Aspect-Oriented Design Patterns

And Their Use for Advanced Modularization

Autochthonous – Intrinsic Aspect-Oriented Patterns

- Cannot be used without aspect-oriented technology (aspect oriented world)
  - usually require to employ some of various kinds of weaving

Examples:

- Wormhole
- Worker Object Creation
- Cuckoo's Egg

Task: Run methods in different order according to their first argument after/before certain method is executed

- Delegate such work on worker objects
- Do not modify already provided code

Code that has to be left intact

```
public class C {
  public void m(int n) {
    System.out.println(n);
  }

  public static void main(String[] args) {
    new C().m(4);
    new C().m(7);
    new C().m(3);
    new C().m(5);
    new C().m(1);
  }
}
```

Source: http://www2.fiit.stuba.sk/~vranic/aosd/poznamky/aspekty-aj.pdf

Worker Creation and Its Application

CREATING WORKER

```
m(new Runnable() {
    public void run() {
        . . . // kod ktory sa ma vykonat
    }
});
```

MAKING WORKER WORK

```
m(Runnable o) {
    o.run();
}
```

Source: http://www2.fiit.stuba.sk/~vranic/aosd/poznamky/aspekty-aj.pdf

Worker Object Creation

CONTRADICTING FORCES:

1. A joint point has to be transferred to another context for execution,
2. but without transforming corresponding code.

THEIR RESOLUTION:

Using threads with ability to execute particular method/functionality (proceed() call) after particular joint point is reached/executed.

Worker Object Creation

```
void around(): <pointcut> {
   Runnable worker = new Runnable() {
      public void run() {
         // calling inner function body
         proceed();
      }
   };
   invoke.Queue.add(worker);
}
```

Why autochthonous?

```
void around(): <pointcut> {
   Runnable worker = new Runnable() {
      public void run() {
         // calling inner function body
         proceed();
      }
   };
   invoke.Queue.add(worker);
}
```

Original method is called inside worker after/before specific joint point is reached - NO TANGLED CODE!!!

Worker Object Creation

```
// Worker Object Creation
public aspect OrderCalls {
  PriorityQueue<MCall> calls = new PriorityQueue<>();

  void around(int n): call(void C.m(..)) && args(n) {
    Runnable m = new Runnable () {
      public void run() {
        proceed(n);
      }
    };
    calls.add(new MCall(m, n)); // do prioritneho radu vkladame zachytene volania obalene do struktury MCall
                  // v tomto priklade prioritu urcuje jednoducho argument – dalo by sa aj inak
  }

  after(): execution(void C.main(..)) { // odlozene volania sa realizuju podla priority, ked skonci main()
    while (calls.peek() != null)
      calls.poll().m.run();
  }
}
```

Source: http://www2.fiit.stuba.sk/~vranic/aosd/poznamky/aspekty-aj.pdf

Worker Object Creation

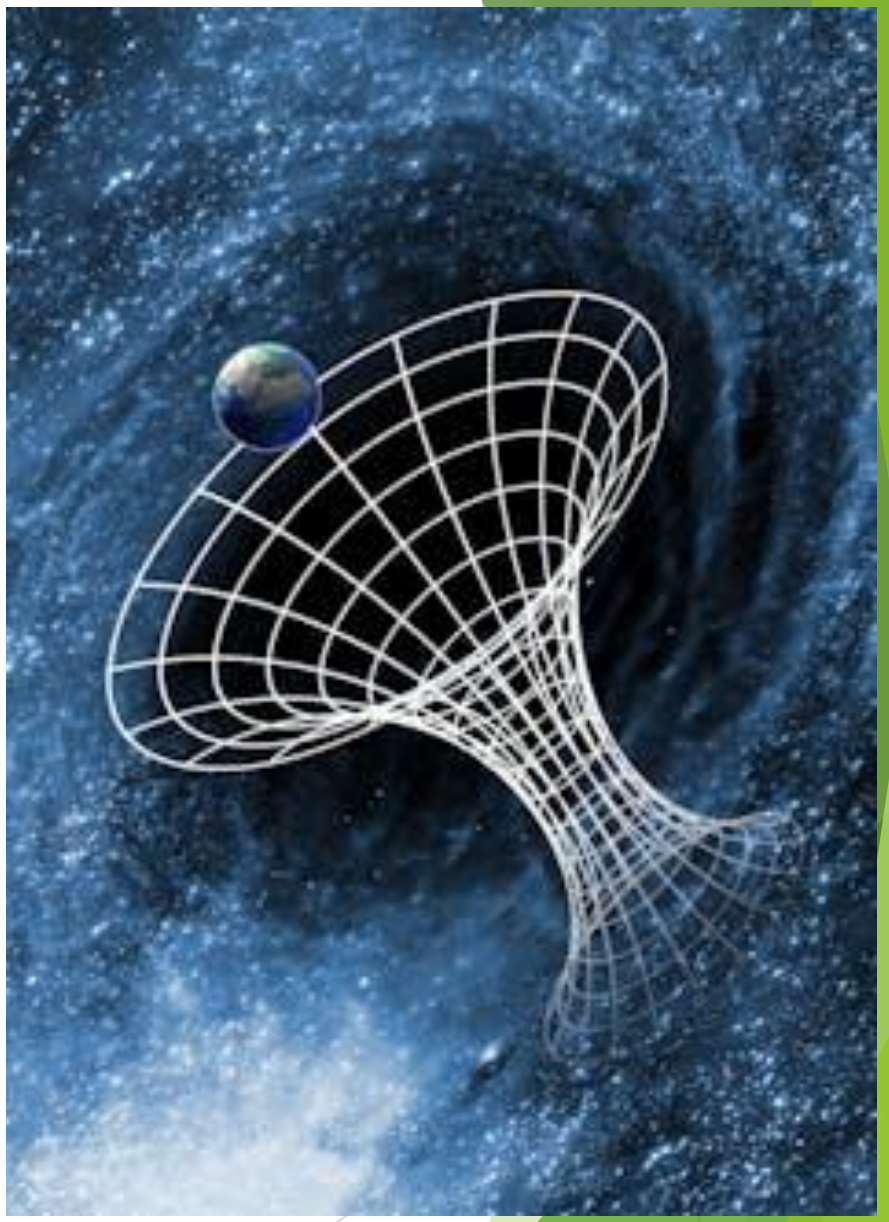
```
class MCall implements Comparable<MCall> {
  Runnable m;
  int n;

  public MCall(Runnable m, int n) {
    this.m = m;
    this.n = n;
  }
  public int compareTo(MCall o) {
    if (this.n > o.n)
      return 1;
    else if (this.n < o.n)
      return -1;
    else
      return 0;
  }
}
```

Source: http://www2.fiit.stuba.sk/~vranic/aosd/poznamky/aspekty-aj.pdf

Wormhole

- As Wormhole to **connect two distinct spaces**:
- **Caller space/concern: <caller context>**
- **Callee space/concern: <callee context>**
  - **BENEFIT:** without need to **extend arguments** in place of these spaces – in place of their **original methods** (implementation of these crosscutting concerns)



Source: https://images.theconversation.com/files/476946/original/file-20220801-62374-uyna4z.jpg?ixlib=rb-4.1.0&q=45&auto=format&w=1000&fit=clip

Task: Solve or Propagate Tasks To Other Employees But Not Propagate Tasks From Authority With ID = 13

```
public class Task {
  public int id;

  public Task( int id) {
    this.id = id;
  }
}

public class Authority {
  public int id;

  public Authority(int id) {
    this.id = id;
  }

  public void setTask(Employee employee, int taskId) {
    employee.doTask(new Task(taskId));
  }
}
```

Source: http://www2.fiit.stuba.sk/~vranic/aosd/poznamky/aspekty-aj.pdf

Source: http://www2.fiit.stuba.sk/~vranic/aosd/poznamky/aspekty-aj.pdf

```
public class Employee {
  public int id;
  public int mood;
  public Simulation simulation;

  public Employee(int id, Simulation simulation) {
    this.id = id;
    this.simulation = simulation;
    mood = new Random().nextInt(3);
  }

  public void solveTask(Task task) {
    System.out.println("Employee " + id + " solves task " + task.id);
  }
  public void tryTask(Task task) {
    System.out.println("Employee " + id + " tries task " + task.id);
    mood = new Random().nextInt(3);
```

```
    yieldTask(task);
  }
  public void yieldTask(Task task) {
    Employee e = simulation.getAnotherEmployee();

    if (e != this) {
      System.out.println("Employee " + id + " yields task " + task.id + " to employee " + e.id);
      e.doTask(task);
    }
  }

  public void doTask(Task task) {
    mood = new Random().nextInt(3);

    switch (mood) {
      case 0: yieldTask(task);
          break;
      case 1: tryTask(task);
          break;
      case 2: solveTask(task);
          break;
    }
  }
}
```

Source: http://www2.fiit.stuba.sk/~vranic/aosd/poznamky/aspekty-aj.pdf

```
public class Simulation {
  public List<Employee> employees = new ArrayList<>();

  public Employee getAnotherEmployee() {
    return employees.get(new Random().nextInt(employees.size()));
  }

  public static void main(String[] args) {
    Simulation simulation = new Simulation();

    for (int i = 0; i < 10; i++)
      simulation.employees.add(new Employee(i, simulation));

    new Authority(1).setTask(simulation.getAnotherEmployee(), 5);
    new Authority(2).setTask(simulation.getAnotherEmployee(), 7);
    new Authority(13).setTask(simulation.getAnotherEmployee(), 99);
  }
}
```

Source: http://www2.fiit.stuba.sk/~vranic/aosd/poznamky/aspekty-aj.pdf

```
public aspect StopPropagatingTasks {
  public int unwantedId = 13;

  pointcut settingTasks(Authority authority):
    execution(void Authority.setTask(..)) && this(authority);

  pointcut propagatingTasks(Employee employee, Task task):
    call(void Employee.yieldTask(..)) && this(employee) && args(task);

  pointcut propagatingTasksByAuthority(Authority authority, Employee employee, Task task):
    propagatingTasks(employee, task) && cflow(settingTasks(authority));

  void around(Authority authority, Employee employee, Task task):
    propagatingTasksByAuthority(authority, employee, task) {
    if (authority.id == unwantedId && employee.mood == 0) {
      System.out.println("Employee " + employee.id + " refuses to propagate task " + task.id);
    } else {
      proceed(authority, employee, task);
    }
  }
}
```

```
public aspect WormHoleAspect() {
    pointcut callerSpace(<caller context>):
        <caller pointcut>;
    pointcut calleeSpace(<callee context>):
        <callee pointcut>;

    pointcut wormhole(
        <caller context>, <callee context>):
        cflow(callerSpace(<caller context>))
        && calleeSpace(<callee context>);
```

Caller space pointcut to capture particular caller join points

Callee space pointcut to capture particular callee join points

Specifying situation where new Concern based on both concerns should be applied

```
void around(
        <caller context>, <callee context>):
    wormhole(
        <caller context>, <callee context>) {
        // implementation
        // of crosscutting concern
    }
}
```

Here the concern demanding the connection of two distinct spaces is implemented in separated aspect – modularized

Wormhole

CONTRADICTING FORCES:

1. The calling object should be known within the context of the method being called,
2. but without transferring it as a parameter.

THEIR RESOLUTION:

Connecting two distinct spaces and resolving crosscutting concern that requires both of them in sparated aspect (original code remains unaffected)

Why autochthonous?

```
public aspect WormHoleAspect() {
    pointcut callerSpace(<caller context>):
            <caller pointcut>;
    pointcut calleeSpace(<callee context>):
            <callee pointcut>;
```

Selects callee and caller join points followed by their connection to solve crosscutting concern using custom advice

```
    pointcut wormhole(
            <caller context>, <callee context>):
cflow(callerSpace(<caller context>))
            && calleeSpace(<callee context>);
```

In object oriented word this can be treated only using function arguments and Possibly tangling concerns

Cuckoo’s Egg

- As Egg similar to other eggs in analogy while **substituting existing functionality under similar/known type in program**:
  - **BENEFIT:** existing functionality remained unchanged only new concern is employed as the substitution while instantiating of the

Source: https://www.google.com/url?sa=i&url=https%3A%2F%2Ftheconversation.com%2Fegg-colours-make-cuckoos-masters-of-disguise-34217&psig=AOvVaw3NA4ZXURt2uxocGpdamS0T&ust=1724701901436000&source=images&cd=vfe&opi=89978449&ved=0CBIQjRxqFwoTCLCqrrT1kIgDFQAAAAAdAAAAABAE

```
public aspect CuckoosEggAspect() {
  pointcut cuckoosConstructors():
    call(EggClass.new()):
      <callee pointcut>;

  AbstractEgg around():
    cuckoosConstructors() {
      return new CuckoosEgg();
    }
}
```

Getting call of constructor join point of EggClass (should be replaced with CuckoosEgg instance)

Original type that must hold

Place where replacement with CuckoosEgg instance will happen

Cuckoo's Egg

CONTRADICTING FORCES:

1. Instead of an object of one type, an object of another type is needed,
2. but the original type must not be changed.

THEIR RESOLUTION:

Instantianting class of another type inherited from origi type and returning this instance instead of original created object instance (during its constructor call)

Why autochthonous?

```
public aspect CuckoosEggAspect() {
    pointcut cuckoosConstructors():
        call(EggClass.new()):
            <callee pointcut>;

    AbstractEgg around()  :
            cuckoosConstructors() {
            return new CuckoosEgg();
    }
}
```

Constructor
has to be captured
in some way – which is in object oriented world solved
by changes to original code

Coplien's form of Cuckoo's Egg

Problem: Instead of an object of the original type, under certain conditions, an object of some other type is needed.

Context: The original type may be used in various contexts. The need for the object of another type can be determined before the instantiation takes place.

Forces: An object of some other type is needed, but the type that is going to be instantiated may not be altered.

Solution: Put the other type instead of the original type before instantiation and provide its instance instead of the original type instance if the conditions for this are fulfilled.

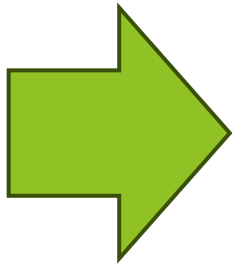
Resulting Context: The original type remains unchanged, while it appears to give instances of the other type under certain conditions. There may be several such types chosen for instantiation according to the conditions

Rationale: No need to adapt the original type

Source: http://www2.fiit.stuba.sk/~vranic/aosd/poznamky/aspekty-aj.pdf

Aspect-oriented recreation of object-oriented design patterns

Gang of Four (GoF) design patterns

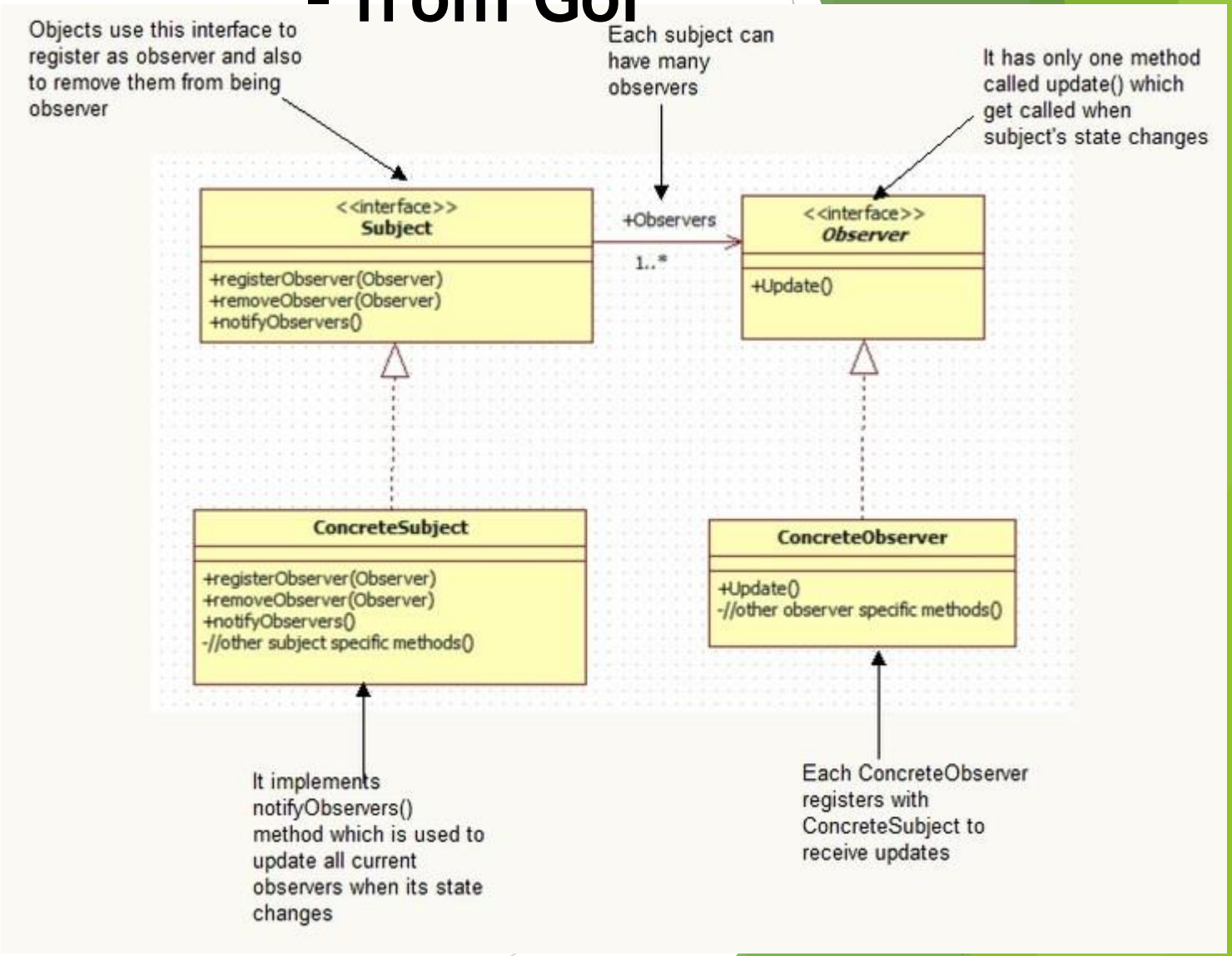


Aspect-oriented recreation of Gang of Four (GoF) design patterns

Are there any benefits?

Aspect-oriented recreation of observer design pattern

Observer design pattern - from GoF



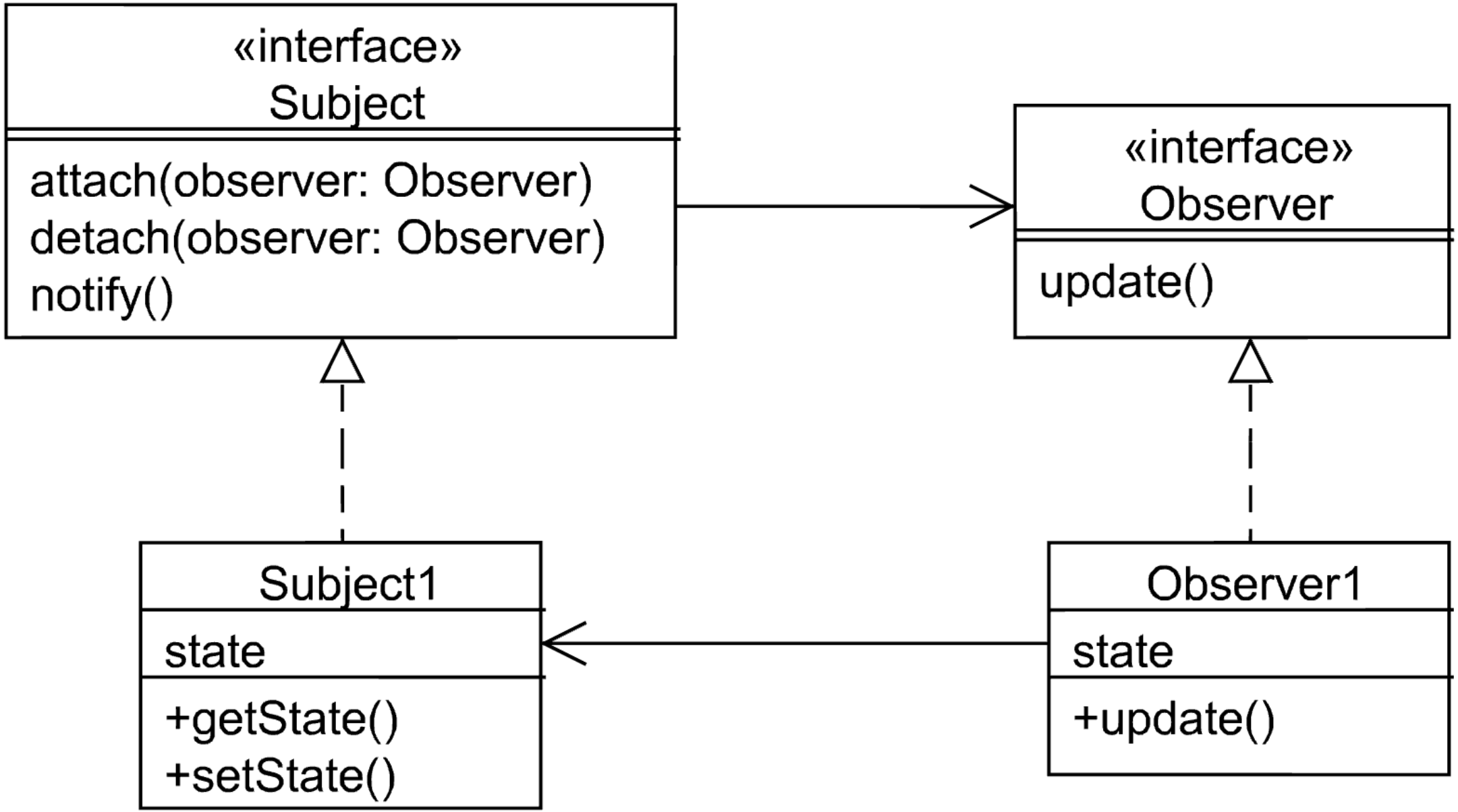
Taken from: https://www.linkedin.com/pulse/observer-pattern-usage-samson-baraka-ipdif

Implementation using pure Java

```
1  public abstract class Termometer{
2       private Subject subject = null;
3       private Celcius tempSource;
4       // getter and setter methods
5       public abstract void drawTemperature();
6       public void update() {
7             drawTemperature();
8        }
9  }
```

```
1  import java.util.Vector;
2  public class Celcius implements Subject{
3       private double degrees;
4       private Vector observers = new Vector();
5       public Object getData() { return this; }
6       public double getDegrees(){
7            return degrees;
8       }
9       public void setDegrees(double aDegrees){
10           degrees = aDegrees;
11           for (int i=0;i<getObservers().size();i++){
12                ((Observer)getObservers().
13                elementAt(i)).update();
14           }
15      }
16      public void add(Observer obs) {
17           observers.addElement(obs);
18           obs.setSubject(this);
19      }
20      public void remove(Observer obs) {
21           observers.removeElement(obs);
22           obs.setSubject(null);
23      }
24      public Vector getObservers()
25      { return observers; }
26      Celcius(double aDegrees){
27           setDegrees(aDegrees);
28      }
29 }
```

According to: *E. Piveta and L. Zancanella, "Observer pattern using aspect-oriented programming," Proceedings of the Third Latin American Conference on Pattern Languages of Programming, p. 12, 12 2003*



According to: http://www2.fiit.stuba.sk/~vranic/aosd/index.html

Implementation using AspectJ

```
1  import java.util.Vector;
2  interface Subject {
3       void add(Observer obs);
4       void remove(Observer obs);
5       Vector getObservers();
6       Object getData();
7  }
```

```
1  interface Observer {
2       void setSubject(Subject s);
3       Subject getSubject();
4       void update();
5  }
```

Celsius class - the subject

```
1  public class Celsius{
2       private double degrees;
3       public double getDegrees(){
4            return degrees;
5       }
6       public void setDegrees(double aDegrees){
7            degrees = aDegrees;
8       }
9       Celsius(double aDegrees){
10           setDegrees(aDegrees);
11      }
12 }
```

Thermometer class - the observers superclass

```
1  public class Thermometer{
2       private Celsius tempSource;
3       public void setTempSource(Celsius atempSource){
4            tempSource = atempSource;
5       }
6       public Celsius getTempSource(){
7            return tempSource;
8       }
9       public void drawTemperature(){}
10 }
```

Specialized observers

```
1  public class CelsiusThermometer extends Thermometer{
2       public void drawTemperature(){
3         System.out.println("Temperature in Celsius:"+
4              getTempSource().getDegrees());
5       }
6  }
```

```
1  public class FahrenheitThermometer extends Thermometer{
2    public void drawTemperature(){
3         System.out.println("Temperature in Fahrenheit:"+
4         (1.8 * getTempSource().getDegrees())+32);
5    }
6  }
```

Separating concerns with intertype declaration in AspectJ

```
1   import java.util.Vector;
2   abstract aspect ObserverPattern {
3        abstract pointcut stateChanges(Subject s);
4        after(Subject s): stateChanges(s) {
5             for (int i = 0; i < s.getObservers().size(); i++)
6               ((Observer)s.getObservers().elementAt(i)).update();
7        }
8        private Vector Subject.observers = new Vector();
9        public void Subject.add(Observer obs) {
10            observers.addElement(obs);
11            obs.setSubject(this);
12       }
13       public void Subject.remove(Observer obs) {
14            observers.removeElement(obs);
15            obs.setSubject(null);
16       }
17       public Vector Subject.getObservers() { return observers; }
18       private Subject Observer.subject = null;
19       public void Observer.setSubject(Subject s) { subject = s; }
20       public Subject Observer.getSubject() { return subject; }
21  }
```

Updates observers when there has been detected a change

Adds add(Observer obs) method into Subject class [INTERTYPE DECLARATION]

Adds remove(Observer obs) method into Subject class [INTERTYPE DECLARATION]

…

[INTERTYPE DECLARATION]

Implementation functionality of observer separated from business logic (measuring temperature)

```
1  import java.util.Vector;
2  aspect ObserverPatternImpl extends ObserverPattern {
3       declare parents: Celsius implements Subject;
4       public Object Celsius.getData() { return this; }
5       declare parents: Thermometer implements Observer;
6       public void Thermometer.update() {
7            drawTemperature();
8       }
9       pointcut stateChanges(Subject s): target(s) &&
10           call(void Celsius.setDegrees(..));
11 }
```

According to: *E. Piveta and L. Zancanella, "Observer pattern using aspect-oriented programming," Proceedings of the Third Latin American Conference on Pattern Languages of Programming, p. 12, 12 2003*

Another Observer Implementation Using Abstract Aspect

```
01 public abstract aspect ObserverProtocol {
02
03   protected interface Subject  { }
04   protected interface Observer { }
05
06   private WeakHashMap perSubjectObservers;07
08   protected List getObservers(Subject s) {
09     if (perSubjectObservers == null) {
10       perSubjectObservers = new WeakHashMap();
11     }
12     List observers =
13       (List)perSubjectObservers.get(s);
14     if ( observers == null ) {
15       observers = new LinkedList();
16       perSubjectObservers.put(s, observers);
17     }
18     return observers;
19   }
20
21   public void addObserver(Subject s,Observer o){
22     getObservers(s).add(o);
23   }
```

Source: J. Hannemann and G. Kiczales, "Design pattern implementation in Java and AspectJ," in Proc. of 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2002. Seattle, Washington, USA: ACM, 2002, pp. 161–173.

Another Observer Implementation Using Abstract Aspect

```
24    public void removeObserver(Subject s,Observer o){
25      getObservers(s).remove(o);
26    }
27
28    abstract protected pointcut
29      subjectChange(Subject s);
30
31    abstract protected void
32      updateObserver(Subject s, Observer o);
33
34    after(Subject s): subjectChange(s) {
35      Iterator iter = getObservers(s).iterator();
36      while ( iter.hasNext() ) {
37        updateObserver(s, ((Observer)iter.next()));
38      }
39    }
40  }
```

Figure 2: The generalized ObserverProtocol aspect

Source: J. Hannemann and G. Kiczales, "Design pattern implementation in Java and AspectJ," in Proc.
of 17th ACM SIGPLAN Conference on Object-Oriented
Programming, Systems, Languages, and Applications,
OOPSLA 2002. Seattle, Washington, USA: ACM, 2002, pp. 161–173.

```
01 public aspect ColorObserver extends ObserverProtocol {
02
03   declare parents: Point  implements Subject;
04   declare parents: Line   implements Subject;
05   declare parents: Screen implements Observer;
06
07   protected pointcut subjectChange(Subject s):
08     (call(void Point.setColor(Color)) ||
09      call(void  Line.setColor(Color)) ) && target(s);
10
11   protected void updateObserver(Subject s,
12                                 Observer o) {
13     ((Screen)o).display("Color change.");
14   }
15 }
```

```
16 public aspect CoordinateObserver extends
17   ObserverProtocol {
18
19   declare parents: Point  implements Subject;
20   declare parents: Line   implements Subject;
21   declare parents: Screen implements Observer;
22
23   protected pointcut subjectChange(Subject s):
24      (call(void Point.setX(int))
25       || call(void Point.setY(int))
26       || call(void Line.setP1(Point))
27       || call(void Line.setP2(Point)) ) && target(s);
28
29   protected void updateObserver(Subject s,
30                                 Observer o) {
31     ((Screen)o).display("Coordinate change.");
32   }
33 }
```

Figure 3. Two different Observer instances.

Source: J. Hannemann and G. Kiczales,
“Design pattern implementation in Java and AspectJ,” in Proc. of 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2002. Seattle, Washington, USA: ACM, 2002, pp. 161–173.

```
01 public aspect ScreenObserver
02                extends ObserverProtocol {
03
04  declare parents: Screen implements Subject;
05  declare parents: Screen implements Observer;
06
07  protected pointcut subjectChange(Subject s):
08    call(void Screen.display(String)) && target(s);
09
10  protected void updateObserver(
11    Subject s, Observer o) {
12      ((Screen)o).display("Screen updated.");
13  }
14 }
```

Figure 4. The same class can be Subject and Observer

Source: J. Hannemann and G. Kiczales,
"Design pattern implementation in Java and AspectJ," in Proc. of 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2002. Seattle, Washington, USA: ACM, 2002, pp. 161–173.

Intertype declaration

- Aspects can declare members (fields, methods, and constructors) that are owned by other types. These are called inter-type members. Aspects can also declare that other types implement new interfaces or extend a new class. Here are examples of some such inter-type declarations:

  Source: https://eclipse.dev/aspectj/doc/released/progguide/language-interType.html

Aspect-Oriented Refactoring of Singleton Design Pattern

Cuckoo´s egg pattern

Adapted from: P. Baca and V. Vranic, "Replacing Object-Oriented Design Patterns with Intrinsic Aspect-Oriented Design Patterns," *2011 Second Eastern European Regional Conference on the Engineering of Computer Based Systems*, Bratislava, Slovakia, 2011, pp. 19-26, doi: 10.1109/ECBS-EERC.2011.13.

Aspect-Oriented Refactoring of Abstract Factory Design Pattern

Abstract Factory

-providing interface for creating families of objects without the specifying the classes

Aspect-Oriented Recreation of Abstract Factory

-adding inter-type declarations to interface to an interface implemented by a factory class

Cuckoo´s egg pattern

Circle2D and Circle3D families of shape classes

Use of and capturing a call to static factory method

FACTORY METHOD to create circle

- inserted to abstract Circle class using inter-type declarations

NO CONCRETE FACTORY ASPECT

- throwing an exception in this case

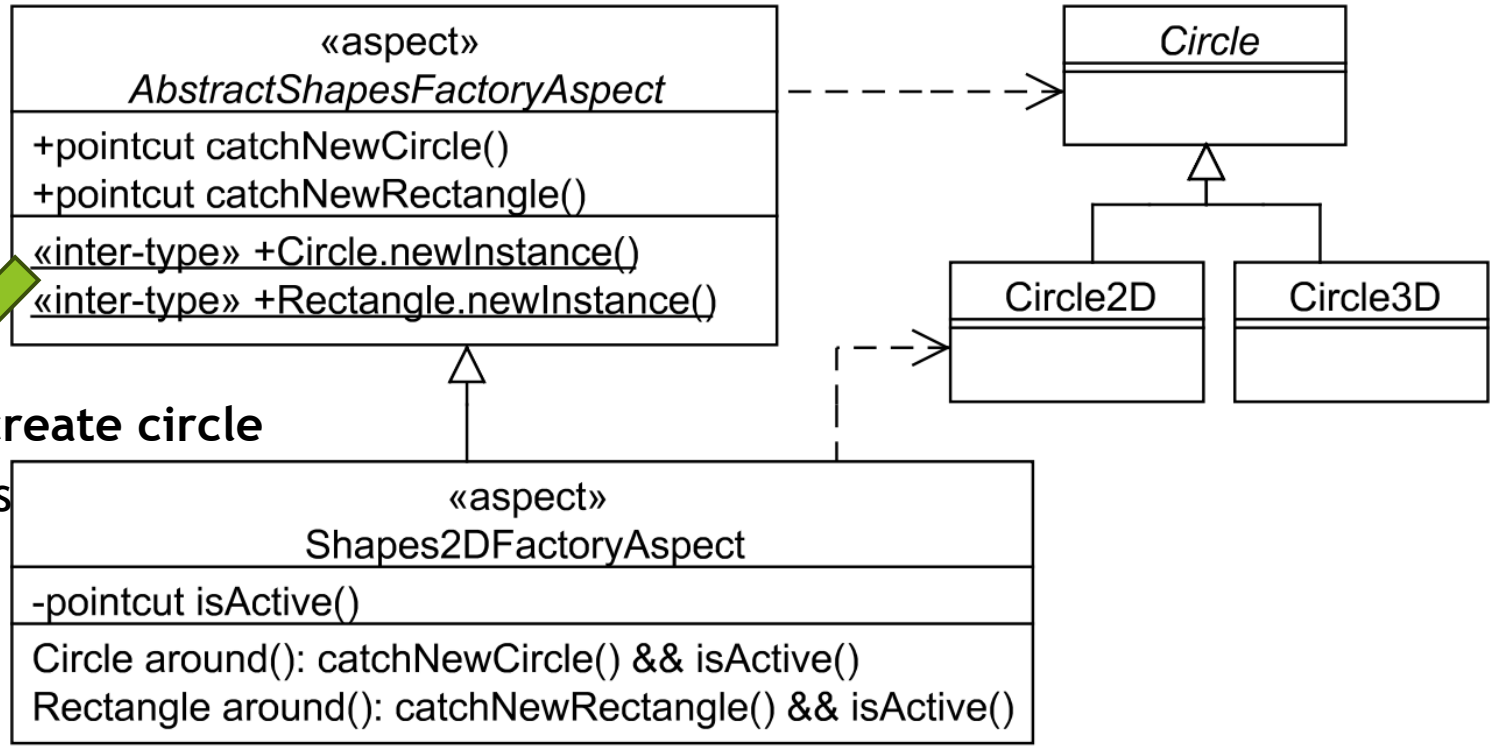


Figure 2. Cuckoo's Egg as a replacement for Abstract Factory.

Aspect-Oriented Refactoring of State Design Pattern

```
public aspect StatePattern {
    protected MachineState stopped = new StoppedMachineState();
    protected MachineState cleaning = new CleaningMachineState();
    protected MachineState program1 = new Program1MachineState();
    // ALL POSSIBLE STATES OF Machine ENTITY AS PART OF MachineState CLASS
    // ……..

    after(Machine machine, MachineState machineState):
                call(void MachineState.stop()) && target(machineState) && this(machine) {
        if (machineState.getState() != stopped) {
            machineState.setState(stopped);
        }
        // if machine cannot be stopped from some state then exception should be thrown!
    }
```

Adapted from: *R. Miles, AspectJ cookbook, 1st ed. Sebastopol, CA; Farnham: O'Reilly Media, 2004.*

Aspect-Oriented Refactoring of State Design Pattern

```
before(Machine machine, MachineState machineState):
    call(void MachineState.clean()) && target(machineState) && this(machine) {
    if (machineState.getState() == stopped) {
            raise new Exception("Stopped machine cannot clean!");
     } else {
            System.out.println("Cleaning has started!");
            machineState.setState(cleaning);
    }
}
// OTHER MANAGED STATES WITHIN STATES
// FOR Machine ENTITY/CLASS
// ..............
}
```

THE STATE AS SEPARATE CONCERN

-not embedded in methods as in OOP

Modularization of rules used in state transition – in one aspect

-easier analysis of state transition – easy to add, modify, and remove the state

Aspect-Oriented Refactoring of Flyweight Design Pattern

ABSTRACTION OF THE PATTERN

```
public abstract aspect FlyweightPattern {
    private Set<Object> flyweightResources = new HashSet<Object>();
    public interface Flyweight {}

    protected abstract Flyweight createNewFlyweight(Object object);
    protected abstract pointcut flyweightPointcut(Object object);

    Object around(Object key): flyweightPointcut(key) &&
                               !within(path.this.aspect.FlyweightPattern) {
        return this.manageFlyweight(key);
    }
```

Adapted from: *R. Miles, AspectJ cookbook, 1st ed. Sebastopol, CA; Farnham: O'Reilly Media, 2004.*

Aspect-Oriented Refactoring of Flyweight Design Pattern

ABSTRACTION OF THE PATTERN

```
public synchronized Flyweight manageFlyweight(Object key) {
        if (flyweightResources.containsKey(key)) {
                return (Flyweight) flyweightResources.get(key);
        } else {
                Flyweight flyweigthHeavyWeight = createNewFlyweight(key);
                flyweightResources.put(key, flyweigthHeavyWeight);
                return flyweigthHeavyWeight;
        }
}
}
```

Adapted from: *R. Miles, AspectJ cookbook, 1st ed. Sebastopol, CA; Farnham: O'Reilly Media, 2004.*

Aspect-Oriented Refactoring of Flyweight Design Pattern

```
public aspect HeavyWeightObjectLifting extends Flyweight {

    declare parents: HeavyWeightObjectLifting implements Flyweight;


    protected pointcut flyweightApplication(Integer weight):
    call(path.heavy.weight.instance.HeavyWeight.new(Integer)) && args(weight);


    protected Flyweight applyNewFlyweight(Integer weight) {
        return new HeavyWeight(weight);
    }
}
```

Adapted from: *R. Miles, AspectJ cookbook, 1st ed. Sebastopol, CA; Farnham: O'Reilly Media, 2004.*

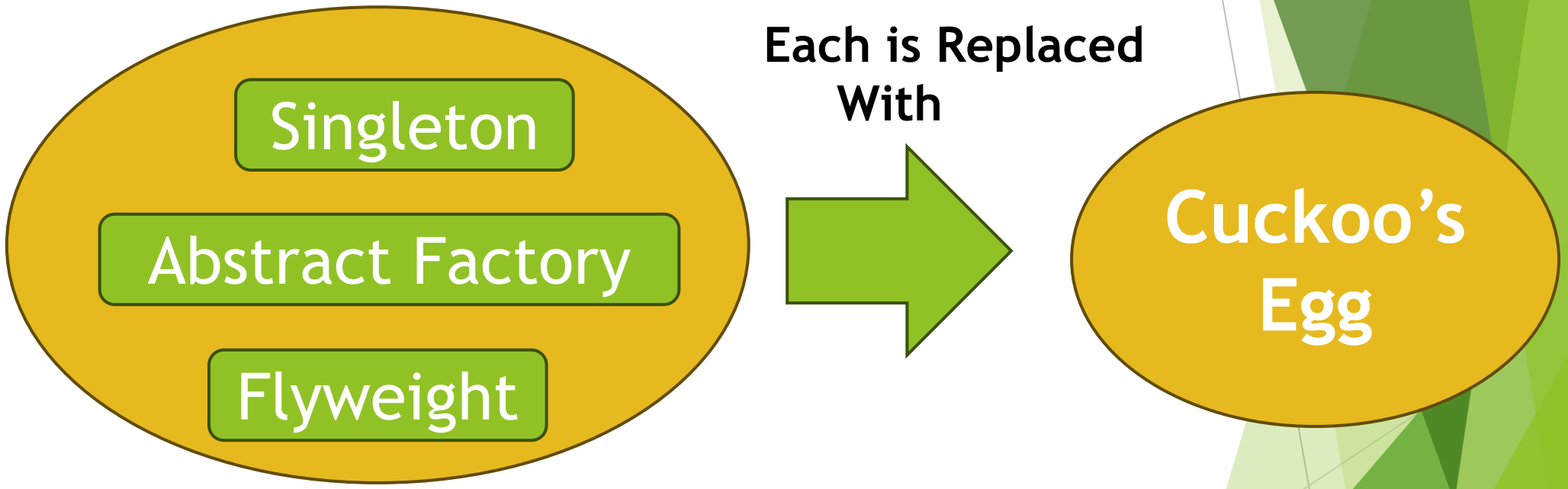
Aspect-Oriented Refactoring of Flyweight Design Pattern

Using Cuckoo´s egg pattern instead

1. Creating the new instance on every request when such instance is needed
2. Searching and optionally getting **instance** from the hash-table if available
3. Otherwise creating new (**heavy weight)** instance calling **proceed()** in aspect method
4. Storing instance in hash map
5. Returning instance

Adapted from: P. Baca and V. Vranic, "Replacing Object-Oriented Design Patterns with Intrinsic Aspect-Oriented Design Patterns," *2011 Second Eastern European Regional Conference on the Engineering of Computer Based Systems*, Bratislava, Slovakia, 2011, pp. 19-26, doi: 10.1109/ECBS-EERC.2011.13.

Replacing OOP Patterns With AOP Intrinsic Ones



Adapted from: P. Baca and V. Vranic, "Replacing Object-Oriented Design Patterns with Intrinsic Aspect-Oriented Design Patterns," *2011 Second Eastern European Regional Conference on the Engineering of Computer Based Systems*, Bratislava, Slovakia, 2011, pp. 19-26, doi: 10.1109/ECBS-EERC.2011.13.

Table 1. Design pattern, roles, and desirable properties of their AspectJ implementations

	Modularity Properties				Kinds of Roles	
Pattern Name	Locality(**)	Reusability	Composition Transparency	(Un)pluggability	Defining(*)	Superimposed
Façade	Same implementation for Java and AspectJ				Façade	-
Abstract Factory	no	no	no	no	Factory, Product	-
Bridge	no	no	no	no	Abstraction, Implementor	-
Builder	no	no	no	no	Builder, (Director)	-
Factory Method	no	no	no	no	Product, Creator	-
Interpreter	no	no	n/a	no	Context, Expression	-
Template Method	(yes)	no	no	(yes)	(AbstractClass), (ConcreteClass)	(AbstractClass), (ConcreteClass)
Adapter	yes	no	yes	yes	Target, Adapter	Adaptee
State	(yes)	no	n/a	(yes)	State	Context
Decorator	yes	no	yes	yes	Component, Decorator	ConcreteComponent
Proxy	(yes)	no	(yes)	(yes)	(Proxy)	(Proxy)

Source: J. Hannemann and G. Kiczales, "Design pattern implementation in Java and AspectJ," in Proc. of 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2002. Seattle, Washington, USA: ACM, 2002, pp. 161–173.

Visitor	(yes)	yes	yes	(yes)	Visitor	Element
Command	(yes)	yes	yes	yes	Command	Commanding, Receiver
Composite	yes	yes	yes	(yes)	(Component)	(Composite, Leaf)
Iterator	yes	yes	yes	yes	(Iterator)	Aggregate
Flyweight	yes	yes	yes	yes	FlyweightFactory	Flyweight
Memento	yes	yes	yes	yes	Memento	Originator
Strategy	yes	yes	yes	yes	Strategy	Context
Mediator	yes	yes	yes	yes	-	(Mediator), Colleague
Chain of Responsibility	yes	yes	yes	yes	-	Handler
Prototype	yes	yes	(yes)	yes	-	Prototype
Singleton	yes	yes	n/a	yes	-	Singleton
Observer	yes	yes	yes	yes	-	Subject, Observer

(*) The distinctions between defining and superimposed roles for the different patterns were not always easy to make. In some cases, roles are clearly superimposed (e.g. the Subject role in Observer), or defining (e.g. State in the State pattern). If the distinction was not totally clear, the role names are shown in parentheses in either or both categories.

(**) Locality: "(yes)" means that the pattern is localized in terms of its superimposed roles but the implementation of the remaining defining role is still done using multiple classes (e.g. State classes for the State pattern). In general, (yes) for a desirable property means that some restrictions apply

Source: J. Hannemann and G. Kiczales, "Design pattern implementation in Java and AspectJ," in Proc. of 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2002. Seattle, Washington, USA: ACM, 2002, pp. 161–173.

Aspect-Oriented Recreation of Design Patterns - Benefits

STRUCTURALLY THE SAME FACADE

- the implementation in AspectJ is the same

-pattern is providing unified interface
-to a set of interfaces of subsystem

-good namespace management required

USING ROLES ONLY WITHIN PATTERN ASPECTS

COMPOSITE, COMMAND, MEDIATOR, CHAIN OF RESPONSIBILITY

-introduced roles as part of only aspect patterns
- not need to expose them to outside world
(such as in OBSERVER pattern)

MODULARIZATION OF SCATTERED CODE STATE, INTERPRETER

-separation of state management
in State pattern

EMPTY (PROTECTED) INTERFACES

INTRODUCTION OF TYPES

-used within pattern

-defined in abstract aspect

Defining abstraction

(concretized later for specific application)

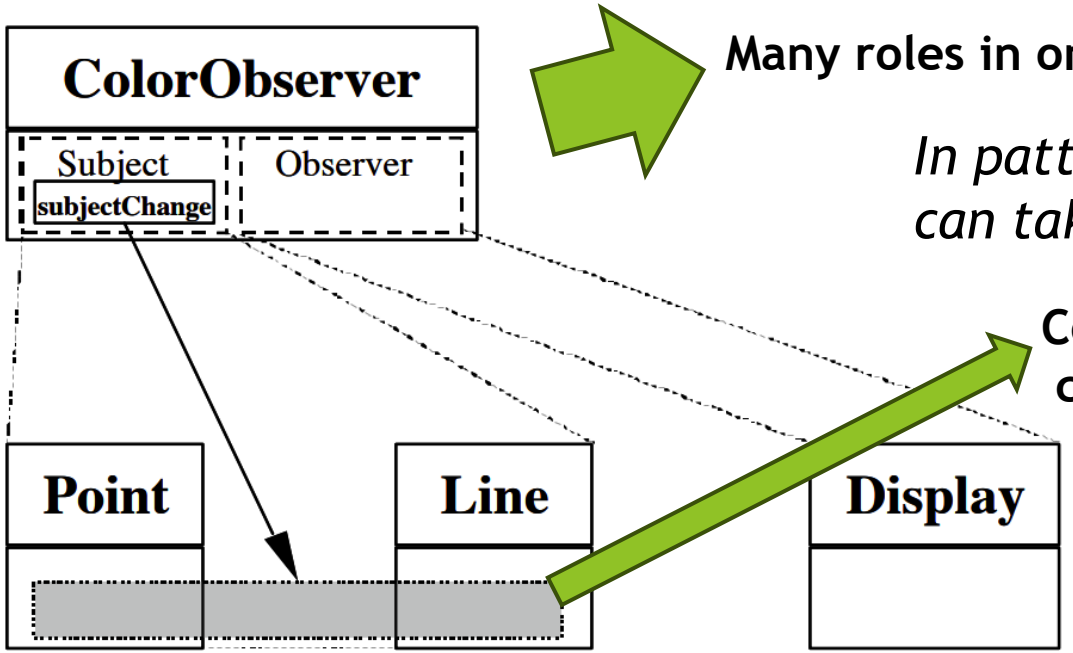
-roles

-implementation where possible

Roles and Their Crosscutting in Patterns

1 role can be represented by many classes and vice-versa

1 conceptual operation can crosscut more methods and vice-versa



Many roles in one class

In pattern language/composition the class from one role can take other role from different pattern

Conceptual methods where 1 method crosscuts many classes

Figure 5: The structure of an instance of the Observer pattern in AspectJ. Subject and Observer roles crosscut classes, and the changes of interest (the subjectChange pointcut) crosscuts methods in various classes.

Source: J. Hannemann and G. Kiczales, "Design pattern implementation in Java and AspectJ," in Proc. of 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2002. Seattle, Washington, USA: ACM, 2002, pp. 161 173.

```
public abstract aspect CompositionProtocol {

  protected interface Component {}
  protected interface Composite extends Component {}
  protected interface Leaf    extends Component {}

  private WeakHashMap perComponentChildren =
    new WeakHashMap();

  private Vector getChildren(Component s) {
    Vector children;
    children = (Vector)perComponentChildren.get(s);
    if ( children == null ) {
      children = new Vector();
      perComponentChildren.put(s, children);
    }
    return children;
  }

  public void addChild(Composite composite,
                       Component component) {
    getChildren(composite).add(component);
  }
```

Composite pattern

EMPTY INTERFACES

-->Defining roles in patterns

-introducing types

[key, value] =
[Composite/Component,
Vector<Component>]

-->Implementing default behavior according to pattern

Source: J. Hannemann and G. Kiczales, “Design pattern implementation in Java and AspectJ,” in Proc. of 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2002. Seattle, Washington, USA: ACM, 2002, pp. 161–173.

```
  public void removeChild(Composite composite,
                          Component component) {
    getChildren(composite).remove(component);
  }

  public Enumeration getAllChildren(Component c) {
    return getChildren(c).elements();
  }

  protected interface FunctionVisitor {
    public Object doIt(Component c);
  }

  protected static Enumeration
   recurseFunction(Component c,
                   FunctionVisitor fv) {
    Vector results = new Vector();
    for (Enumeration enum = getAllChildren(c);
      enum.hasMoreElements(); ) {
        Component child;
        child = (Component)enum.nextElement();
        results.add(fv.doIt(child));
    }
    return results.elements();
  }
}
```

-->**Implementing default behavior according to pattern**

Source: J. Hannemann and G. Kiczales, "Design pattern implementation in Java and AspectJ," in Proc. of 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2002. Seattle, Washington, USA: ACM, 2002, pp. 161–173.

```
public aspect FileSystemComposite extends
              CompositeProtocol {

  declare parents: Directory implements Composite;
  declare parents: File      implements Leaf;

  public int sizeOnDisk(Component c) {
    return c.sizeOnDisk();
  }

  private abstract int Component.sizeOnDisk();

  private int Directory.sizeOnDisk() {
     int diskSize = 0;
     java.util.Enumeration enum;
     for (enum =
      SampleComposite.aspectOf().getAllChildren(this);
      enum.hasMoreElements(); ) {
        diskSize +=
         ((Component)enum.nextElement()).sizeOnDisk();
     }
     return diskSize;
  }

  private int File.sizeOnDisk()      {
    return size;
  }
}         }
```

Figure 7. Part of a Composition pattern instance aspect

-->**Specific Aspect to uniformly traverse files/directories**

-->**Applying roles from parent for specific case**

- Directory =has role= Composite
- File =has role= Leaf

Client get size on disk using **public methods**

-others are encapsulated within pattern (aspect)!!!

-->**Specific implementation of how the size on disk is calculated -using intertype declaration**

Source: J. Hannemann and G. Kiczales, "Design pattern implementation in Java and AspectJ," in Proc. of 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2002. Seattle, Washington, USA: ACM, 2002, pp. 161-173.

ASPECTS USED AS OBJECT FACTORIES

SINGLETON, PROTOTYPE, MEMENTO, ITERATOR, FLYWEIGHT

-factory methods:

a) PARAMETERIZED METHODS ON ABSTRACT ASPECT

b) METHODS ATTACHED ON PARTICIPANTS

Nordberg's factory example:

-empty factory method returning null or default object

around advice *Extending factory without modification of original code*

- is used to return specific instances of particular types wrapping this factory method

-new object is instantiated according to provided arguments

-otherwise default or null value is returned calling original method

-removes close coupling between original object and its representants / accessor is removed in Memento, Iterator

Source: J. Hannemann and G. Kiczales, "Design pattern implementation in Java and AspectJ," in Proc. of 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2002. Seattle, Washington, USA: ACM, 2002, pp. 161–173.

Aspect-Oriented Recreation of Design Patterns - Benefits

ALLOWS MUTIPLE INHERITANCE STRUCTURALLY THE SAME

BRIDGE, BUILDER, FACTORY METHOD, ABSTRACT FACTORY

- **replacing abstract classes (from original implementations) with interfaces**
  - Preserving ability to attach implementation to their default methods

Limited form of multiple inheritance

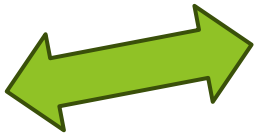
- **Open Class Mechanism**
  - Attaching fields, methods – ***extending classes***
  - Attaching fields, methods – ***extending classes***

INTRODUCING NEW LANGUAGE CONSTRUCTS

-aspect implementation **fully replaces object-oriented one**

VISITOR, ADAPTER

-extending interface of Adaptee



PROXY, STRATEGY, DECORATOR

-attaching the advice

Policy Pattern

- Defining policies or rules with the application

Compiler warning or error if such policy is broken

Police Enforcement

```
public abstract aspect GeneralPolicy {
    protected abstract pointcut warnAbout();

    declare warning: warnAbout(): "Warning...";
}

public aspect MyAppPolicy extends GeneralPolicy {
    protected pointcut warnAbout():
        call(* *.myMethod(..)) || call(* *.myMethod2());
}
```

Figure 3. The Policy pattern.

IN ONE ASPECT ⟺ **IN TWO ASPECT**

-project wide rules or policies

-local rules or exceptions

-defining abstraction using abstract aspect

-allows to specify pointcuts later during development

Source: R. Menkyna, V. Vranić and I. Polášek, "Composition and categorization of aspect-oriented design patterns," *2010 IEEE 8th International Symposium on Applied Machine Intelligence and Informatics (SAMI)*, Herlany, Slovakia, 2010, pp. 129-134, doi: 10.1109/SAMI.2010.5423751.

Exception Introduction Pattern

- If exception is not handled by advice **Should be handled in higher context**

Advice cannot declare throwing a checked exception

-advised join point has to declare this exception

-**unlikely** BASE CONCERNS TEND TO BE ADAPTED TO THEIR ASPECTS (HANDLING CROSSCUTTING CONCERNS)

ESSENCE OF THIS PATTERN

1) CATCHING A CHECKED EXCEPTION
2) WRAPPING IT INTO A NEW CONCERN SPECIFIC RUNTIME EXCEPTION

```
public abstract aspect ConcernAspect {
    abstract pointcut operations();

    before(): operations() {
        try {
            concernLogic();
        } catch (ConcernCheckedException ex) {
            throw new ConcernRuntimeException(ex);
        }
    }
    void concernLogic() throws ConcernCheckedException {
        . . .
    }
}
```

Figure 4. The Exception Introduction pattern (adapted from [10]).

Source: R. Menkyna, V. Vranić and I. Polášek, "Composition and categorization of aspect-oriented design patterns," *2010 IEEE 8th International Symposium on Applied Machine Intelligence and Informatics (SAMI)*, Herlany, Slovakia, 2010, pp. 129-134, doi: 10.1109/SAMI.2010.5423751.

Border Control Pattern

- Defining regions in the application

  -to restrict processing/modification only to particular places

```
public aspect MyRegions {
    public pointcut myTypes1(): within(mypackage1.+);
    public pointcut myTypes2(): within(mypackage2.+);
    public pointcut myTypes(): myTypes1() || myTypes2();
    . . .
}
```

Figure 1. The Border Control pattern.

In case of changes

Only regions in respective aspect will be redefined/changed

-other dependencies (aspect using it) are automatically redirected to newly defined places

Single aspect containing only pointcuts That define boundaries of regions

Using pointcuts (primitives) based on lexical structure

within() and **withincode()**

Source: R. Menkyna, V. Vranić and I. Polášek, "Composition and categorization of aspect-oriented design patterns," *2010 IEEE 8th International Symposium on Applied Machine Intelligence and Informatics (SAMI)*, Herlany, Slovakia, 2010, pp. 129-134, doi: 10.1109/SAMI.2010.5423751.

Pattern Compositions

```
public aspect Regions {
    public pointcut Testing():
        within(com.myapplication.testing.+);
    public pointcut MyApplication():
        within(com.myapplication.+);
    public pointcut ThirdParty():
        within(com.myapplication.thirdpartylibrary.+);
    public pointcut ClassSwitcher():
        within(com.myapplication.ClassSwitcher);
}
```

Figure 6. The Border Control pattern used to partition code into regions.

```
public aspect Warning {
    protected pointcut allowedUse():
        Regions.ThirdParty() || Regions.Testing();

    declare warning: call(Display.new()) && !allowedUse():
        "Class OldClass deprecated.";
}
```

Figure 7. Composing Policy with Border Control.

Figure 8 shows how Cuckoo's Egg may be applied to replace OldClass constructions with NewClass construction.

```
public aspect ClassSwitcher {
    public pointcut oldClassConstructor():
        call(*.OldClass.new()) &&
        !Regions.ThirdParty() && !Regions.Testing();

    Object around(): oldClassConstructor() {
        return new MyApplication.NewClass();
    }
}
```

Figure 8. Composing Cuckoo's Egg with Border Control.

```
public class SwitchLoggingException extends RuntimeException {
    public SwitchLoggingException(Throwable cause) {
        super(cause);
    }
}
```

```
public aspect SwitchLogging {
    before(): adviceexecution() && Regions.ClassSwitcher() {
        try {
            logSwapEvent()
        } catch(IOException e) {
            throw new SwitchLoggingException(e);
        }
    }
}
```

Figure 9. Composing Exception Introduction with Cuckoo's Egg and Border Control.

Source: R. Menkyna, V. Vranić and I. Polášek, "Composition and categorization of aspect-oriented design patterns," *2010 IEEE 8th International Symposium on Applied Machine Intelligence and Informatics (SAMI)*, Herlany, Slovakia, 2010, pp. 129-134, doi: 10.1109/SAMI.2010.5423751.

Použitá literatúra

Patterns in AspectJ - 8. chapter: The AspectJ in Action *Laddad, Ramnivas, 2003. AspectJ in action: practical aspect-oriented programming. Greenwich, CT: Manning. ISBN 978-1-930110-93-9.*

Aspect-oriented recreation of Observer design pattern: *E. Piveta and L. Zancanella, "Observer pattern using aspect-oriented programming," Proceedings of the Third Latin American Conference on Pattern Languages of Programming, p. 12, 12 2003*

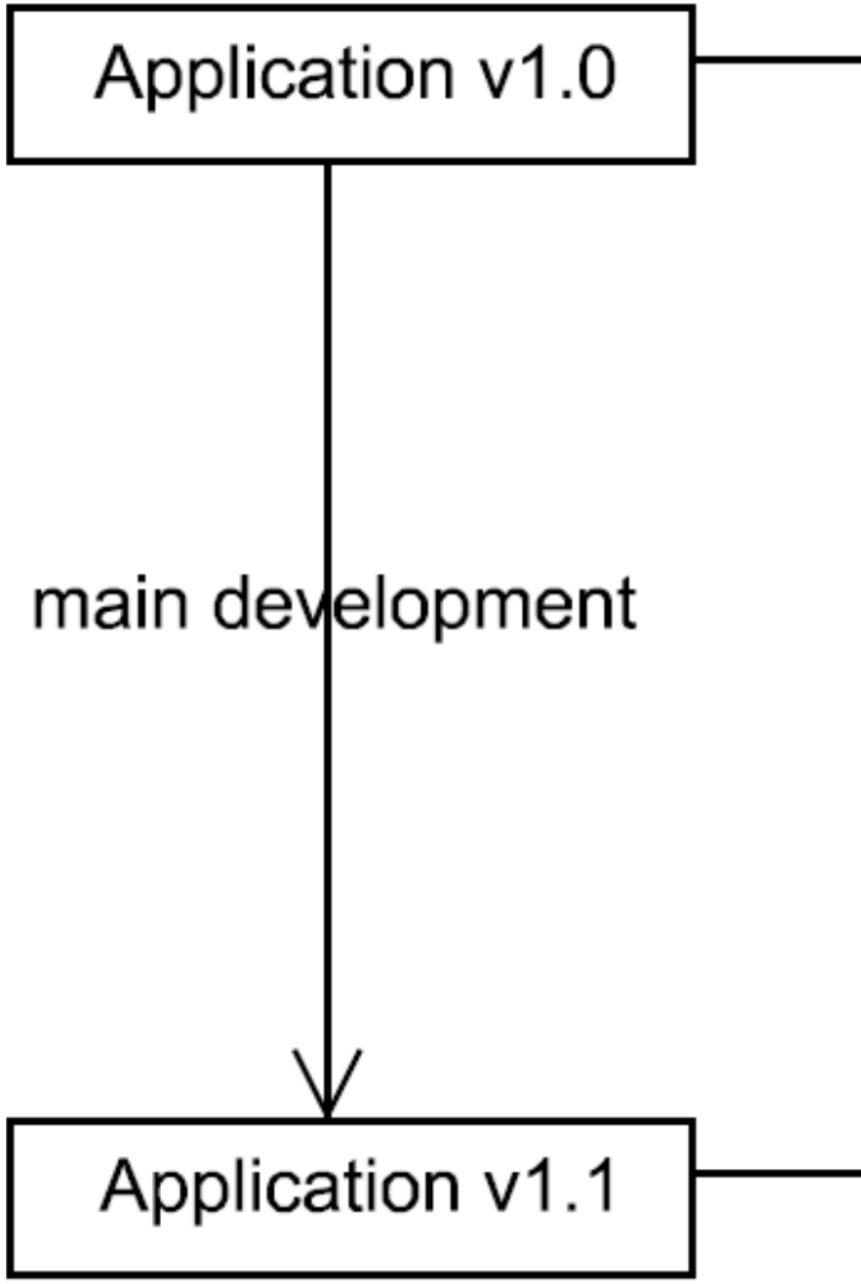
Aspect-oriended recreation of design patterns, application of patterns: *R. Miles, AspectJ cookbook, 1st ed. Sebastopol, CA; Farnham: O'Reilly Media, 2004.*

HANNEMANN, Jan a Gregor KICZALES, Design Pattern Implementation in Java and AspectJ. 2002, s. 13.

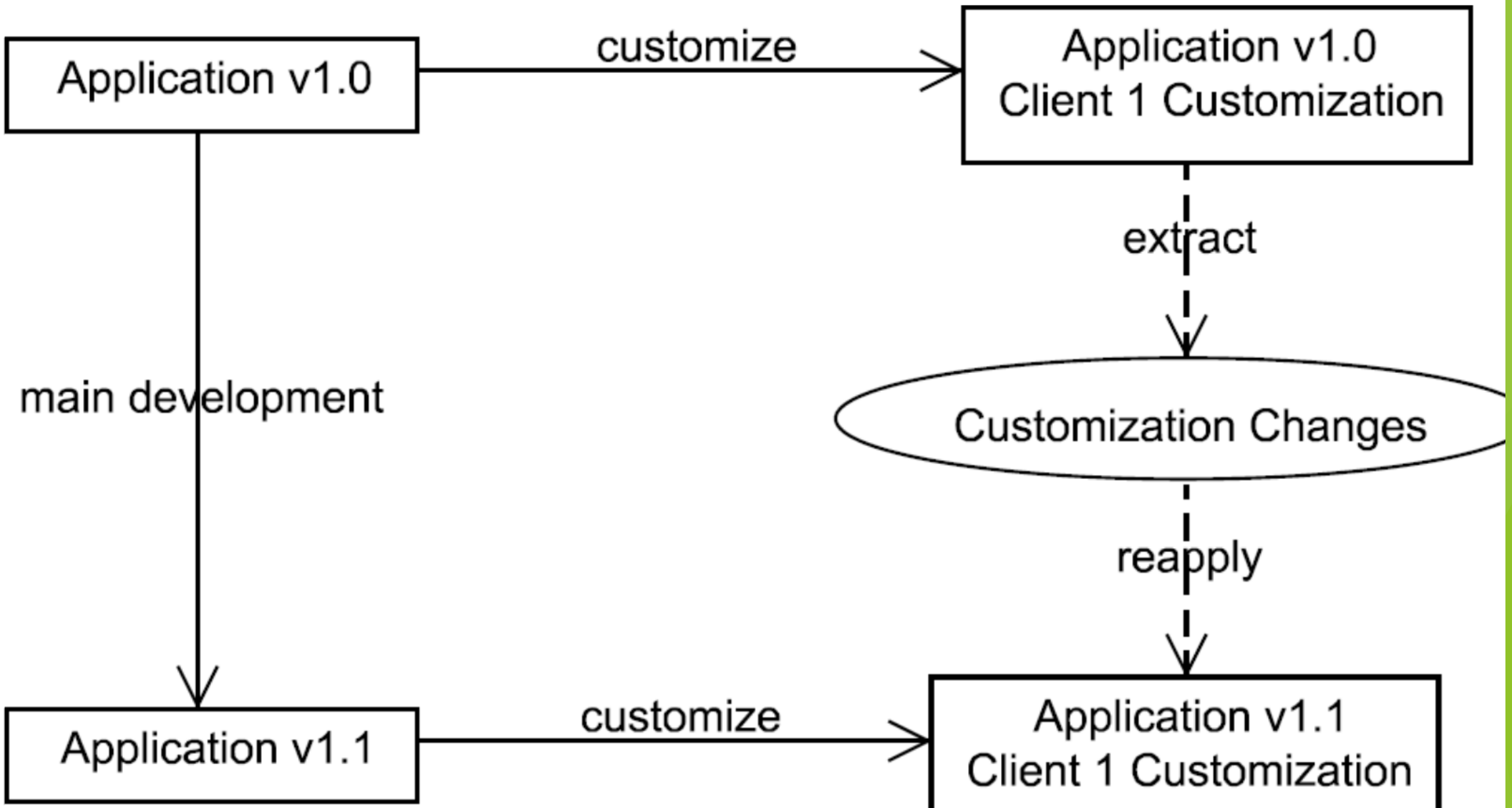
R. Menkyna, V. Vranić and I. Polášek, "Composition and categorization of aspect-oriented design patterns," *2010 IEEE 8th International Symposium on Applied Machine Intelligence and Informatics (SAMI)*, Herlany, Slovakia, 2010, pp. 129-134, doi: 10.1109/SAMI.2010.5423751.

P. Baca and V. Vranic, "Replacing Object-Oriented Design Patterns with Intrinsic Aspect-Oriented Design Patterns," *2011 Second Eastern European Regional Conference on the Engineering of Computer Based Systems*, Bratislava, Slovakia, 2011, pp. 19-26, doi: 10.1109/ECBS-EERC.2011.13.

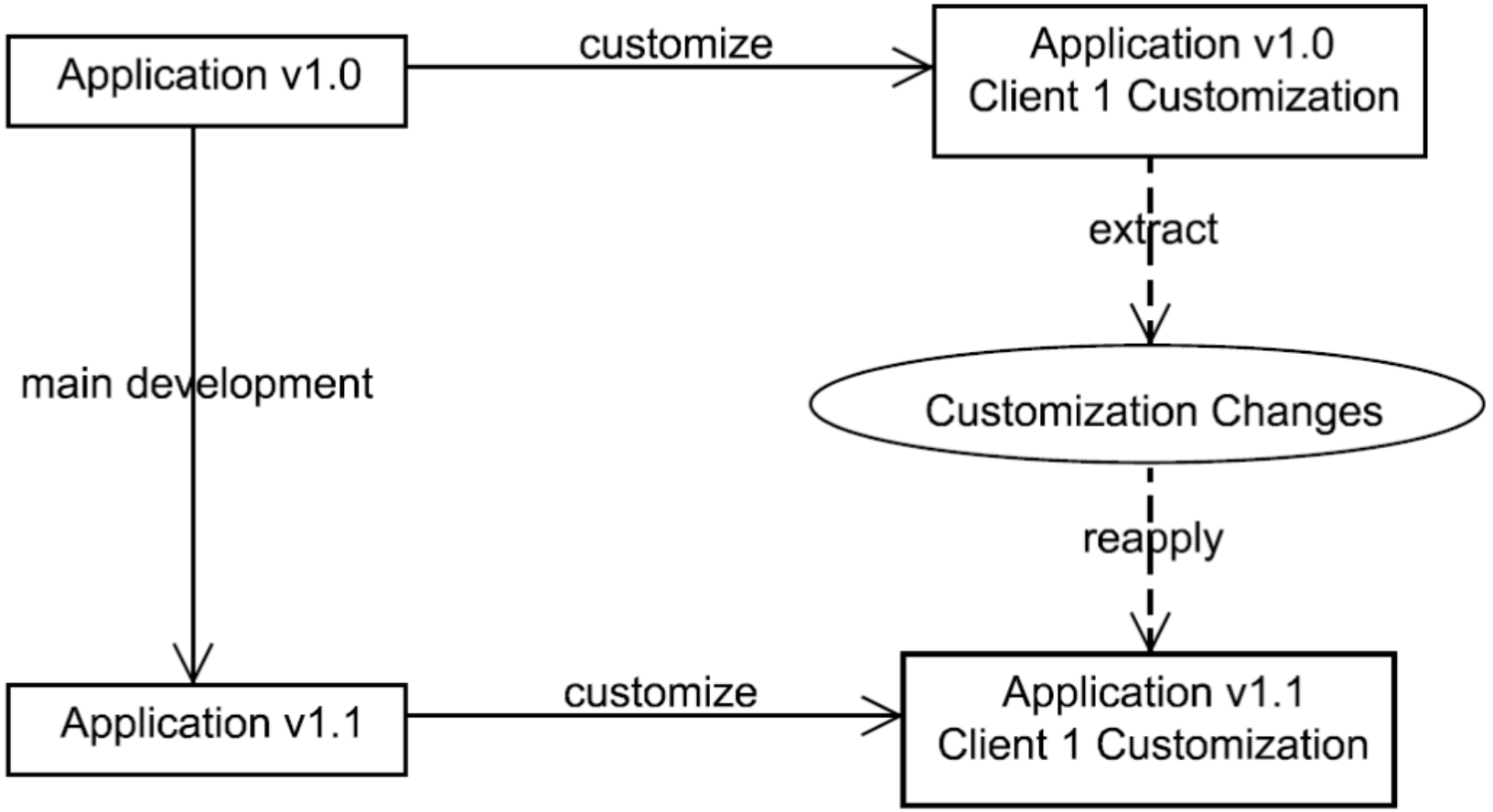
Use of aspects in change realization?



According to: http://www2.fiit.stuba.sk/~vranic/aosd/index.html

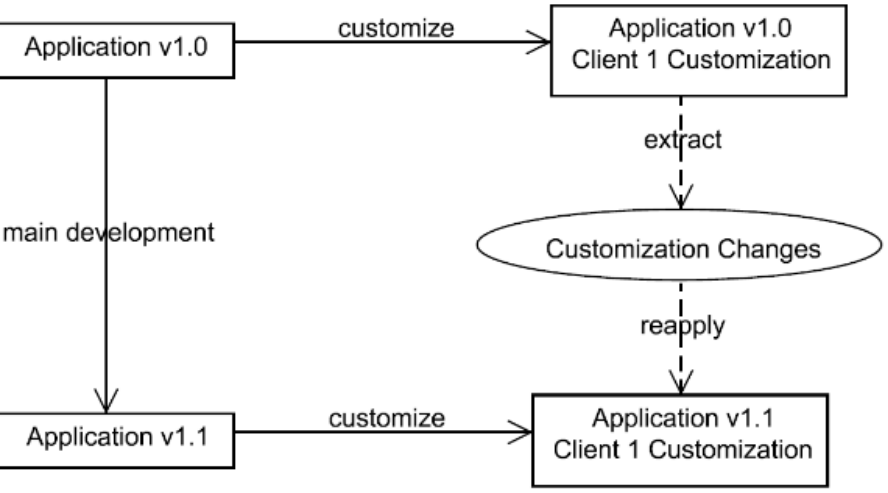


According to: http://www2.fiit.stuba.sk/~vranic/aosd/index.html



Aspects

According to: http://www2.fiit.stuba.sk/~vranic/aosd/index.html



Aspects

Aspect-oriented change realization

According to: http://www2.fiit.stuba.sk/~vranic/aosd/index.html

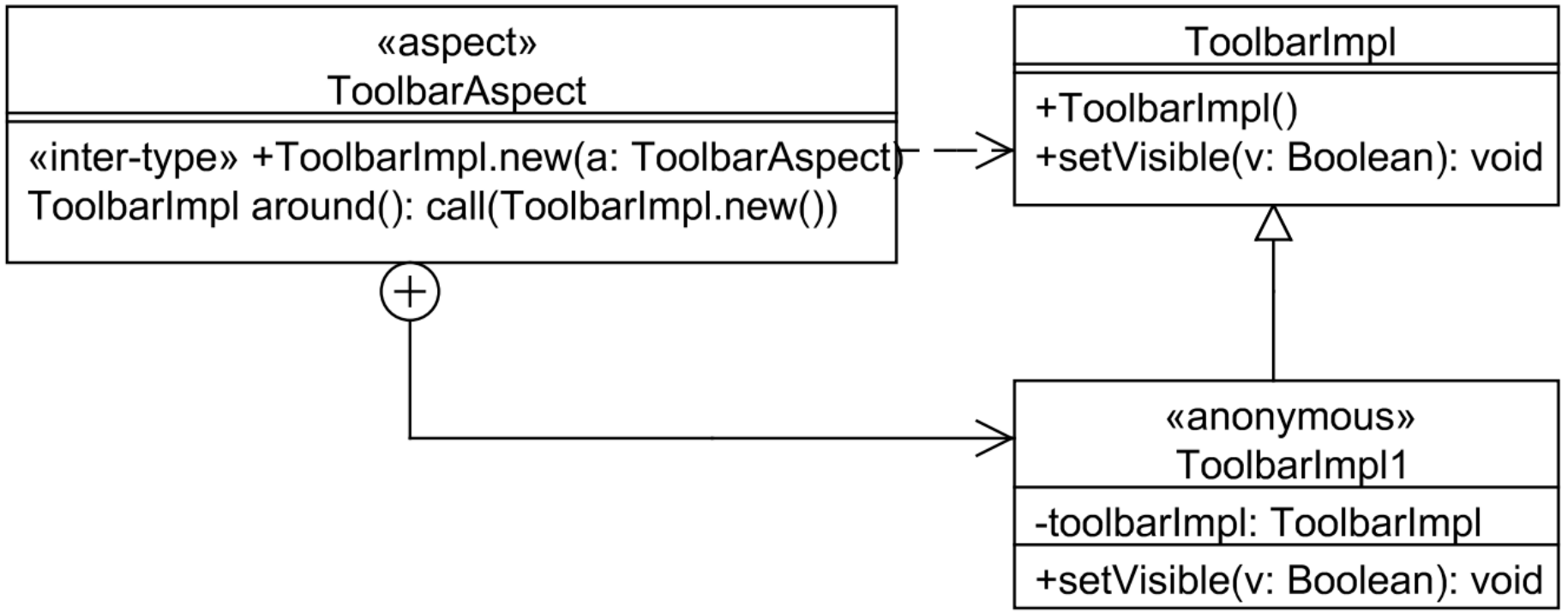


Figure 1. Worker Object Creation as a replacement for Proxy.