To Be Destructive or Not To Be, That is the Question on Modular Extensions

Shigeru Chiba
The University of Tokyo
Quotes

• Aspect oriented programming is quantification and obliviousness.
  – Robert E. Filman Daniel P. Friedman

• Obliviousness is not mandatory but desirable.
  – Awais Rashid?
AOP functionality (1)

• Obliviousness is useful and practical!
  – An advice can obliviously modify a method.
  • The original source code is not modified at all when the software is extended.

```java
class AddExpr {
    Value eval() { … }
}

Value around(AddExpr ae):
    execution(Value AddExpr.eval())
    && this(ae) {
        if (...) proceed();
        else ... ;
    }
```
AOP functionality (2)

- Limited scope
  - An advice can modify a method call in a body
  - Breaking modularity?

```java
class VarDecl {
    Value init() {
        v = right.eval();
    }
}

class AddExpr {
    Value eval() { ... }
}

aspect Logging {
    before():
        call(void Expr.eval())
        && withincode(* VarDecl.init()) {
            ...
        }
}
```
A new scripting language in two weeks

A new scripting language in two weeks

When I was a student….
I was a server/net. admin. of our lab.

When I was a postdoc….
I was a server/net. admin. of our dept.

When I was a prof….
still server/network admin. of our dept.

Now I’m a dept. chair….
I’m writing PHP for our dept. web site.

A boss in name only?
Shall I change my job?
Éric Tanter said to me.

• If the code in the book is in Scheme, you don’t need obliviousness or AOP.

• … Right. But Scheme also provides “obliviousness” or destructive extension I call.

– The code in my book is in Java.
GluonJ: A reviser

• **Destructive extension**
  – A reviser can add and override a method, and add a field to an existing class.
  • It cannot have an explicit constructor.

```java
class AddExpr {
    Value eval() { ... }
}

class FloatEx revises AddExpr {
    Value eval() {
        if (...) super.eval();
        else ... ;
    }
}
```
GluonJ: A within method

• Limited scope
  – A method may have a predicate.
  – Its method overriding is effective only when it is called from ...

```java
class VarDecl {
    Value init() {
        v = right.eval();
    }
}

class AddExpr {
    Value eval() {
        ...
    }
}

class Log revises AddExpr {
    Value eval()
    within VarDecl.init() {
        ...
    }
}
```
Contextual predicate dispatch

• GluonJ
  – Predicates refers to non-local contexts i.e. within who is a caller.
  • Currently only within is available.
  – to deal with crosscutting concerns

• Original predicate dispatch
  – Predicates refers to only local contexts such as arguments and receiver’s fields
  • for unambiguity and exhaustiveness
• **Non-destructive extension**
  – Both the original and the extension coexist.
  • The source code is not modified as in AOP.

```java
class AddExpr {
    Value eval() { ... }
}

AddExpr e1, e2;
e1 = new AddExpr();
e2 = new FloatEx();

class FloatEx extends AddExpr {
    Value eval() {
        if (…) super.eval();
        else ... ;
    }
}
```
To Be Destructive or Not To Be, That is the Question on Modular Extensions
Abstract Factory pattern or dependency injection

• AOP-like modification by non-destructive extension (= subclassing)

```java
class Factory {
    AddExpr makeAddExpr() {
        return new AddExpr();
    }
}

class FactoryEx extends Factory {
    AddExpr makeAddExpr() {
        return new FloatEx();
    }
}

AddExpr e = factory.makeAddExpr();

class AddExpr {
    Value eval() { ... }
}

class FloatEx extends AddExpr {
    Value eval() {
        if (...) super.eval();
        else ... ;
    }
}
```

Shigeru Chiba, The University of Tokyo
Abstract Factory pattern or dependency injection

- To switch classes, the main method must be modified by hand.

```java
void main(String[] args) {
    factory = new FactoryEx();
    program.start(args);
}
```

Or, another main method must be written from scratch.
Abstract Factory pattern or dependency injection

• To switch classes, the main method must be modified by hand.

```java
void main(String[] args) {
    factory = new FactoryEx();
    program.start(args);
}
```

Or, another main method must be written from scratch.

```java
void main(String[] args) {
    factory = new Factory();
    program.start(args);
}
```
Also,

• Family polymorphism/virtual classes
  
  – are non-destructive
    like Abstract Factory pattern.
Destructive or Non-destructive

- Modification or Another copy
Destructive or Non-destructive

- Modification or Another copy
  - when an intermediate module is modified
Destructive extension

• OK, it’s useful when I want to modify only a piece of code in my program.

• But, I often want to reuse the original code in the same program.
Scope!

- Destructive
  - Always modify

- Conditionally
  - AspectJ’s within, withincode, and cflow
  - GluonJ’s within
  - ContextJ

- Non-destructive
  - Only specific instances
Various kinds of scopes

• In Ruby

```ruby
class Integer
  def div(x)
    # returns an integer
    # returns a rational num.
    end
  end
end
def average(list)
  sum = ...
  sum / list.size
end
class Integer
  def div(x)
    # returns an integer result
    end
  end
end
list = ...
a = average(list)
a2 = a / 10 * 10
```
Reusable destructive extensions

- Module users should specify where they are effective.
  - Module writers should not.

```ruby
class Integer
def div(x)
  # returns an integer
  result
end
end
```

```ruby
def average(list)
  sum = ...
  sum / list.size
end
```

```ruby
class Integer
def div(x)
  # returns a rational num.
  num.
end
end
def average(list)
  sum = ...
  sum / list.size
end
```

```ruby
list = ...
a = average(list)
a2 = a / 10 * 10
```
Mentioning the scope

• At the side of the module user.

• AspectJ
  – abstract pointcut

• Dynamic Aspect-Oriented Programming
  – deploy(...) { ... } in CaesarJ

• Context-Oriented Programming
  – with(...) { ... } in ContextJ
More structural scope

- Method shelters [Akai&Chiba, AOSD’12]
- Method shells [Takeshita&Chiba, SC’13]

- When a `module` is imported,
  - the scope of the destructive extensions in it is **declaratively** specified.
Method shells

[Takeshita&Chiba’13]

• Two kinds of module import
  – Include
  – link

Include put in the same scope

Link makes methods in a different scope visible.

def average(list)
  sum = ...
  sum / list.size
end

class Integer
  def div(x)
    # returns a rational num.
  end
end
def div(x)
  # returns an integer result
end

class Integer
  def div(x)
    # returns an integer result
  end
end

list = ...
a = average(list)
a2 = a / 10 * 10
The semantics of link is complex

• Takeshita’s master thesis in 2014
  – When are scopes switched?

```ruby
class Integer
  def div(x)
    # rational num
    end
  def toInt(r)
    # ...
    end
end

def average(list)
  sum = ...
  sum / list.size
end

list = ...
a = average(list)
a2 = a / 10 * 10
a3 = toInt(a2)
```

All the methods in this scope are visible.
Summary

• To Be Destructive or Not To Be, That is the Question on Modular Extensions

• Destructive extensions Always

• Conditionally ...
  – Specified by extension-users
  – Structural scope e.g. Method Shelters/Shells

• Non-destructive Only specific instances