An Introduction to Object-Oriented Analysis and Design

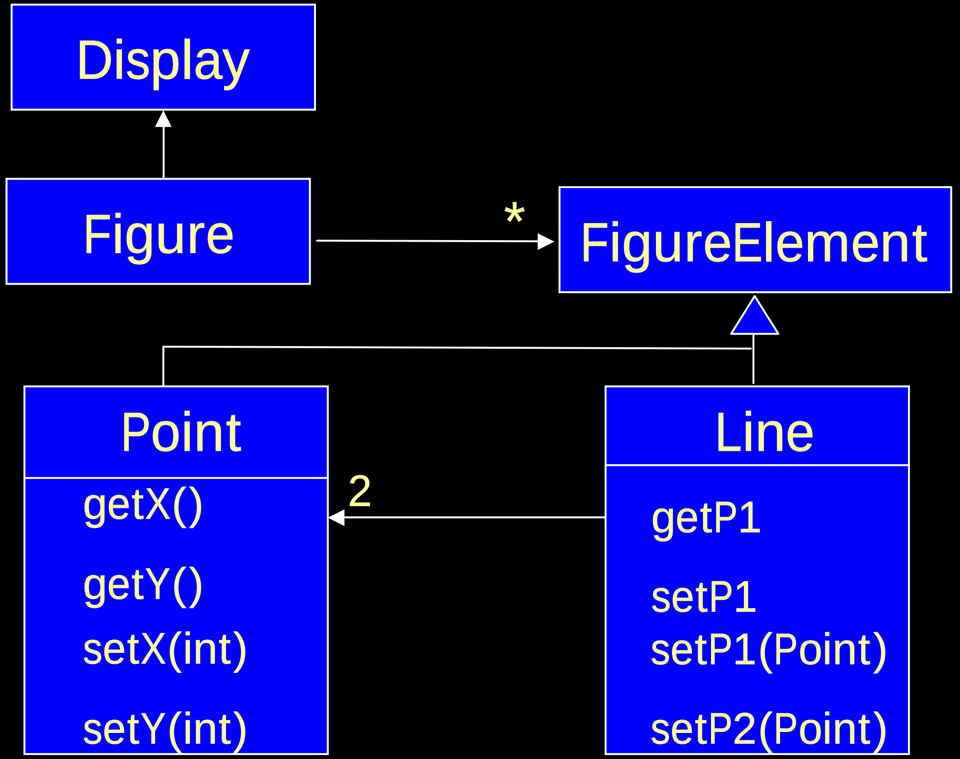
PART VI: Aspect Oriented Programming

Dr. 邱 明
Software School of Xiamen University
mingqiu@xmu.edu.cn
Fall, 2009

Example - Figure Editor

- A *figure* consists of several *figure elements*.
  - A figure element is either a *point* or a *line*.
- Figures are drawn on *Display*.
  - A point includes X and Y coordinates.
  - A line is defined as two points.

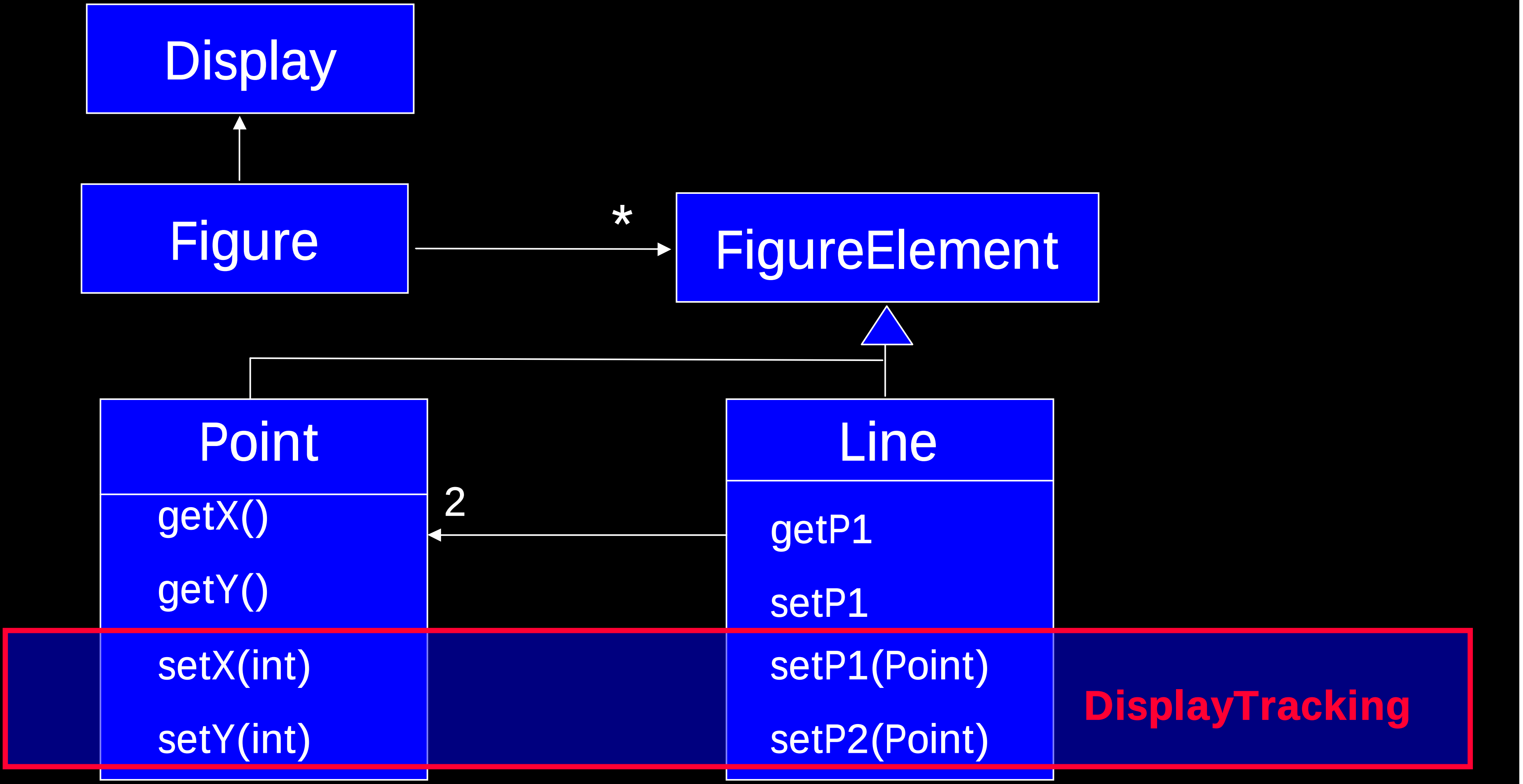
Example - Figure Editor - Design



Components are
- Cohesive
- Loosely Coupled
- Have well-defined interfaces (abstraction, encapsulation)

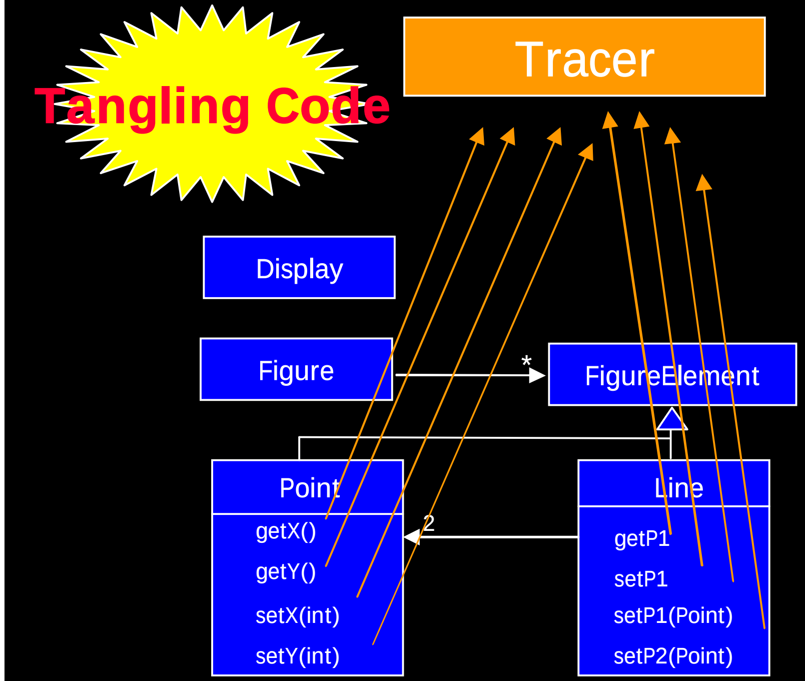
Crosscutting Concern - Example

Notify ScreenManager if a figure element moves



Example - Tracing

Scattered Concern



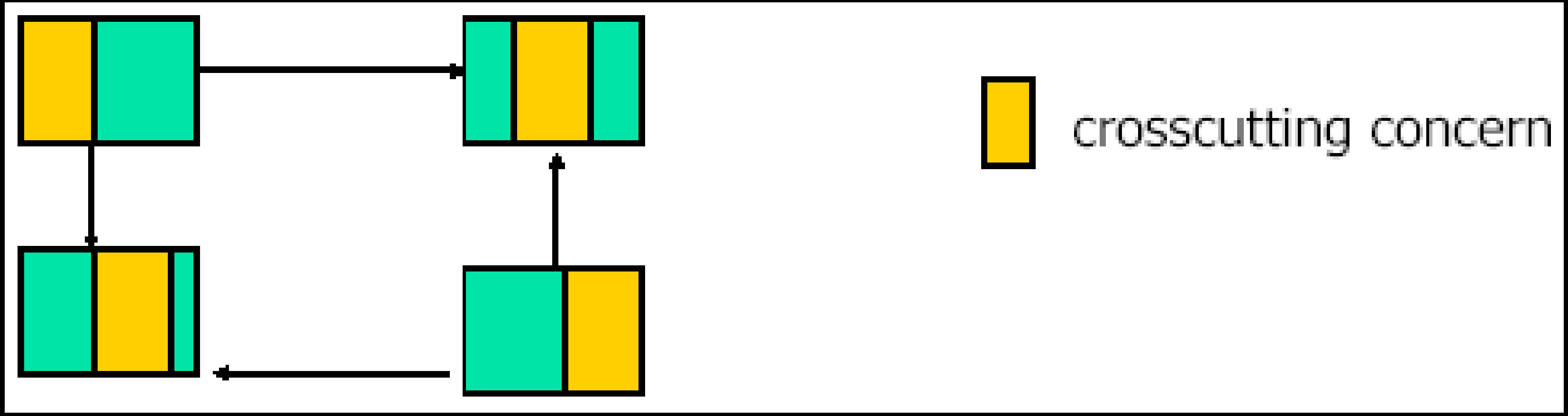
```
class Tracer {

  static void traceEntry(String str)
  {
        System.out.println(str);
  }
  static void traceExit(String str)
  {
        System.out.println(str);
  }
}
```

```
class Point {
 void setX(int x) {
 Tracer.traceEntry(“Entry Point.set”);
   _x = x;
 Tracer.traceExit(“Exit Point.set”);
}
}
```

```
class Line {
  void setP1(Point p1 {
   Tracer.traceEntry(“Entry Line.set”);
    _p1 = p1;
   Tracer.traceExit(“Exit Line.set”);
}
}
```

Crosscutting Concerns

- Concerns that naturally tend to be scattered over multiple components
- Connot be localized into single units
  - components, objects, procedures, functions
- If not appropriately coped with:
  - Scattered over multiple components
  - Tangled code per component



Crosscutting, Scattering and Tangling

- Crosscutting
  - Concern that inherently relates to multiple components
  - Results in scattered concern and tangled code
- Scattering
  - Single concern affects multiple modules
- Tangling
  - Multiple concerns are interleaved in a single module

The Cost of Crosscutting Concerns

- Reduced understandability
  - Redundant code in many places
  - Non-explicit structure
- Decreased adaptability
  - Have to find all the code involved
  - Have to be sure to change it consistently
  - Have to be sure not to break it by accident
  - New concerns cannot be easily added
- Decreased reusability
  - Component code is tangled with specific tangling code
- Decreased maintainability
  - ‘Ripple effect’

Example of Crosscutting Concerns

- Synchronization
- Real-time constraints
- Error-checking
- Object interaction constraints
- Memory management
- Persistency
- Security
- Caching
- Logging
- Monitoring
- Testing
- Domain specific optimization
- ...

Many crosscutting concerns may appear in one system

w Example: Distributed System Design

w Component interaction

w Synchronization

w Remote invocation

w Load balancing

w Replication

w Failure handling

w Quality of service

w Distributed transactions

What to Do...?

Historical Context

- Crosscutting concerns are new type of concerns that have not been (appropriately) detected/handled before.
- No explicit management until recently at programming level
- No explicit consideration in design methods
- No explicit consideration in process
- No explicit consideration in case tools

Aspect-Oriented Software Development

w Provides better separation of concerns by explicitly considering crosscutting concerns (as well)

w Does this by providing explicit abstractions for representing crosscutting concerns, i.e. aspects

w And composing these into programs, i.e. aspect weaving or aspect composing.

w As such AOSD improves modularity

w And supports quality factors such as

   § Maintainability, Adaptability, Reusability, Understandability

w ...

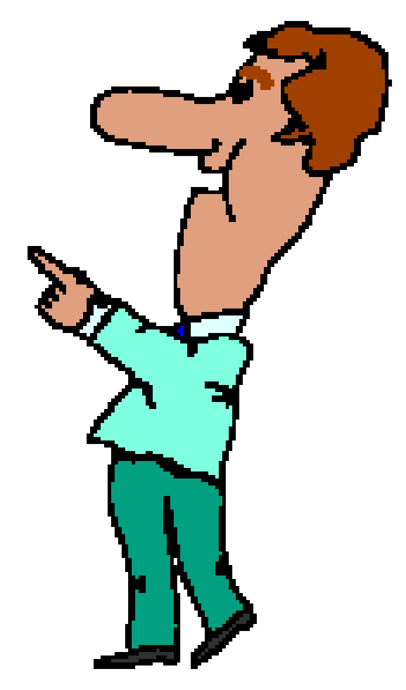
Impact of AOSD on Society...

w MIT Technology Review lists AOP as one of the top 10 emerging technologies that will change the world

§ –(MIT Technology Review, January 2001)

Basic AOSD Technologies

- Composition Filters (since 1991)
  - University of Twente, The Netherlands
- AspectJ (since 1997)
  - XEROX PARC, US
- DemeterJ/DJ (1993)
  - Northeastern University, US
- Multi-dimensional separation of Concerns/HyperJ (1999)
- JBoss AOP
  - Supported in JBoss AS 5



AspectJ

- A general purpose AO programming language
  - just as Java is a general-purpose OO language
- An integrated extension to Java
  - accepts all java programs as input
  - outputs .class files compatible with any JVM
  - integrated with tools
    - http://www.eclipse.org/aspectj/

Example – Without AOP

```
class Line {
  private Point _p1, _p2;

  Point getP1() { return _p1; }
  Point getP2() { return _p2; }

  void setP1(Point p1) {
    Tracer.traceEntry(“entry setP1”);
    _p1 = p1;
    Tracer.traceExit(“exit setP1”);
  }

 void setP2(Point p2) {
    Tracer.traceEntry(“entry setP2”);
    _p2 = p2;
    Tracer.traceExit(“exit setP2”);
  }
class Point {
  private int _x = 0, _y = 0;

  int getX() { return _x; }
  int getY() { return _y; }

  void setX(int x) {
   Tracer.traceEntry(“entry setX”);

    _x = x;
   Tracer.traceExit(“exit setX”)
  }
  void setY(int y) {
   Tracer.traceEntry(“exit setY”);

    _y = y;
   Tracer.traceExit(“exit setY”);
  }
}
```

```
class Tracer {

  static void traceEntry(String str)
  {
          System.out.println(str);
  }
  static void traceExit(String str)
  {
          System.out.println(str);
  }
}
```

Tangling Code

Scattered Concern

Example – With AOP

```
class Line {
  private Point _p1, _p2;

  Point getP1() { return _p1; }
  Point getP2() { return _p2; }

  void setP1(Point p1) {
    _p1 = p1;
  }
  void setP2(Point p2) {
    _p2 = p2;
  }
}

class Point {
   private int _x = 0, _y = 0;

  int getX() { return _x; }
  int getY() { return _y; }

  void setX(int x) {
    _x = x;
  }
  void setY(int y) {
    _y = y;
  }
}
```

```
aspect Tracing {

  pointcut traced():
    call(* Line.* ||
    call(* Point.*);

  before(): traced() {
    println(“Entering:” +
            thisjopinpoint);

  void println(String str)
  {<write to appropriate stream>}

  }
}
```

Aspect is defined in a separate module
Crosscutting is localized
No scattering; No tangling
Improved modularity

Aspect Language Elements

- Join Point (JP) model
    - certain principled points in program execution
        - such as method calls, field accesses, and object construction
- Means of identifying JPs
    - picking out join points of interest (predicate)
    - ***pointcuts:*** set of join points
- Means of specifying behavior at JPs
    - what happens
    - ***advice*** declarations
        - the additional code that you want to apply to your existing model.

Joinpoints

- method call join points
  - when a method is called
- method reception join points
  - when an object receives a message
- method execution join points
  - when the body of code for an actual method executes
- field get joint point
  - when a field is accessed
- field set joint point
  - when a field is set
- exception handler execution join point
  - when an exception handler executes
- object creation join point
  - when an instance of a class is created

Some primitive pointcuts

w call(Signature)

  § picks out method or constructor call based on Signature

w execution(Signature)

  § picks out a method or constructor execution join point based on Signature

w get(Signature)

  § picks out a field get join point based on Signature

w set(Signature)

  § picks out a field set join point based on Signature

w handles(TypePattern)

  § picks out an exception handler of any of the Throwable types of TypePattern

w instanceOf(ClassName)

  § picks out join points of currently executing objects of class ClassName

w within(ClassName)

  § picks out join points that are in code contained in ClassName

w withinCode(Signature)

  § picks out join points within the member defined by methor or constructor (Signature)

w cflow(pointcut)

  § picks out all the join points in the control flow of the join points picked out by the pointcut

Advice

- Piece of code that attaches to a pointcut and thus injects behavior at all joinpoints selected by that pointcut.
- example:

  before (args): pointcut
  { Body }

  where *before* represents a before advice type (see next slide).

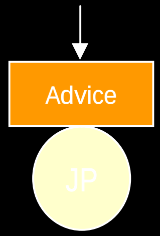
- Can take parameters with pointcuts

Advice Types

Advice code executes

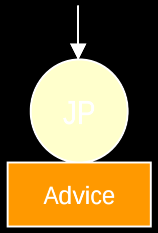
- *before*, code is injected before the joinpoint

  before (args): pointcut
    { Body }



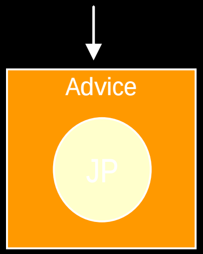
- *after*, code is injected after the joinpoint

  after (args): pointcut
    { Body }



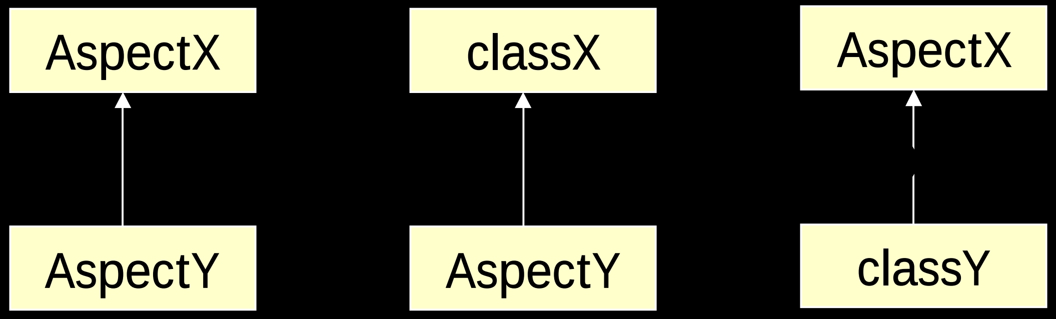
- *around,* code is injected around (in place of) code from joinpoint

  ReturnType around (args): pointcut
    { Body }



Aspect

- A modular unit of cross-cutting behavior.
- Like a class, can have methods, fields, initializers.
- can be abstract, inherit from classes and abstract aspects and implement interfaces.
- encapsulates pointcuts and advices
- can introduce new methods / fields to a class



Example - AspectJ

```
class Line {
  private Point _p1, _p2;

  Point getP1() { return _p1; }
  Point getP2() { return _p2; }

  void setP1(Point p1) {
    _p1 = p1;
  }
  void setP2(Point p2) {
    _p2 = p2;
  }
}

class Point {
   private int _x = 0, _y = 0;

  int getX() { return _x; }
  int getY() { return _y; }

  void setX(int x) {
    _x = x;
  }
  void setY(int y) {
    _y = y;
  }
}
```

```
aspect Tracing {

  pointcut traced():
    call(* Line.* ||
    call(* Point.*);

  before(): traced() {
    println(“Entering:” +
            thisjopinpoint);

   after(): traced() {
    println(“Exit:” +
            thisjopinpoint);


  void println(String str)
  {<write to appropriate stream>}

  }
}
```

aspect

pointcut

advice

Code Weaving

w Before compile-time (pre-processor)

w During compile-time

w After compile-time

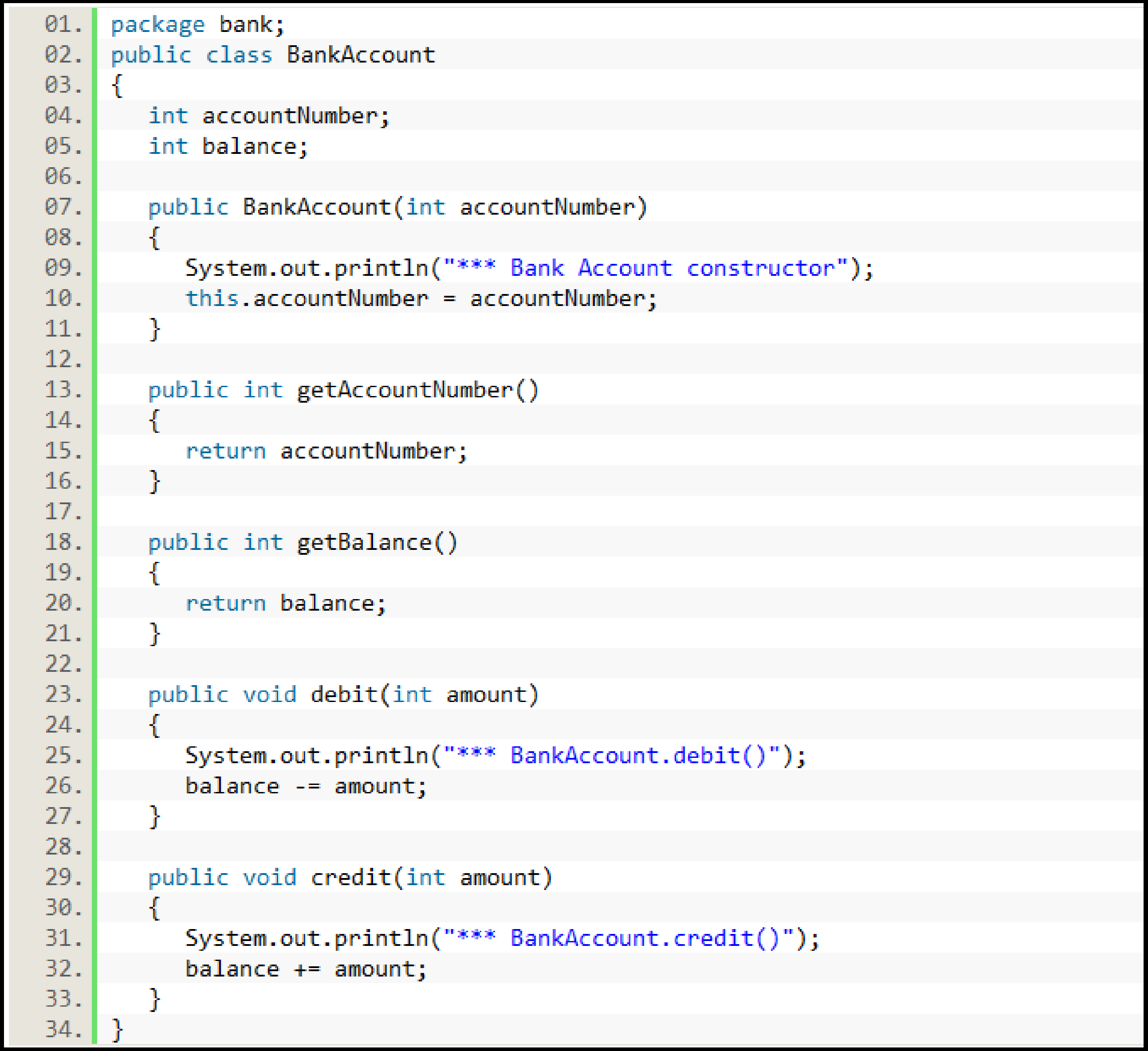
w At load time

w At run-time

JBoss AOP

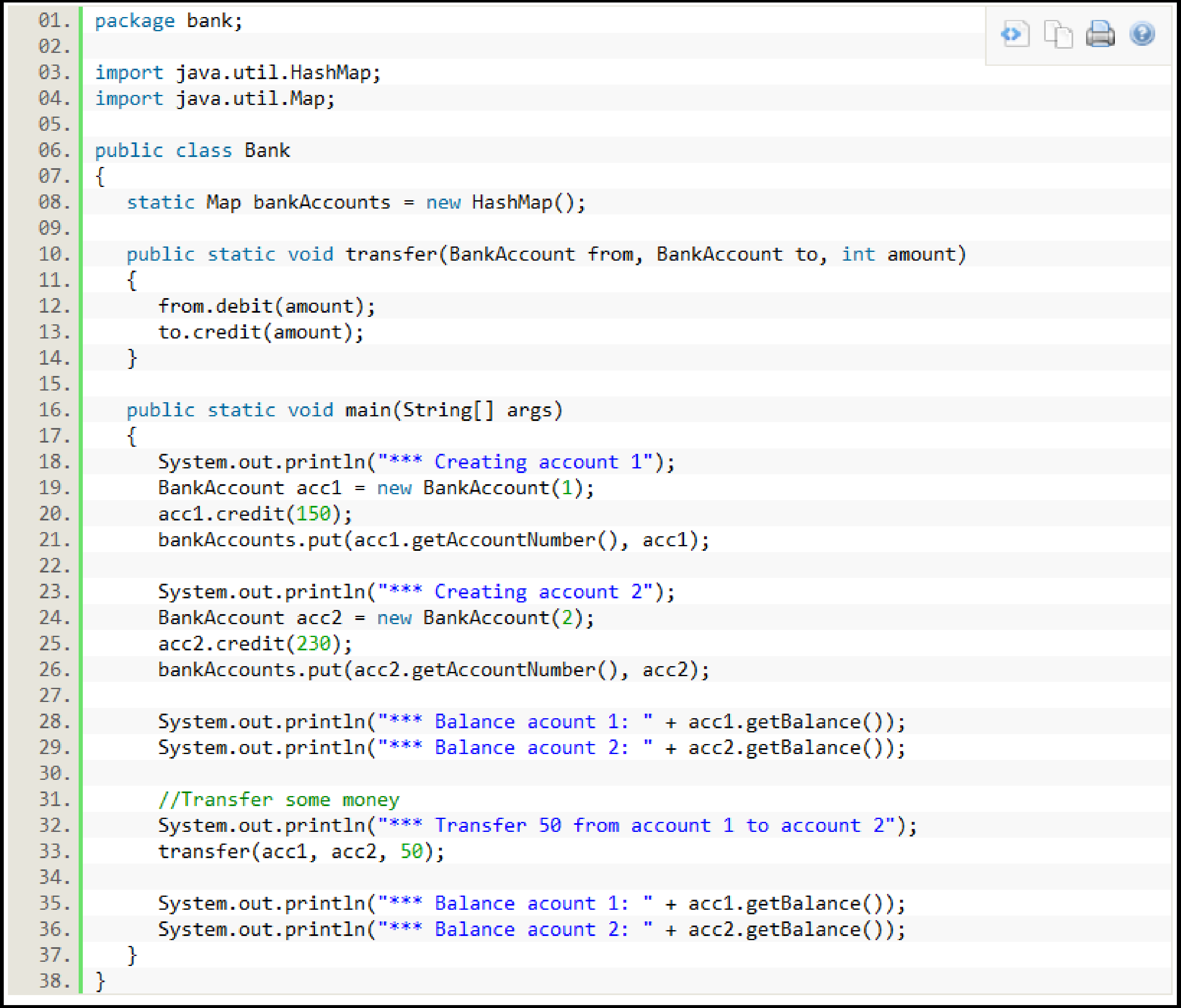
- a 100% Pure Java aspected oriented framework usuable in any programming environment or tightly integrated with JBoss application server.

JBoss AOP - Example



```java
package bank;
public class BankAccount
{
   int accountNumber;
   int balance;

   public BankAccount(int accountNumber)
   {
      System.out.println("*** Bank Account constructor");
      this.accountNumber = accountNumber;
   }

   public int getAccountNumber()
   {
      return accountNumber;
   }

   public int getBalance()
   {
      return balance;
   }

   public void debit(int amount)
   {
      System.out.println("*** BankAccount.debit()");
      balance -= amount;
   }

   public void credit(int amount)
   {
      System.out.println("*** BankAccount.credit()");
      balance += amount;
   }
}
```

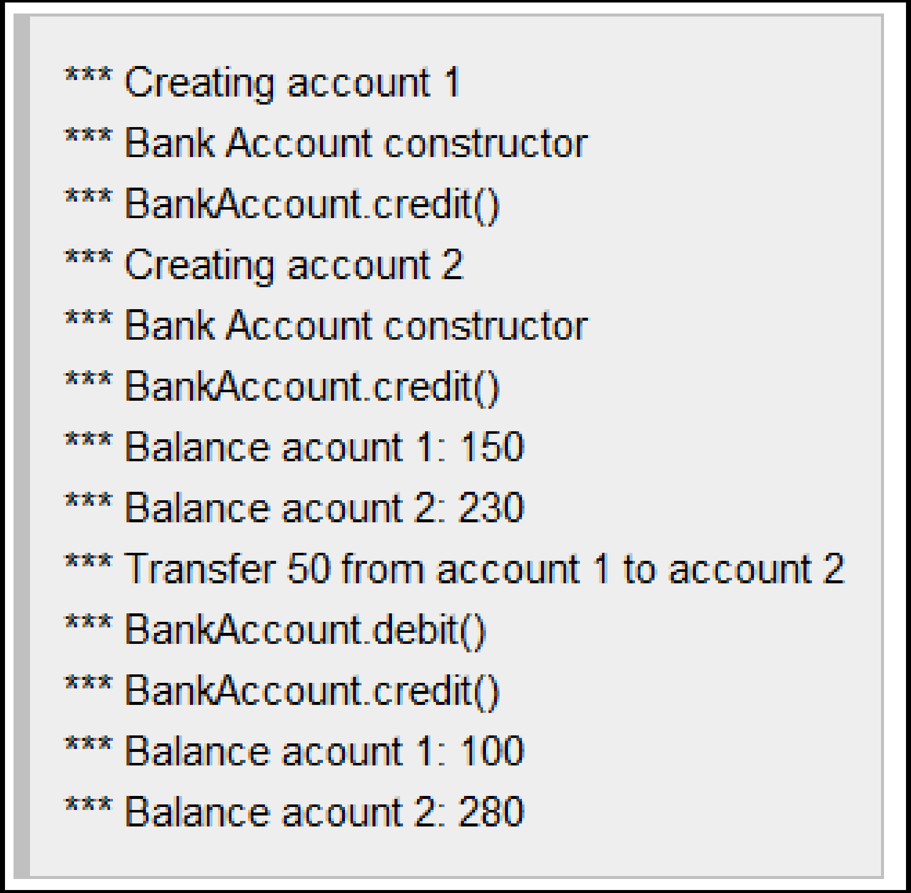
JBoss AOP - Example



```java
package bank;

import java.util.HashMap;
import java.util.Map;

public class Bank
{
   static Map bankAccounts = new HashMap();

   public static void transfer(BankAccount from, BankAccount to, int amount)
   {
      from.debit(amount);
      to.credit(amount);
   }

   public static void main(String[] args)
   {
      System.out.println("*** Creating account 1");
      BankAccount acc1 = new BankAccount(1);
      acc1.credit(150);
      bankAccounts.put(acc1.getAccountNumber(), acc1);

      System.out.println("*** Creating account 2");
      BankAccount acc2 = new BankAccount(2);
      acc2.credit(230);
      bankAccounts.put(acc2.getAccountNumber(), acc2);

      System.out.println("*** Balance acount 1: " + acc1.getBalance());
      System.out.println("*** Balance acount 2: " + acc2.getBalance());

      //Transfer some money
      System.out.println("*** Transfer 50 from account 1 to account 2");
      transfer(acc1, acc2, 50);

      System.out.println("*** Balance acount 1: " + acc1.getBalance());
      System.out.println("*** Balance acount 2: " + acc2.getBalance());
   }
}
```

JBoss AOP - Example

output of the program

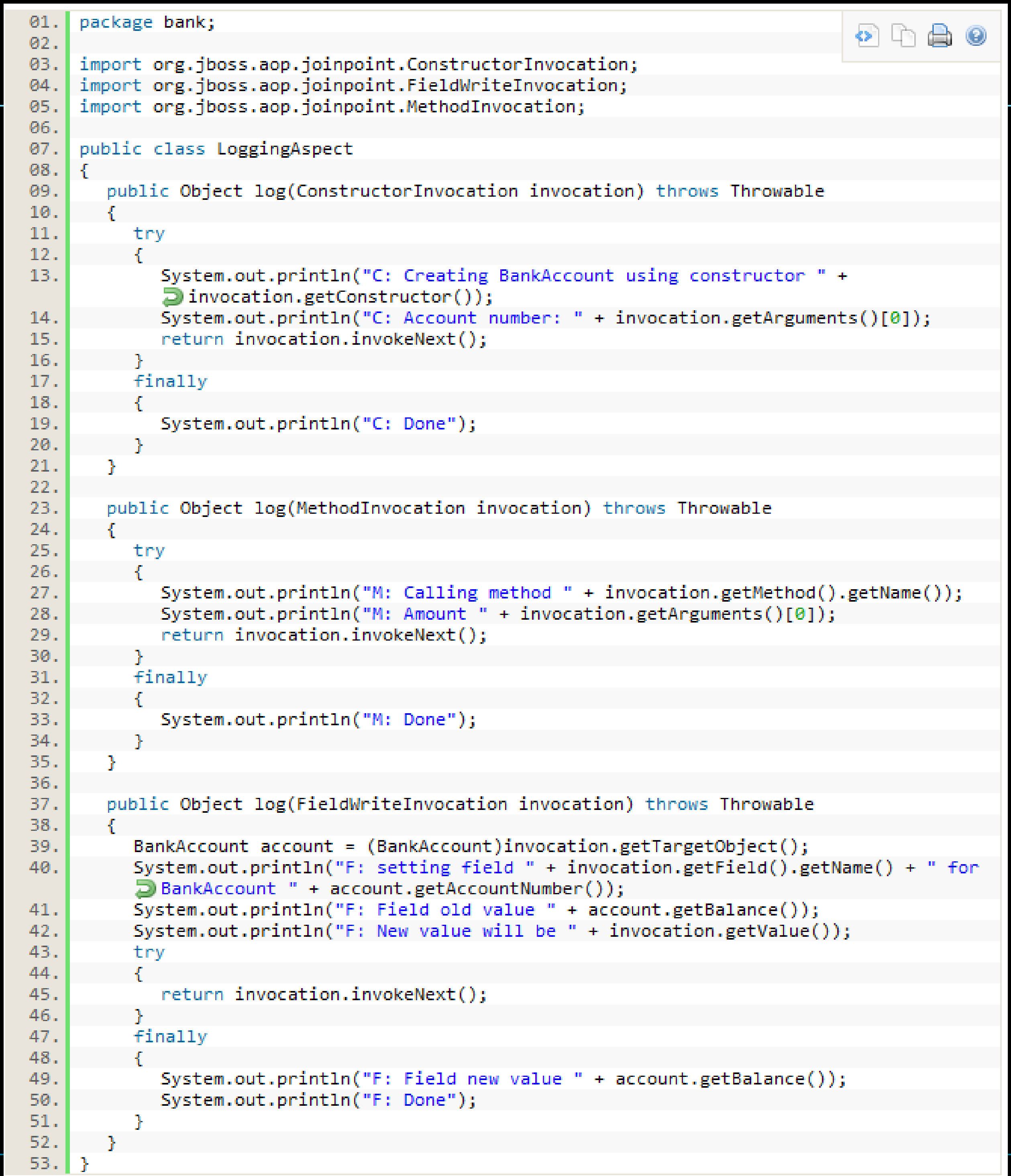


```
*** Creating account 1
*** Bank Account constructor
*** BankAccount.credit()
*** Creating account 2
*** Bank Account constructor
*** BankAccount.credit()
*** Balance acount 1: 150
*** Balance acount 2: 230
*** Transfer 50 from account 1 to account 2
*** BankAccount.debit()
*** BankAccount.credit()
*** Balance acount 1: 100
*** Balance acount 2: 280
```

JBoss AOP - Example

w **Logging as a Cross-Cutting Concern**

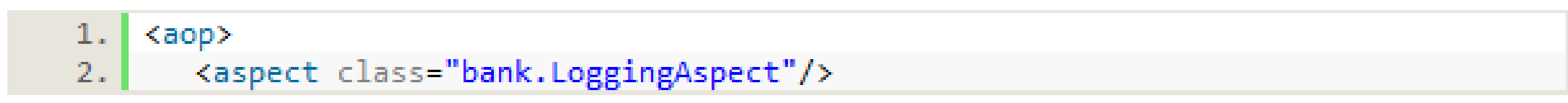
- § **Add some logging whenever an object is created, when its fields are set, and when its methods are called**



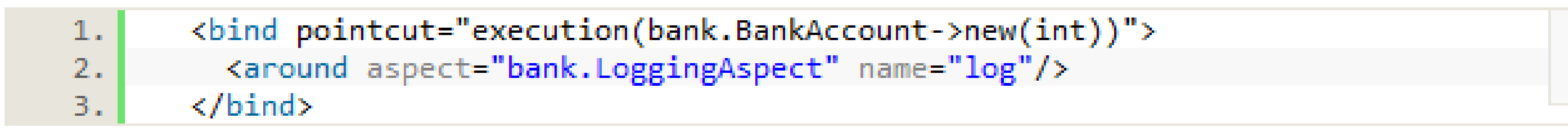
```java
package bank;

import org.jboss.aop.joinpoint.ConstructorInvocation;
import org.jboss.aop.joinpoint.FieldWriteInvocation;
import org.jboss.aop.joinpoint.MethodInvocation;

public class LoggingAspect
{
   public Object log(ConstructorInvocation invocation) throws Throwable
   {
      try
      {
         System.out.println("C: Creating BankAccount using constructor " + invocation.getConstructor());
         System.out.println("C: Account number: " + invocation.getArguments()[0]);
         return invocation.invokeNext();
      }
      finally
      {
         System.out.println("C: Done");
      }
   }

   public Object log(MethodInvocation invocation) throws Throwable
   {
      try
      {
         System.out.println("M: Calling method " + invocation.getMethod().getName());
         System.out.println("M: Amount " + invocation.getArguments()[0]);
         return invocation.invokeNext();
      }
      finally
      {
         System.out.println("M: Done");
      }
   }

   public Object log(FieldWriteInvocation invocation) throws Throwable
   {
      BankAccount account = (BankAccount)invocation.getTargetObject();
      System.out.println("F: setting field " + invocation.getField().getName() + " for BankAccount " + account.getAccountNumber());
      System.out.println("F: Field old value " + account.getBalance());
      System.out.println("F: New value will be " + invocation.getValue());
      try
      {
         return invocation.invokeNext();
      }
      finally
      {
         System.out.println("F: Field new value " + account.getBalance());
         System.out.println("F: Done");
      }
   }
}
```

JBoss AOP - Example

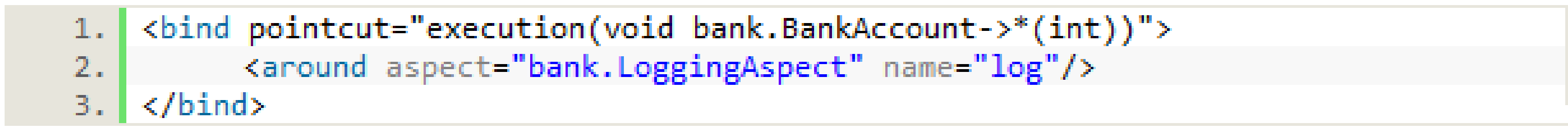
- Declare aspects in jboss-aop.xml



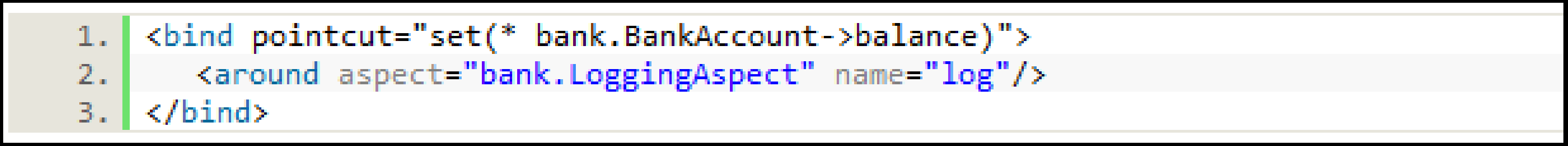
```
<aop>
   <aspect class="bank.LoggingAspect"/>
```



```
   <bind pointcut="execution(bank.BankAccount->new(int))">
     <around aspect="bank.LoggingAspect" name="log"/>
   </bind>
```



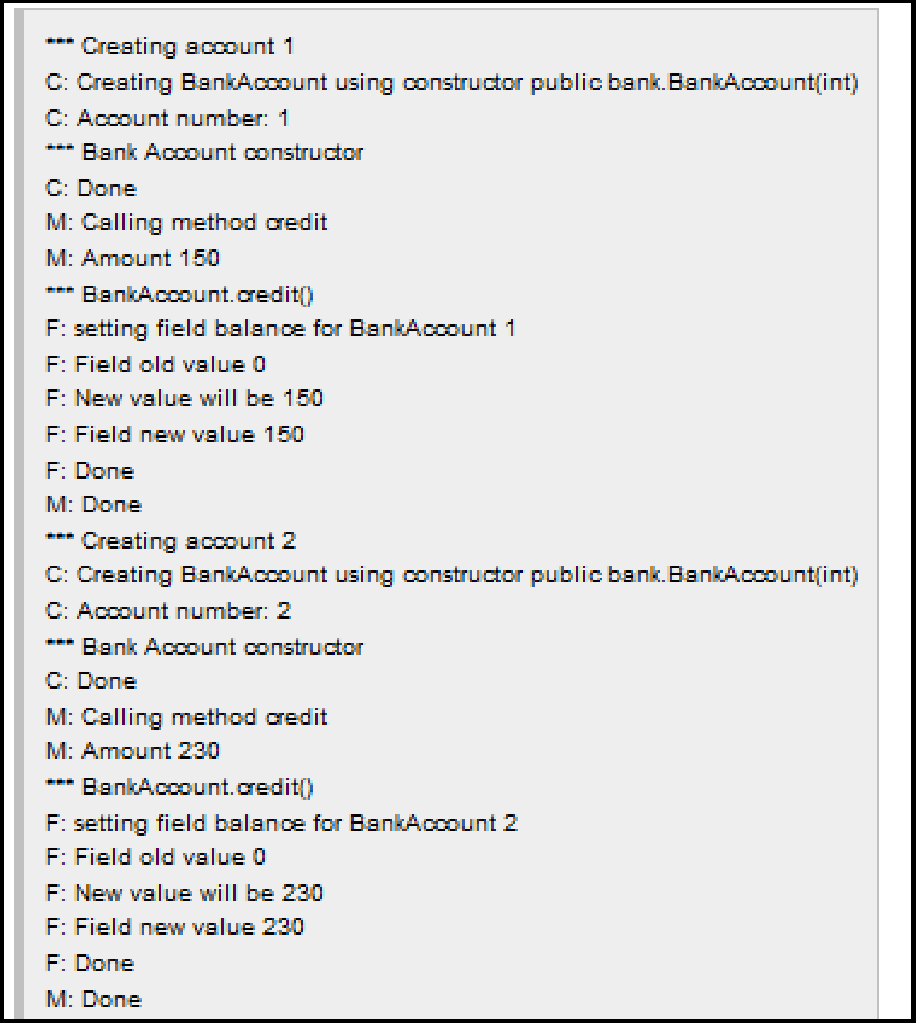
```
<bind pointcut="execution(void bank.BankAccount->*(int))">
      <around aspect="bank.LoggingAspect" name="log"/>
</bind>
```



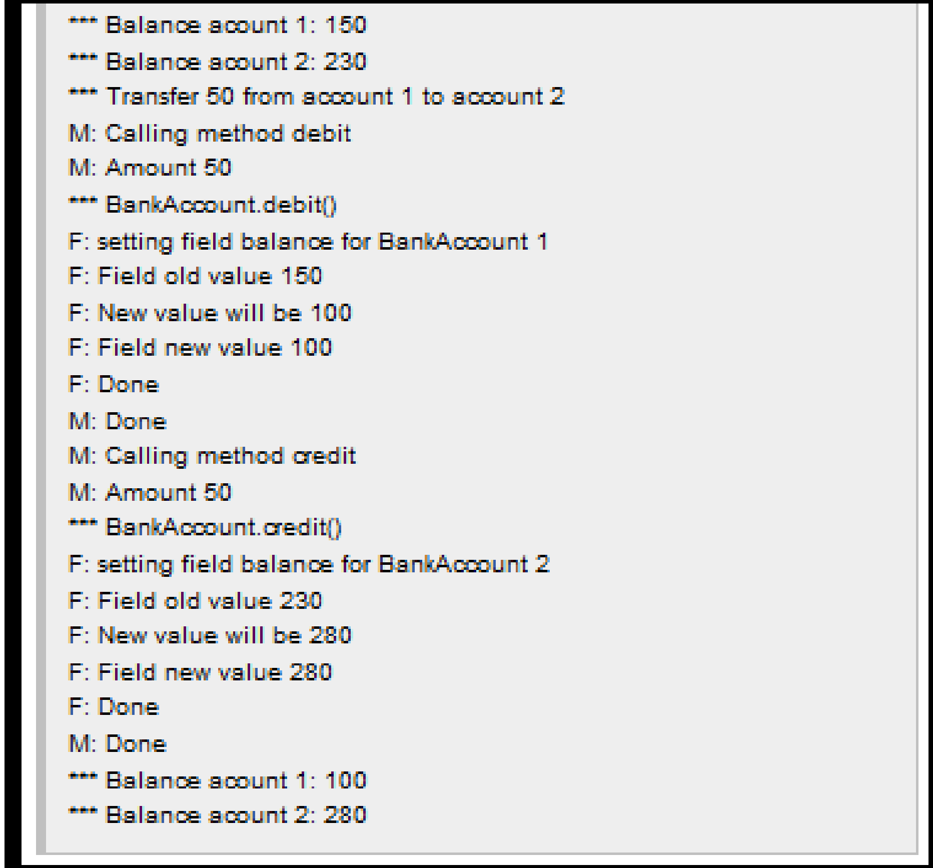
```
<bind pointcut="set(* bank.BankAccount->balance)">
   <around aspect="bank.LoggingAspect" name="log"/>
</bind>
```

JBoss AOP - Example

w Output of the program



```
*** Creating account 1
C: Creating BankAccount using constructor public bank.BankAccount(int)
C: Account number: 1
*** Bank Account constructor
C: Done
M: Calling method credit
M: Amount 150
*** BankAccount.credit()
F: setting field balance for BankAccount 1
F: Field old value 0
F: New value will be 150
F: Field new value 150
F: Done
M: Done
*** Creating account 2
C: Creating BankAccount using constructor public bank.BankAccount(int)
C: Account number: 2
*** Bank Account constructor
C: Done
M: Calling method credit
M: Amount 230
*** BankAccount.credit()
F: setting field balance for BankAccount 2
F: Field old value 0
F: New value will be 230
F: Field new value 230
F: Done
M: Done
```



```
*** Balance acount 1: 150
*** Balance acount 2: 230
*** Transfer 50 from account 1 to account 2
M: Calling method debit
M: Amount 50
*** BankAccount.debit()
F: setting field balance for BankAccount 1
F: Field old value 150
F: New value will be 100
F: Field new value 100
F: Done
M: Done
M: Calling method credit
M: Amount 50
*** BankAccount.credit()
F: setting field balance for BankAccount 2
F: Field old value 230
F: New value will be 280
F: Field new value 280
F: Done
M: Done
*** Balance acount 1: 100
*** Balance acount 2: 280
```

Conclusion

w Crosscutting concerns are typically scattered over several modules and result in tangled code.

w This reduces the modularity and as such the quality of the software system.

w AOSD provides explicit abstractions mechanisms to represent these so-called aspects and compose these into programs

w This increases the modularity of systems.