Pattern Transformation for Two-Dimensional Separation of Concerns

Xiaoqing Wu (Advisor: Barrett R. Bryant)
Department of Computer and Information Sciences – The University of Alabama at Birmingham
http://www.cis.uab.edu/cde Email: wuxi@cis.uab.edu

Software Composition & Modeling Laboratory SoFTCoM

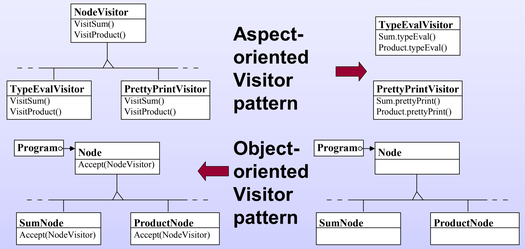
Department of Computer & Information Sciences

NO DESIGN PATTERN IS A PANACEA

- 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 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.

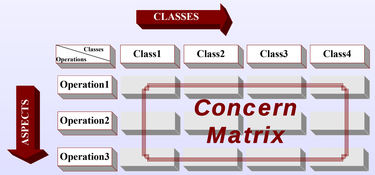
ASPECT-ORIENTED DESIGN PATTERNS

Most object-oriented design patterns (e.g., Visitor, Mediator, Abstract Factory) are generally defined as collaborations between several objects, which emerge as crosscutting concerns. Applying AOP toward modularizing object-oriented design patterns provides a more straightforward implementation strategy.



TWO-DIMENSIONAL CONCERN MATRIX

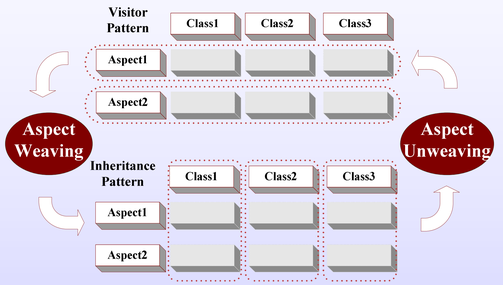
For some software systems, the abstraction of all the necessary constructs in the system can be considered as a two-dimensional subject-function concern matrix.



Vertical modularization
→ each column = a class
→ Inheritance Pattern

Horizontal modularization
→ each row = an aspect
→ Visitor, Mediator,…,Pattern

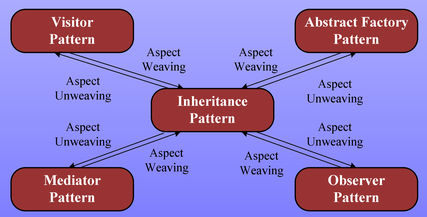
PATTERN TRANSFORMATION APPROACH



When the two-dimensional concern matrix exists, pattern transformation can be implemented by source code transformation between object-orientation and aspect-orientation, based on the pluggable aspects.

DESIGN PATTERN PAIRS

The Visitor, Abstract Factory, Observer, and Mediator patterns all have drawbacks in adding new kinds of subject classes. Based on a similar 2D class-aspect concern matrix, the problem can be eliminated by transforming those patterns back and forth to the Inheritance pattern to accommodate change decisions.



PATTERN TRANSFORMATION APPLICATION

The Inheritance pattern and Visitor pattern transformation scheme has also been utilized successfully in the field of compiler design.

AST nodes are modularized as subject classes.

	Expression	Term	Sum	Difference
ValueEval				
TypeEval				
PrettyPrint				

Each phase of semantic analysis is encapsulated as an aspect.

The transformation between the Inheritance and Visitor patterns enables the compiler to evolve along two dimensions: defining new abstract syntax tree (AST) classes, or adding new operations.

CASE STUDY & TRANSFORMATION DETAILS

System description: Consider to build a simple payroll system of a company. Initially, there are three kinds of employees in the system: regulars, executives, and contractors. Each type of employee has different wage calculation function and output information.

Employees / Functions	Executive	Regular	Contractor
Name			
Wage			
Print			

System evolution is performed along two dimensions:

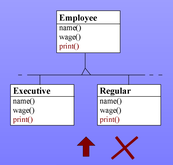
Changing or defining type of employees (e.g. Sales)

Changing or adding operations (e.g. Tax)

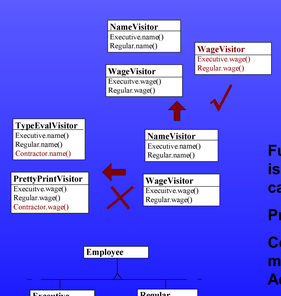
A pure object-oriented approach based on entities

Pros: Any new kind of employee can be added to the system by creating a new type of the Employee class.

Cons: The functional operations crosscut the various other class boundaries. Adding a new operation common to all subclasses is difficult.



Inheritance Pattern



Visitor Pattern

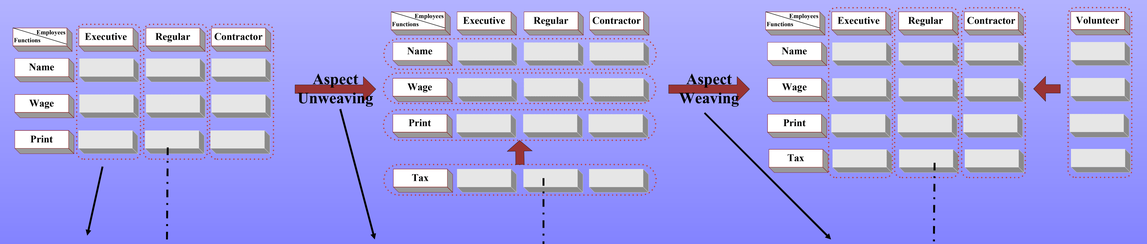
Functional operation oriented. Visitor is separated with subject classes and can be freely added or deleted.

Pros: Adding new operations is easy.

Cons: Each operation crosscuts multiple visitor classes/aspects. Adding a new visitable type is hard.

Pattern Transformation

The underlying technology to realize the pattern transformation is aspect weaving and unweaving between object-orientation and aspect-orientation. Aspect weaving for inter-type declarations simply plugs those fields and methods defined in the aspect back to their corresponding classes. The aspect's own fields/methods will be moved to the super class (e.g., Employee) as static members during the weaving process. Aspect unweaving for inter-type declarations and aspects' own fields/methods simply undo the above processes. The aspect weaving and unweaving for join point-advice are not supported yet.



1. Initially use either the Inheritance pattern or the Visitor pattern to build the system.

2.Once new functional behaviors need to be added or old functions need to be changed, transform the Inheritance pattern to the Visitor pattern (if it is not) by unweaving the operation methods of each class into individual aspect specifications, and then change the operations inside a visitor aspect or add new visitors to the system.

3.Once new subject classes need to be added, transform the Visitor pattern to the Inheritance pattern (if it is not) by weaving the operations in every aspect into the corresponding class, and add the new subject classes using the Inheritance pattern.

```
class Executive extends Employee{
  class Contractor extends Employee{
    class Regular extends Employee{
      public Regular(String name, String ssn, double wage){
        super(name, ssn);
        this.wage = wage;
      }
      public double wage(){ return wage; }
      public void print(){
        System.out.print ("Regular: ");
        super.print();
      }
    }
  }
}
```

```
aspect Name {
aspect Wage {
aspect Print {
aspect Tax {
  public abstract void Employee.tax();
  public double Executive.tax(){
    return wage() * 30%;
  }
  public double Contractor.tax(){
    return wage() * 20%;
  }
  public double Regular.tax(){
    return wage() * 25%;
  }
}
```

```
class Sales extends Employee{
  class Executive extends Employee{
    class Contractor extends Employee{
      class Regular extends Employee{
        public Regular(String name, String ssn, double wage){
          super(name, ssn);
          this.wage = wage;
        }
        public double wage(){      return wage;   }
        public void print(){
          System.out.print ("Regular: ");
          super.print();
        }
        public double tax(){
          return wage() * 25%;
        }
      }
```