even though they belong to the same concern and should be put into the same aspect. One possible solution is to add the corresponding aspect name as a prefix to all the operations and attribute names during the aspect weaving process. Therefore, the constructs that belong to the same functional behavior can be easily identified at the time of unweaving.

The aspect specification supported in the current approach is only for inter-type declarations. More general join point models (such

as *before()* and *after()*) and their related weaving and unweaving issues should be addressed in the future work, which will greatly generalize the usage of the approach.

In addition to the sample payroll system introduced in this paper, the Inheritance pattern and Visitor pattern transformation technique has already been used successfully in the field of compiler design [9]. Similar analysis has been done on pattern transformation of Inheritance-Abstract Factory, Inheritance-Mediator and Inheritance-Observer. More transformable design patterns are under investigation.

7. RELATED WORK

The closest related work was described in [10], which also concerns about AOP's problem in the case that one aspect crosscuts classes. The provided solution is based on two alternative views of the same program written in the Decal language, allowing developers to edit the program either as decomposed classes or as decomposed modules that crosscut classes. In our approach, the problem is solved by design pattern transformation without abandoning AOP and well-developed programming languages such as Java and AspectJ.

Several papers have mentioned the use of AOP as an approach in design pattern implementation. Hannemann and Kiczales use Java and AspectJ to implement all 23 design patterns in [3] and illustrate implementation details using the Observer pattern as an example. Hachani and Bardou [4] further emphasize implementation of the Visitor pattern using AspectJ. The benefits of using aspect-oriented techniques are described in both of these works. However, a major drawback of the Visitor pattern still remains in the resulting implementation of [4].

The drawbacks of the Inheritance and Visitor patterns are discussed in [5] and the author has claimed that TreeCC can be a better alternative to both of these patterns. However, the essence of TreeCC is still aspect-oriented visitors with strongly typed properties. It can not solve the major problem associated with the Visitor pattern when new nodes are added to an existing node structure.

Tarr et al. first introduced the concept of Multi-Dimensional Separation of Concerns (MDSOC) [1], which is implemented using hyperspaces that allow developers to identify explicit concerns and dimensions, and align units according to concerns. A tool supporting hyperspaces in Java, called Hyper/J [11], was developed, where a system can be composed in many ways from the hypermodules. Each hypermodule specifies a set of hyperslices and each hyperslice addresses a particular concern.

Our contribution differs from the above approaches in that we not only use design patterns and aspect-oriented techniques to implement the two dimensional separation of concerns in software evolution and isolate the crosscutting concerns, but also make patterns and concerns interchangeable to adapt to the various development needs. We simplify the complexity of MDSOC by only focusing on the two orthogonal dimensions and use a straightforward aspect weaving and unweaving approach to modularize different dimensions of concerns.

8. CONCLUSION

There are always multi-dimensional concerns in software development. No single design principle or pattern offers a panacea toward addressing problems of change evolution. Transformation techniques applied to design patterns offer an alternative to alleviating this problem. This paper analyzed the essence of the two-dimensional concern matrix and presented a pattern transformation approach for software evolution in two dimensions using object-orientation and aspect-orientation. The implementation of a simple payroll system and its possible extension was shown using Java and AspectJ. Due to space restrictions, several implementation details are omitted in this paper. Interested readers may refer to the source code at the project web site (http://www.cis.uab.edu/softcom/cde) for more implementation details.

9. REFERENCES

[1] P. Tarr, H. Ossher, W. Harrison, and S. Sutton. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proc. Int. Conf. Software Engineering (ICSE)*, 1999, pp. 107-119.

[2] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proc. 11th European Conf. Object-Oriented Programming (ECOOP)*, Springer-Verlag, LNCS 1241, 1997, pp. 220-242.

[3] J. Hannemann and G. Kiczales. Design Pattern Implementation in Java and AspectJ. In *Proc. Object-Oriented Programming, Systems, and Applications (OOPSLA)*, 2002, pp. 161–173.

[4] O. Hachani and D. Bardou. Using Aspect-Oriented Programming for Design Patterns Implementation. In *Proc. Workshop Reuse in Object-Oriented Information Systems Design*, 2002.

[5] R. Weatherley. TreeCC: An Aspect-Oriented Approach to Writing Compilers. http://www.southern-storm.com.au/treecc.html.

[6] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[7] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *Proc. 15th European Conf. on Object-Oriented Programming (ECOOP)*, Springer-Verlag, LNCS 2072, 2001, pp. 327–355.

[8] I. Baxter, C. Pidgeon, and M. Mehlich. DMS: Program Transformation for Practical Scalable Software Evolution. In *Proc. Int. Conf. Software Engineering (ICSE)*, 2004, pp. 625-634.

[9] X. Wu, S. Roychoudhury, B. Bryant, J. Gray, and M. Mernik. A Two-Dimensional Separation of Concerns for Compiler Construction. In *Proc. ACM Symposium on Applied Computing (SAC)*, 2005, pp.1365-1369.

[10] D. Janzen and K. D. Volder. Programming With Crosscutting Effective Views. In *Proc. 18th European Conf. on Object-Oriented Programming (ECOOP)*, 2004, pp. 195–218.

[11] HyperJ website: http://www.alphaworks.ibm.com/tech/hyperj