Replace Conditional with Visitor

Ivan Mitrovic
Ivan.Mitrovic@S1.com

You have an "aggressive" conditional that chooses different behaviour depending on the type of an object and repeats itself in a large number throughout the code.

Create concrete Visitor per conditional.
Move each leg of the conditional to a visit method of the concrete Visitor.
Make subclasses accept Visitor. Make Visitor visit subclasses.

Motivation

If the number of switch conditional blocks based on the same type code becomes large and the number of type codes is unlikely to grow, then creating subclasses based on each type code may lead to a creation of low cohesive subclasses with a lot of unrelated methods.
We can imagine a situation where a large number of switch blocks based on the same type code is scattered throughout the code. Replacing each switch block with an abstract method of a superclass followed by the concrete method implementation in subclasses could make subclass or State (Strategy) low cohesive (*Replace Type Code with subclass*[1] or *Replace Type Code with State / Strategy*[1] and, then *Replace Conditional with Polymorphism*[1] refactoring pattern). Subclass could contain a large number of unrelated methods where each switch block is replaced by one of those methods. Moreover, if the number of "switch like situations" tends to grow in the future, each subclass has to be changed, as a new method has to be added in order to replace a possible switch block. Beside the fact that this violates the Open/Closed principle, we can end up in a situation that we can't even come up with a good name for such a low cohesive class. Getting in a situation where the class can't be easily named means that it has to be refactored.
Replace Conditional with Visitor can be a solution for that problem as this refactoring really doesn't care how many switch conditionals based on the same type code you have in the system. Of course, you have to come up with a conclusion if the method would be in a subclass or in a concrete Visitor. You'll probably have to exercise *Move Method*[1] until you achieve nirvana.

This is actually an implementation of Visitor Design Pattern (GOF).

Mechanics

Before you can begin with *Replace Conditional with Visitor* you need to have the necessary inheritance structure. You may already have this structure from previous refactorings. If you don't have this structure you need to create it applying *Replace Type Code with Subclasses*[1] or *Replace Type Code with State/Strategy*[1]. In fact, prerequisites for *Replace Conditional with Visitor* are the same ones as for *Replace Conditional with Polymorphism*[1]. If you are not familiar with *Replace Conditional with Polymorphism*[1], please study it first, as *Replace Conditional with Visitor* addresses the same problem *Replace Conditional with Polymorphism*[1] does by fully accompanying it and overcoming its limitations where a large number of conditionals are present in the code. In the simpler situation where the number of conditionals is not large (whatever large means to you) and is unlikely to grow, or, if created

[1] Refactoring
Improving the Design of Existing Code
Martin Fowler
AW, 1999.

subclasses are not low cohesive, *Replace Conditional with Polymorphism*[1] is probably a better solution. Even then, you are encouraged to freely mix those two refactorings.

- Create necessary Visitor inheritance structure.
- Provide accept method in the subclasses. Make the original accept method abstract.
- If the conditional statement is one part of a larger method, take apart the conditional statement and use *Extract Method*[1].
- Create an instance of concrete Visitor that encapsulates the logic of the conditional.
- Supply concrete Visitor with all parameters necessary to process that logic (usually by passing them to a concrete Visitor's constructor). If necessary, apply other refactorings to make parameters reachable from a Visitor.
- Visit each subclass, one by one, by a Visitor and let the double dispatch do the magic.
- Get result from a Visitor (if necessary).
- Test. Repeat the process for each leg of the conditional.
- Repeat the process for another conditional.

Example

I'll use similar example and the similar inheritance structure as an example from the Refactoring[1] book. I'm using the classes after using *Replace Type Code with Subclasses*[1], so the objects actually look the same as in Figure 1 (you can see the example for *Replace Type Code with Subclasses*[1] from the Refactoring[1] book to see how we got here):

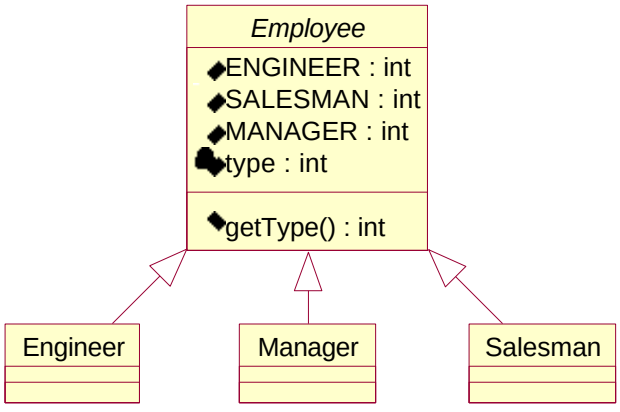

Figure 1

Conditional to be refactored:

```
class EmployeeSorter…
	private Vector engineers;
	private Vector salesmen;
	private Vector managers;
	…
	void sortEmployee(Employee emp) {
		switch(emp.getType()) {
			case Employee.ENGINEER:
				engineers.addElement(emp);
				break;
			case Employee.SALESMAN:
				salesmen.addElement(emp);
				break;
			case Employee.MANAGER:
				managers.addElement(emp);
				break;
			default:
				throw new IllegalArgumentException("Incorrect Employee");
		}
	}
	…
```

Before I can begin with *Replace Conditional with Visitor* I must create the necessary Visitor inheritance structure:

```
public interface Visitor {
	public void visitEngineer(Engineer engineer);
	public void visitSalesman(Salesman salesman);
	public void visitManager(Manager manager);
}
```

Employee type hierarchy has to be visited by a Visitor so I provided the necessary infrastructure. I also implement accept method in each subclass of Employee while I live the original method abstract.

```
class Employee …
	…
	public abstract void accept(Visitor visitor);
	…

class Engineer…
	…
	public void accept(Visitor visitor) {
		visitor.visitEngineer(this);
	}
	…

class Salesman…
	…
	public void accept(Visitor visitor) {
		visitor.visitSalesman(this);
	}
	…

class Manager…
	…
	public void accept(Visitor visitor) {
		visitor.visitManager(this);
	}
	…
```

After everything was prepared, I'm ready to create a concrete Visitor that would replace conditional. As I remove leg by leg of the conditional, I will implement visit method in a concrete Visitor that corresponds to each subclass:

```
import java.util.*;

public class EmployeeSortVisitor implements Visitor {
	private Vector engineers;
	private Vector salesmen;
	private Vector managers;

	public EmployeeSortVisitor() {
		engineers = new Vector();
		salesmen = new Vector();
		managers = new Vector();
	}

	public void visitEngineer(Engineer engineer) {
		engineers.addElement(engineer);
	}

	public void visitSalesman(Salesman salesman) {
		throw new RuntimeException("Shouldn't be here");
	}

	public void visitManager(Manager manager) {
		throw new RuntimeException("Shouldn't be here");
	}

	public Enumeration engineers() {
		return engineers.elements();
	}
}
```

I can now apply refactoring on a conditional:

```
 class EmployeeSorter…
	private Vector engineers;
	private Vector salesmen;
	private Vector managers;
	private EmployeeSortVisitor visitor = new EmployeeSortVisitor();
	…
	void sortEmployee(Employee emp) {
		switch(emp.getType()) {
			case Employee.ENGINEER:
				emp.accept(visitor);
				break;
			case Employee.SALESMAN:
				salesmen.addElement(emp);
				break;
			case Employee.MANAGER:
				managers.addElement(emp);
				break;
			default:
				throw new IllegalArgumentException("Incorrect Employee");
		}
	}
	…
```

I compile and test. If everything went right I can replace another leg of conditional:

```
import java.util.*;

public class EmployeeSortVisitor implements Visitor {
	private Vector engineers;
	private Vector salesmen;
	private Vector managers;

	public EmployeeSortVisitor() {
		engineers = new Vector();
		salesmen = new Vector();
		managers = new Vector();
	}

	public void visitEngineer(Engineer engineer) {
		engineers.addElement(engineer);
```

```
	}

	public void visitSalesman(Salesman salesman) {
		salesmen.addElement(salesman);
	}

	public void visitManager(Manager manager) {
		throw new RuntimeException("Shouldn't be here");
	}

	public Enumeration engineers() {
		return engineers.elements();
	}

	public Enumeration salesmen() {
		return salesmen.elements();
	}
}

class EmployeeSorter…
	private Vector engineers;
	private Vector salesmen;
	private Vector managers;
	private EmployeeSortVisitor visitor = new EmployeeSortVisitor();

	…
	void sortEmployee(Employee emp) {
		switch(emp.getType()) {
			case Employee.ENGINEER:
			case Employee.SALESMAN:
				emp.accept(visitor);
				break;
			case Employee.MANAGER:
				managers.addElement(emp);
				break;
			default:
				throw new IllegalArgumentException("Incorrect Employee");
		}
	}
	…
```

Compile and test. I repeat until all the legs of the conditional are replaced:

```
import java.util.*;

public class EmployeeSortVisitor implements Visitor {
	private Vector engineers;
	private Vector salesmen;
	private Vector managers;

	public EmployeeSortVisitor() {
		engineers = new Vector();
		salesmen = new Vector();
		managers = new Vector();
	}

	public void visitEngineer(Engineer engineer) {
		engineers.addElement(engineer);
	}

	public void visitSalesman(Salesman salesman) {
		salesmen.addElement(salesman);
	}

	public void visitManager(Manager manager) {
		managers.addElement(manager);
	}

	public Enumeration engineers() {
		return engineers.elements();
	}

	public Enumeration salesmen() {
```

```
                return salesmen.elements();
        }

        public Enumeration managers() {
                return managers.elements();
        }
}

class EmployeeSorter…
        private Vector engineers;
        private Vector salesmen;
        private Vector managers;
        private EmployeeSortVisitor visitor = new EmployeeSortVisitor();
        …
        void sortEmployee(Employee emp) {
                switch(emp.getType()) {
                        case Employee.ENGINEER:
                        case Employee.SALESMAN:
                        case Employee.MANAGER:
                                emp.accept(visitor);
                                break;
                        default:
                                throw new IllegalArgumentException("Incorrect Employee");
                }
        }
        …
```

After compilation and test I am free to remove the switch block:

```
class EmployeeSorter…
        private Vector engineers;
        private Vector salesmen;
        private Vector managers;
        private EmployeeSortVisitor visitor = new EmployeeSortVisitor();
        …
        void sortEmployee(Employee emp) {
                emp.accept(visitor);
        }
        …
```

I can use a result obtained from the Visitor wherever needed:

```
class EmployeeSorter…
        private Vector engineers;
        private Vector salesmen;
        private Vector managers;
        private EmployeeSortVisitor visitor = new EmployeeSortVisitor();

        …
        void sortEmployee(Employee emp) {
                emp.accept(visitor);
        }
        …
        //After all Employees are traversed
        printNames(visitor.engineers());
etc.
```

Eeach concrete Visitor is highly cohesive as it has only one responsibility (e.g. EmployeeSortVisitor sorts employees, ToPayVisitor can calculate salaries, HolidayVisitor can calculate holidays etc.) This design is quite flexible and we can create as many Visitors as we want without changing underlying structure. Visitor is, by nature, highly coupled with the hierarchy of objects it has to visit. If that hierarchy is stable and unlikely to change in the future, Visitor becomes a powerful tool and enables you to volatile functions upon hierarchy. This becomes more important if the number of switch conditionals is likely to grow.

Create concrete Visitor for each conditional.