



Tehnologii Java


Curs -

Cristian Frăsinaru

`acf@infoiasi.ro`

Facultatea de Informatică

Universitatea "Al. I. Cuza" Iași



Programare orientată aspect

Cuprins

- Paradigme de programare: OOP, AOP
- Problemele POO
- Conceptele AOP
- AspectJ
- Exemple



Introducere



Programare orientată obiect

OOP = paradigmă de programare caracterizată prin:

- modelarea naturală a entităților unei probleme prin **clase**
- descrierea unei aplicații sub forma unei mulțimi de **obiecte** ce comunică prin transmiterea de **mesaje**
- implementarea unor concepte:
 - abstractizare
 - încapsulare
 - polimorfism, ...
- beneficiază de **șabloane de proiectare**

Identificarea funcționalităților

Un **concern** este o funcționalitate necesară unui sistem software care trebuie implementată de o anumită secvență de cod.

Analiză → Modelare → Implementare

Exemplu: "Se cere construirea unei aplicații care să țină evidența unor *produse, mărfuri, obiecte de inventar* caracterizate de *denumire, pret, etc.*".

Clasa Product (1)

Concern 1: Product

```
public class Product {
    private String name;
    private float price;

    public Product() {
    }

    public float getPrice() {
        return price;
    }

    public void setPrice(float price) {
        this.price = price;
    }
    ...
}
```

Schimbarea specificațiilor

"Se dorește ca schimbările de preț ale articolelor să fie memorate într-o arhivă"

Concern 2: Logger

```
public class Logger {  
  
    public Logger() {  
        // instantiaza mecanismul de logging  
    }  
  
    public void write(String text) {  
        // scrie textul  
    }  
}
```

Clasa Product (2)

```
public class Product {
    private float price;
    private Logger logger;
    ...

    public Product() {
        logger = new Logger();
    }

    public float getPrice() {
        return price;
    }

    public void setPrice(float price) {
        logger.write("Price changed: " + this.price + " -> " + price);
        this.price = price;
    }
    ...
}
```

Crosscutting



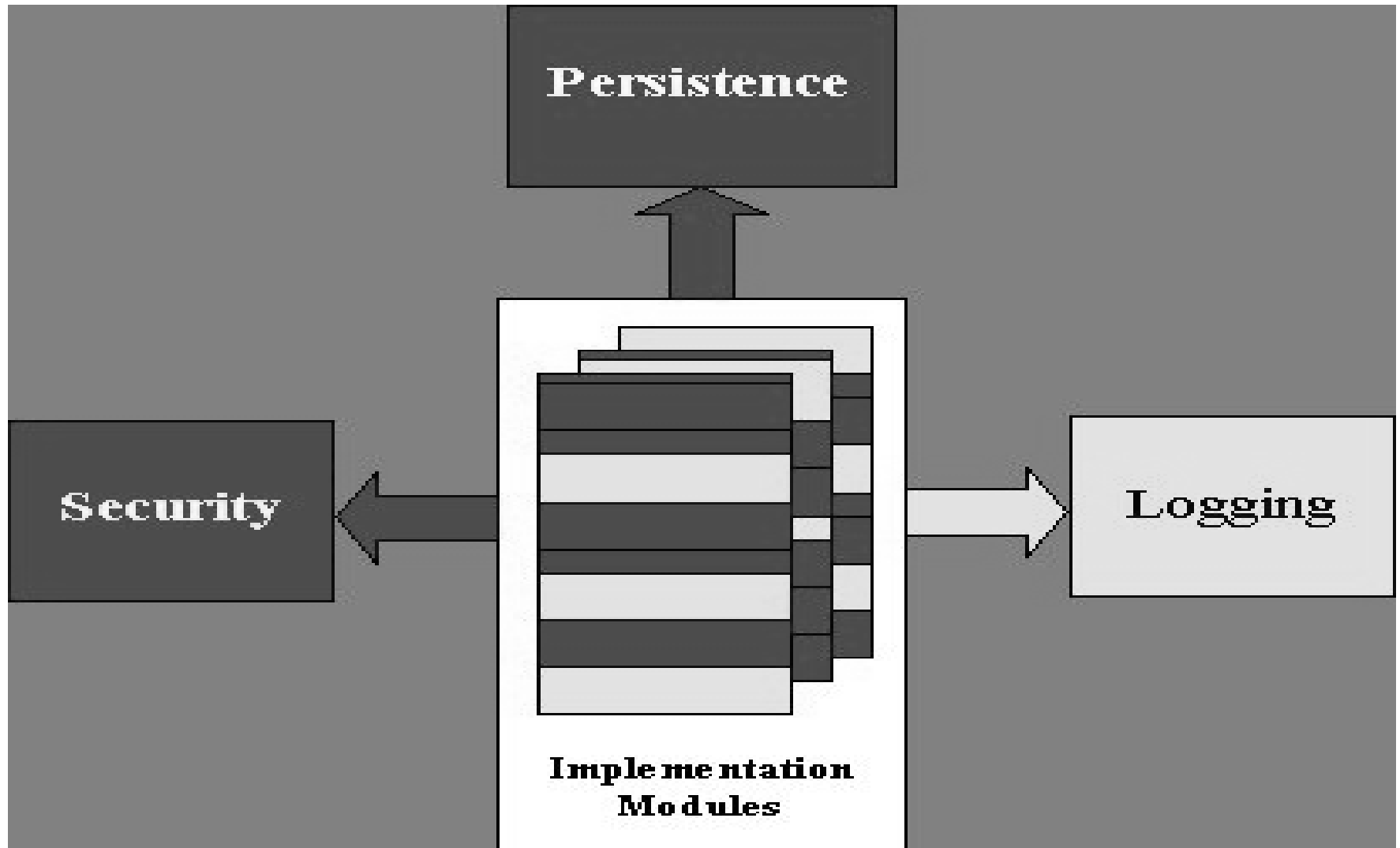
Clasa `Product` nu mai gestionează doar interesele sale (Concern 1) ci trebuie să îndeplinească și cerințe legate de al doilea interes (Concern 2).

Clasa a fost **tăiată (crosscut)** de *concern*-uri în sistem.

Crosscutting = situația când o cerință a sistemului este îndeplinită prin plasarea de cod în diferite clase dar acest cod nu ține de funcționalitatea specifică a conceptelor modelate de respectivele clase.



Crosscutting



Imprăștierea codului

Imprăștierea codului / Scattered code= codul necesar îndeplinirii unui *concern* este răspândit în multiple clase și metode, necesare pentru îndeplinirea altui *concern*.

```
class Product {
    Logger logger;
    float price;
    ...
}
public class Commodity {
    Logger logger;
    float purchasePrice;
    float sellPrice;
    ...
}
```

Incâlcirea codului

Incâlcirea codului / Tangled code= folosirea unei singure metode sau clase pentru implementarea a numeroase *concern*-uri.

```
class Product {
    ...
    public String setPrice(float price) {
        // Autorizare
        User user = Application.getCurrentUser();
        if (!user.hasPermission("price.change")) {
            throw new AuthorizationException(user, "price.change");
        }
        // Logging
        logger.write("Price changed: " + this.price + " -> " + price);
        this.price = price;
    }
    public void save() { // Persistenta
        ...
    }
}
```

Efectele întretăierii intereselor

- **Clase dificil de schimbat**
Rescrierea mecanismului de autorizare implică rescrierea tuturor claselor care îl utilizează.
- **Cod care nu poate fi refolosit**
Clasa `Product` este legată de mecanismele de logging, autorizare, persistență.
- **Dificultăți în a asigura mentenanța:**
 - testare
 - depanare
 - optimizare

Soluții OOP



Scopul:

- Eliminarea / Diminuarea efectelor întretăierii intereselor
- Separarea aspectelor algoritmice de structura entităților

Soluții OOP:

- **Template Pattern**
- **Visitor Pattern**



Template Pattern

```
public abstract class Game {
    protected int playersCount;
    abstract void initializeGame();
    abstract void makePlay(int player);
    abstract boolean endOfGame();
    abstract void printWinner();

    final void playOneGame(int playersCount) { // <-- Metoda sablon
        this.playersCount = playersCount;
        initializeGame();
        int j = 0;
        while (!endOfGame()) {
            makePlay(j);
            j = (j + 1) % playersCount;
        }
        printWinner();
    }
}

public class Chess extends Game { ... }
public class Poker extends Game { ... }
```

Visitor Pattern

```
interface Visitable {
    public void accept(Visitor visitor);
}

class Product implements Visitable {

    public void accept(Visitor visitor) {
        visitor.visit(this);
    } ...
}

class Algorithm implements Visitor {
    public void visit(Product p) {
        ...
    }
}
```

Soluția AOP

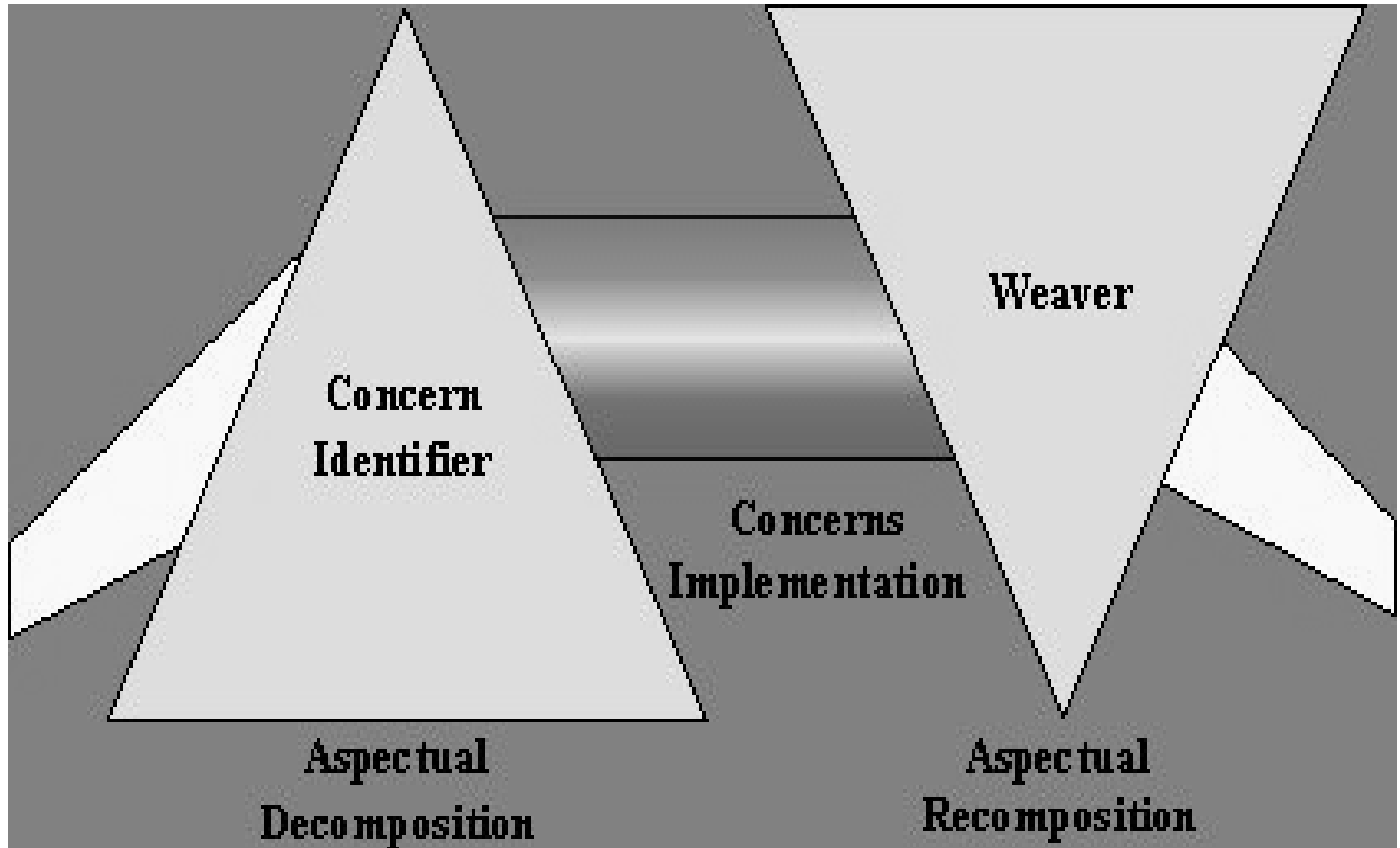
Ce este AOP ?

- 1996, Xerox Palo Alto Research Center (PARC), Gregor Kiczales
- Paradigmă de programare ce adresează separarea *concern*-urilor
- AOP nu are ca scop înlocuirea OOP ci oferă un nivel suplimentar de abstractizare
- Extinde platforme de programare existente
- Pune la dispoziție un mecanism (limbaj, extensie) pentru descrierea *concern*-urilor ce întrețin alte componente

Abordarea orientată aspect

- **Descompunere:** Analiza cerințelor va identifica două tipuri de funcționalități:
 - *common concerns* (product)
 - *crosscutting concerns* (logging)
- **Implementare:** se va face folosind instrumente specifice metodologiei folosite:
 - clase (Product)
 - aspecte (Logging)
- **Recompunere:** se va face folosind un **integrator (weaver)** pentru care trebuie definite **reguli de compunere**

Abordarea orientată aspect



Funcționarea unui integrator



Intrare

- Clasa Product
- Clasa Logger
- Reguli de integrare: "memorează schimbarea prețului"

Weaver

Rezultat

- Clasa ProductWithLogging



Tipuri de integratoare

- **Compile-time** (Compiler)
- **Link-time** (Linker)
- **Load-time** (ClassLoader)
- **Run-time** (Mașina virtuală)

Anatomia AOP

Limbaje de programare orientat pe aspecte: AspectJ, JBoss AOP, Spring AOP, etc

- **Specificații** pentru:
 - Implementarea concern-urilor
 - Definirea regulilor de integrare
- **Implementare**



AspectJ



Ce este AspectJ ?

- Extensie *orientată aspect* a platformei de programare Java
- Sintaxa limbajului *Java-like*
- Dezvoltat inițial la Xerox PARC (2001)
- Disponibil în cadrul *Eclipse Foundation*
- Integrare cu medii de lucru (Eclipse, Netbeans, etc.)
- Standard de-facto pentru AOP
- "Crosscutting objects for better modularity"

Specificațiile limbajului

- **Joinpoint**
- **Pointcut**
- **Advice**
- Regulă de integrare: Pointcut + Advice
- **Aspect**

Instrumente: compiler (ajc), debugger (ajdb), documentation generator (ajdoc), program structure browser (ajbrowser)

HelloWorld



```
// HelloWorld.java
public class HelloWorld {

    public static void say(String message) {
        System.out.println(message);
    }

    public static void sayToPerson(String message, String name) {
        System.out.println(name + ", " + message);
    }
}
```



MannersAspect

```
// MannersAspect.java
public aspect MannersAspect {
    pointcut callSayMessage() :
        call(public static void HelloWorld.say*(..));

    before() : callSayMessage() {
        System.out.println("Good day!");
    }

    after() : callSayMessage() {
        System.out.println("Thank you!");
    }
}
```

Puncte de legătura (*joinpoints*)

Joinpoint = o locație bine definită în codul primar unde un concern va întretăia aplicația.

- Apelul / execuția unei metode
- Apelul / execuția unui constructor
- Accesări ale unei proprietăți (citire / scriere)
- Tratarea erorilor
- Inițializarea statică a claselor

Pointcuts

Pointcut = construcție sintactică care specifică un punct de legătură și expune contextul acestuia.

- `call(MethodOrConstructorSignature)`
- `execution(MethodOrConstructorSignature)`
- `get(FieldSignature)`,
`set(FieldSignature)`
- `handler(ExceptionTypePattern)`
- `staticinitialization(TypePattern)`

Exemple de puncte de tăiere

```
call(public void MyClass.myMethod(String))  
call(* *.myMethod(...))  
call(MyClass.new())
```

```
execution(MyClass+.new(...))
```

```
get(PrintStream System.out)  
set(int MyClass.x)
```

```
handler(IOException+)  
handler(RemoteException)  
handler(CreditCard*)
```

```
staticinitialization(MyClass)  
staticinitialization(MyClass+)
```

Advice

Advice = secvență de cod ce trebuie executată la un moment specificat de un pointcut:

- **before**

- **after**

- **around**

```
before() : call(public * MyClass.*(..)) {
    System.out.println("Before: " + thisJoinPoint + " " +
        System.currentTimeMillis());
}
after() : call(public * MyClass.*(..)) {
    System.out.println("After: " + thisJoinPoint + " " +
        System.currentTimeMillis());
}
```

Expunerea contextului

- **this**: Obiectul apelant
- **target**: Obiectul apelat
- **args**: Argumentele

```
void around(Connection conn) :
    call(Connection.close()) && target(conn) {

    if (enablePooling) {
        connectionPool.put(conn);
    } else {
        proceed();
    }
}
```

Aspect

Aspectele sunt create în aceeași manieră ca și clasele și permit încapsularea completă a codului legat de un anumit concern.

```
aspect LoggingAspect {
    public void log(float oldPrice, float newPrice) {
        System.out.println(oldPrice + " -> " + newPrice);
    }

    pointcut priceChange(Product prod, float price) :
        call(public * Product.setPrice(float)) &&
        args(price) && target(prod);

    before(Product prod, float price): priceChange(prod, price) {
        log(prod.getPrice(), price);
    }
}
```

Aspect vs. Clasă (1)

Aspects are similar to classes because...

- aspects have type
- aspects can extend classes and other aspects
- aspects can be abstract or concrete
- non-abstract aspects can be instantiated
- aspects can have static and non-static state and behavior
- aspects can have fields, methods, and types as members
- the members of non-privileged aspects follow the same accessibility rules as those of classes

Aspect vs. Clasă (2)

Aspects are different than classes because...

- aspects can additionally include as members pointcuts, advice, and inter-type declarations;
- aspects can be qualified by specifying the context in which the non-static state is available
- aspects can't be used interchangeably with classes
- aspects don't have constructors or finalizers, and they cannot be created with the new operator; they are automatically available as needed.
- privileged aspects can access private members of other types



Scenarii de utilizare a aspectelor



Tracing

Tracing = urmărirea fluxului informațiilor - al apelurilor de metode făcute în timpul execuției.

```
public aspect Trace {
    pointcut method() : execution(* *.*(..));
    before() : method() {
        System.out.println("--> enter " +
            thisJoinPoint.getSignature() + "method");
    }
    after() : method() {
        System.out.println("<-- exit " +
            thisJoinPoint.getSignature() + "method");
    }
}
```

Verificarea condițiilor

Validarea și prelucrarea argumentelor primite de metode.

```
public aspect ArgumentValidator {

    pointcut method(String s) :
        call(public void myClass.func(String)) && args(s);

    void around(String s) : method(s) {
        if (s == null || s == "")
            proceed("empty")
        else if (s.length > 64) {
            proceed("string too long")
        }
        else
            proceed(s);
    }
}
```

Logging

Logging = Înregistrarea cronologică a unor evenimente (apeluri de metode, excepții) cu scopul de a analiza un sistem software aflat în producție.

```
public aspect ExceptionLogging {
    pointcut someMethod() : execution(* *.*(..));
    after() throwing : someMethod() {
        System.out.println("Exception thrown at: " +
            thisJoinPoint.getSignature() + "method");
    }

    pointcut someHandler() : handler(Exception);
    before() : someHandler() {
        System.out.println("Exception caught at: " +
            thisJoinPoint.getSignature() + "method");
    }
}
```

Profiling

```
public aspect Profiling {
    long t1, t2;
    pointcut method() : execution(* *.*(..));
    before() : method() {
        t1 = System.currentTimeMillis();
    }
    after() : method() {
        t2 = System.currentTimeMillis();
        System.out.println(thisJoinPoint.getSignature() + " " + (t2 - t1));
    }
}
```

Autorizare

```
public aspect Authorization {
    pointcut name(User user, String s) :
        (setFirstName(User, String) || setLastName(User, String))
        && args(user, s);

    pointcut contact(User user, String s) :
        (setPhone(User, String) || setEmail(User, String))
        && args(user, s);

    around(User user, String s) : name(user, s) {
        if (user.hasPermission("name.change")) {
            proceed(user, s);
        }
    }
    around(User user, String s) : contact(user, s) {
        if (user.hasPermission("contact.change")) {
            proceed(user, s);
        }
    }
}
```

Recalculare coș compărături

Aspect pentru urmărirea schimbărilor stărilor ce afectează valoarea coșului.

```
private boolean ShoppingBasket.isDirty = false;
```

```
pointcut basketChange(ShoppingBasket basket):  
    (execution(public void addItem(ShoppingBasketItem, ..))  
    || execution(public void remove*(..))) && this(basket);
```

```
pointcut groupChange(ShoppingBasketGroup group):  
    (execution(public void setDeliveryMedium(..))  
    || execution(public void setCopies(..))) && this(group);
```

```
after(ShoppingBasket basket) returning : basketChange(basket) {  
    basket.isDirty = true;  
}
```

Subject/Observer

Folosirea aspectelor pentru a implementa diverse șabloane de proiectare.

- Interfața `Observer`
- Interfața `Subject`
- Protocolul abstract `SubjectObserverProtocol`
- Clase concrete de tip `Observer` și `Subject`
- Implementare concretă a protocolului

Observer, Subject

```
interface Subject {  
    void addObserver(Observer obs);  
    void removeObserver(Observer obs);  
    Vector getObservers();  
    Object getData();  
}
```

```
interface Observer {  
    void setSubject(Subject s);  
    Subject getSubject();  
    void update();  
}
```

SubjectObserverProtocol

```
abstract aspect SubjectObserverProtocol {
    private Vector Subject.observers = new Vector();
    private Subject Observer.subject = null;

    abstract pointcut stateChanges(Subject s);
    after(Subject s): stateChanges(s) {
        for (int i = 0; i < s.getObservers().size(); i++) {
            ((Observer)s.getObservers().elementAt(i)).update();
        }
    }
    public void Subject.addObserver(Observer obs) {
        observers.addElement(obs);
        obs.setSubject(this);
    }
    public void Subject.removeObserver(Observer obs) {
        observers.removeElement(obs);
        obs.setSubject(null);
    }
    public Vector Subject.getObservers() { return observers; }
    public void      Observer.setSubject(Subject s) { subject = s; }
    public Subject  Observer.getSubject() { return subject; }
}
```

Button

```
class Button extends java.awt.Button {
    Button() {
        super();
        setLabel("cycle");
        addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                Button.this.click();
            }
        });
    }
    public void click() {}
}
```

ColorLabel

```
class ColorLabel extends Label {  
  
    ColorLabel() {  
        super();  
    }  
  
    final static Color[] colors = {Color.red, Color.blue,  
                                    Color.green, Color.magenta};  
  
    private int colorIndex = 0;  
    private int cycleCount = 0;  
    void colorCycle() {  
        cycleCount++;  
        colorIndex = (colorIndex + 1) % colors.length;  
        setBackground(colors[colorIndex]);  
        setText("" + cycleCount);  
    }  
}
```

SubjectObserverProtocolImpl

Subiectul: Button, **observatorul:** ColorLabel

```
aspect SubjectObserverProtocolImpl extends SubjectObserverProtocol {  
  
    declare parents: Button implements Subject;  
    public Object Button.getData() { return this; }  
  
    declare parents: ColorLabel implements Observer;  
    public void ColorLabel.update() {  
        colorCycle();  
    }  
  
    pointcut stateChanges(Subject s):  
        target(s) &&  
        call(void Button.click());  
}
```

Concluzii

- Princiile AOP și OOP se completează reciproc
 - OOP adresează funcționalitățile principale
 - AOP adresează funcționalitățile ce taie sistemul
- Limbajele dedicate AOP extind platforme existente de programare
- Există interes crescut din partea industriei (JBoss)
- AOP = "The next big thing" ?