introduction to AspectJ

CS 119

a program instrumentation and monitoring framework
monitoring

event generation

instrumentation
specification

behavior
specification

event evaluation
why AspectJ?

• so, … monitoring a program’s execution requires these two elements:
  – instrumentation specification
  – behavior specification

• both elements are provided by AspectJ:
  – instrumentation specification
    • AspectJ’s extension to Java
  – behavior specification
    • Java
outline

• this lesson : introducing the language
• next lesson : monitoring with AspectJ
very simplified view of AOP

while(more())
{
    ...
    send(msg);
    ...
}

program

while(more())
{
    ...
    check(msg);
    send(msg);
    ...
}

aspect

when send(msg)
{
    check(msg);
}

weaver

informal notation
that's it

except for all the details, motivation, usage, ...
AspectJ

- AspectJ, launched 1998 at Xerox PARC
- It is an extension of Java
- A new way of modularizing programs compared to object-oriented programming
- Emphasis on separating out cross-cutting concerns. Logging for example is a concern. That is, code for one aspect of the program is collected together in one place
- We shall use it purely for monitoring, and we do not focus on the broader application of AOP as a programming paradigm
- We will, however, briefly explain the more general purpose of AOP
- The AspectJ compiler is free and open source
- AspectJ works with Eclipse, and other IDEs
- Outputs .class files compatible with any JVM
basic mechanisms

• join points
  • points in a Java program

• four additions to Java
  – pointcut
    • picks out join points and values at those points
      – primitive and user-defined pointcuts
  – advice
    • additional action to take at join points matching a pointcut
  – inter-type declaration
    • add fields, methods to classes
  – aspect
    • a modular unit of crosscutting behavior
      – normal Java declarations
      – pointcut definitions, advices, inter-type declarations
privileged aspect OrdinaryAspect percflow (call(void Foo.m()))
{
    declare warning: call(*1.Foo+.new(..)): “don’t new Foo+”;

    pointcut notKeywords():
        call(void *if*..*while*)(int,boolean,*for*));
    pointcut hasSpecialIf(): if (Tracing.isEnabled());

    after (Point p) returning(int x):
        target(p) && call(int getX())
    {
        System.out.println(“Returning ” + x + “ for p = ” + p);
    }

    int OrdinaryJavaClass.incr2(int i) { return(x+2);}

    int x;

    static int incr3(int x) { return x+3;}
}
Resources

- [http://www.eclipse.org/aspectj](http://www.eclipse.org/aspectj)  
  
  optional reading
good modularity

- XML parsing in org.apache.tomcat
  - red shows relevant lines of code
  - nicely fits in one box
bad modularity

• where is logging in org.apache.tomcat
  – red shows lines of code that handle logging
  – not in just one place
  – not even in a small number of places
two central problems
AOP tries to solve

code trangling:
one module
many concerns

example:
logging

code scattering:
one concern
many modules
two central problems
AOP tries to solve

code tangled: one module many concerns
code scattering: one concern many modules

example: logging
class Power {
    int balance;

    void deposit(int amount) {
        balance = balance + amount;
    }

    boolean withdraw(int amount) {
        if (balance - amount > 0) {
            balance = balance - amount;
            return true;
        } else return false;
    }
}
class Logger {
    private PrintStream stream;

    Logger() {
        ... create stream
    }

    void log(String message) {
        stream.println(message);
    }
}
class Power {
    int balance;
    Logger logger = new Logger();

    void deposit(int amount) {
        balance = balance + amount;
        logger.log("deposit amount: "+ amount);
    }

    boolean withdraw(int amount) {
        if (balance - amount >= 0) {
            balance = balance - amount;
            logger.log("withdraw amount: "+ amount);
            return true;
        } else return false;
    }
}

logging the traditional way
the problem

• the flow of the core logic in the Power module gets obscured, harder to follow, the core logic is tangled with the logging logic.
• the logging code gets scattered throughout the code base
  – lots of typing
  – big picture (in one place) is missing
  – difficult to find what is logged where and how
  – difficult to change logging formats
  – increases probability of consistency errors
examples of crosscutting code

- logging (tracking program behavior to a file) and tracing (determining what methods are called when)
- profiling (exploring where a program spends its time)
- verification (for test oracles or for deployment)
- policy checking (correcting behavior in case of errors)
- security management
- memory management
logging the AOP way

```java
aspect Logging {
    Logger logger = new Logger();

    when deposit(amount) {
        logger.log("deposit amount : " + amount);
    }

    when withdraw(int) {
        logger.log("withdraw amount : ") + amount);
    }
}
```

that’s not quite how it is written though
logging the AOP way

aspect Logging {
  Logger logger = new Logger();

  before(int amount):
    call(void Power.deposit(int)) && args(amount) {
      logger.log("deposit amount : " + amount);
    }

  before(int amount):
    call(boolean Power.withdraw(int)) && args(amount) {
      logger.log("withdraw amount : " + amount);
    }
}

advice kind
advice parameter
advice body
the real thing
primitive pointcuts

a pointcut is a predicate on join points that:
  – can match or not match any given join point and
  – optionally, can pull out some of the values at that join point

\texttt{call(\text{void}\ Power.deposit(\text{int}))}

matches any join point that is a call of function with this signature
explaining parameters…

• variable is bound by advice declaration
  – pointcut supplies value for variable
  – value is available in advice body

```java
before(int amount) : 
  call(void Power.deposit(int)) && args(amount) {
    logger.log("deposit amount : " + amount);
  }
```
parameter data flow

• value is ‘pulled’
  – right to left across ‘:’ from pointcuts to advice
  – and then to advice body

```java
before(int amount) : 
  call(void Power.deposit(int)) && args(amount) {
    logger.log("deposit amount : " + amount);
  }
```
terminology

- a **join point** is a well-defined point in the program flow
- a **pointcut** is a group of **join points**
- **advice** is code that is executed at a **pointcut**
- **introduction** modifies the members of a class and the relationships between classes
- a **compile time declaration** introduces a compile time warning or error upon detection of certain usage patterns
- an **aspect** is a module for handling crosscutting concerns
  - Aspects are defined in terms of pointcuts, advice, and introduction
  - Aspects are reusable and inheritable
terminology as equations

Joinpoint = well-defined point in the program flow

Pointcut = Joinpoint-set

Advice = Kind × Pointcut × Code

Aspect = Advice-list
aspect Balance {

  pointcut powerChange(Power power):  
      (call(* deposit(..)) || call(* withdraw(..))) 
      && target(power);

  after(Power power): powerChange(power) {
      System.out.println("balance = " + power.balance);
  }
}

"after" advice

named pointcut

pointcut patterns

target pointcut
privileged aspects
can access private fields and methods

privileged aspect Balance {
    pointcut powerChange(Power power) :
        (call(* deposit(..)) || call(* withdraw(..)))
        && target(power);
    after(Power power) : powerChange(power) {
        System.out.println("balance = " + power.balance);
    }
}
args, this and target pointcuts

```java
before(Rover rover, Power power, int amount):
  call(void Power.deposit(int))
  && args(amount) && this(rover) && target(power) {...}
```

```java
class Rover {
  ...
  void execute(...) {
    ...
    power.deposit(500);
    ...
  }
  ...
}
```

```java
class Power {
  ...
  void deposit(int amount) {
    ...
  }
  ...
}
```
target pointcut

target( TypeName | VariableName )

does two things:
  - predicate on join points - any join point at which target object
    is an instance of TypeName or of the same type as VariableName.
    “any join point“ can be:
    • method call join points
    • method & constructor execution join points
    • field get & set join points
    • dynamic initialization join points
  - exposes target if argument is a variable name

\[
\text{target}(\text{Power}) : \\
- \text{matches when target object is of type} \ \text{Power}
\]

\[
\text{target}(\text{power}) : \\
- \text{ditto, since power is of type} \ \text{Power} \\
- \text{in addition it binds the target object to power}
\]
parameter data flow again

- value is ‘pulled’
  - right to left across ‘:’ from pointcuts to user-defined pointcuts
  - from pointcuts to advice
  - and then to advice body

```java
pointcut powerChange(Power power) :
   (call(* deposit(..)) || call(* withdraw(..)))
   && target(power);

after(Power power) : powerChange(power) {
   System.out.println("balance = " + power.balance);
}
```
contract checking

• pre-conditions
  – check whether parameter is valid

• post-conditions
  – check whether values were set

• policy enforcement
  – check, and correct if check fails

• invariants
using before advice

pre-conditions

aspect WithDrawPreCond {
    final int MIN_BALANCE = 50;

    before(Power power, int amount) :
        call(boolean Power.withdraw(int)) &&
        target(power) && args(amount)
        {
            assert power.balance - amount > MIN_BALANCE :
                "withdrawal too big: " + amount;
        }
}
using before & after advice

post-conditions

```java
public aspect WithDrawPostCond {
    int old_balance;

    before(Power power) :
        call(boolean Power.withdraw(int)) && target(power)
    {
        old_balance = power.balance;
    }

    after(Power power, int amount) returning(boolean changed) :
        call(boolean Power.withdraw(int)) &&
        target(power) && args(amount)
    {
        assert changed == (old_balance - amount) >= 0 &&
        power.balance ==
        (changed ? old_balance-amount : old_balance);
    }
}
```
post-conditions

aspect WithdrawPostCondAround {
    int old_balance;

    boolean around(Power power, int amount):
        call(boolean Power.withdraw(int)) &&
        target(power) && args(amount)
    {
        old_balance = power.balance;
        boolean changed = proceed(power, amount);
        assert changed == (old_balance - amount) >= 0;
        assert power.balance ==
            (changed ? old_balance - amount : old_balance);
        return changed;
    }
}
the proceed “method”

for each around advice with the signature:

\[ T \text{ around}(T_1 \ arg_1, \ T_2 \ arg_2, \ ...) \]

there is a special method with the signature:

\[ T \ \text{proceed}(T_1, \ T_2, \ ...) \]

Means:

“run what would have run if this around advice had not been defined”
policy enforcement

aspect WithdrawCorrect {
    final int MIN_BALANCE = 50;

    boolean around(Power power, int amount) {
        call(boolean Power.withdraw(int)) &&
        target(power) && args(amount)
    
    {
        if (power.balance - amount >= MIN_BALANCE)
            return proceed(power, amount);
        else {
            System.out.println("withdrawal rejected");
            return false;
        }
    }
}
}

around advice

proceed statement
aspect Invariant {
    boolean invariant(int balance) {
        return balance >= 500;
    }

    pointcut write(int balance):
        set(int Power.balance) && args(balance);

    before(int balance): write(balance) {
        //every update
        if (!invariant(balance))
            System.out.println("invariant violated");
    }

    after(Power power):
        execution(* Power.*(..)) && target(power)
        {
            if (!invariant(power.balance))
                System.out.println("invariant violated");
        } // at method bounderies
examples of patterns

Type names:
Command
*Command
java.*.Date
Java.*
Javax.*Model+

Combined Types:
!Vector
Vector || HashTable
java.util.RandomAccess+
&& java.util.List+

Method Signatures:
public void Power.set*(*)
boolean Power.withdraw(int)
bo* Po*.wi*w(i*)
!static * *.*(..)
rover..command.Command+.check(int,..)
reflexive information available at all joinpoints

• **thisJoinPoint**
  – getArgs() : Object[]
  – getTarget() : Object
  – getThis() : Object
  – getStaticPart() : JoinPointStaticPart

• **thisJoinPointStaticPart**
  – getKind() : String
  – getSignature() : Signature
  – getSourceLocation() : SourceLocation
logging exceptions using thisJoinPoint

```java
aspect LogExceptions {
    Logger logger = new Logger();

    after() throwing (Error e): call(* *(..)) {
        logger.log("exception thrown " + thisJoinPoint + ":" + e);
    }
}
```

logged information in the case of an assertion error in call of withdraw:

```java
... exception thrown call(boolean core.Power.withdraw(int)):java.lang.AssertionError ...
```
checking object creation

```java
class CmdFactory {
    static Command mkTurnCommand(int budget, int degrees) {
        return new TurnCommand(budget, degrees);
    }
    ...
}
```

want to ensure that any creation of commands goes through the factory methods `mk`...

```java
aspect FactoryCheck {
    pointcut illegalNewCommand():
        call(Command+.new(..)) &&
        !withincode(* CmdFactory.mk*(..));

    before(): illegalNewCommand() {
        throw new Error("Use factory method instead.");
    }
}
```
checking object creation

```java
class CmdFactory {
    static Command mkTurnCommand(int budget, int degrees) {
        return new TurnCommand(budget, degrees);
    }
}
```

```
aspect FactoryCheckStatic {
    pointcut illegalNewCommand():
        call(Command+.new(..)) &&
        !withincode(* CmdFactory.mk*(..));

    declare error : illegalNewCommand() :
        "Use factory method instead."
}
```

want to ensure that any creation of commands goes through the factory methods mk...

must be a “static pointcut”

static check
inter-type declarations

- like member declarations

```c
int counter = 0;
void count() {counter++;}
```
inter-type declarations

• like member declarations
  but with a type prefix

```cpp
int Command::counter = 0;
void Command::count() {counter++;}
```
inserting fields and methods

verify that a command is executed no more than once!

requires a counter per Command object.

field and method inserted in Command object but accessible only to aspect.

aspect ExecuteOnlyOnce {
  private int Command.counter = 0;
  private void Command.count() {counter++;}

  before(Command cmd) :
    call(void Command+.execute()) && target(cmd)
    {
      assert cmd.counter == 0 : "command executed again";
      cmd.count();
    }
}
same property

verify that a command is executed no more than once!

eliminating: private, +, count method, assert message

```java
aspect ExecuteOnlyOnce {
    int Command.counter = 0;

    before(Command cmd) :
        call(void Command.execute()) && target(cmd)
    {
        assert cmd.counter++ == 0;
    }
}
```

it does not get much shorter than this
aspect association

• instances of aspects:
  – one per virtual machine (the default)
  – one per object (perthis, pertarget)
  – one per control-flow (percflow, percflowbelow)
aspect association

- **perthis**(pc):
  - when a pointcut satisfying pc is reached, and this(x) holds, and x does not already have an associated aspect instance of this type, a new instance is created for x (to track x)

- **pertarget**(pc):
  - similar, except we use target(x)

- **percflow**(pc):
  - when a pointcut satisfying pc is reached, a new instance is created, which lasts as long as the control flow under this pc does
same property

verify that a command is executed no more than once!

this time using object association: one aspect per Command target of the execute command.

```java
aspect ExecuteOnlyOnce2 {pertarget(execute()){
    int counter = 0;

    pointcut execute() : call(void Command.execute());

    before() : execute() {
        assert counter++ == 0;
    }
}}
```
tracing

a clear crosscutting structure

all modules of the system use the trace facility in a consistent way: entering the methods and exiting the methods
tracing: object vs. aspect

- using an object captures tracing support, but does not capture its consistent usage by other objects

- using an aspect captures the consistent usage of the tracing support by the objects
Tracing aspect

```java
aspect Tracing {
  private int callDepth = -1;

  pointcut tracePoint() : !within(Tracing);

  before() : tracePoint() {
    callDepth++; print("Before", thisJoinPoint);
  }

  after() : tracePoint() {
    print("After", thisJoinPoint); callDepth--; }

  private void print(String prefix, Object message) {
    for(int i = 0, spaces = callDepth * 2; i < spaces; i++)
      System.out.print(" ");
    System.out.println(prefix + ": " + message);
  }
}
```

all pointcuts, except within Tracing aspect
output
inheritance

• what if we want to trace specific events? do we edit the Tracing aspect? no, we can define the pointcut as abstract
• a pointcut can be defined as abstract without a “right-hand” side:
  
  abstract pointcut something(T x);

• advices can be defined on the abstract pointcut
• specialization of aspect can later define the pointcut
• this resembles parameterization with poincuts
• similar to the way methods can be defined abstract and later defined in sub-classes
abstract Tracing aspect

abstract aspect Tracing {
private int callDepth = -1;

abstract pointcut tracePoint();

before() : tracePoint() {
    callDepth++; print("Before", thisJoinPoint);
}

after() : tracePoint() {
    print("After", thisJoinPoint); callDepth--;
}

private void print(String prefix, Object message) {
    for(int i = 0, spaces = callDepth * 2; i < spaces; i++)
        System.out.print(" ");
    System.out.println(prefix + ": " + message);
}
}
concrete tracing aspect

```
aspect ConcreteTracing extends AbstractTracing {
  pointcut tracePoint() :
    call(* Power.*(..)) || withincode(* Command+.*(..));
}
```

We just define the pointcut.

It’s a bit like function application:

```
aspect ConcreteTracing =
  AbstractTracing(
    call(* Power.*(..)) || withincode(* Command+.*(..))
  )
```

Not AspectJ syntax.
control flow pointcuts

cflow(\textit{Pointcut})
all join points in the dynamic control flow of any
join point picked out by \textit{Pointcut}

cflowbelow(\textit{Pointcut})
all join points in the dynamic control flow \textit{below}
any join point picked out by \textit{Pointcut}

top pointcut not included
example

context-passing aspects

introduce check where we need to know the caller
context-passing aspects

introduce check where we need to know the caller
context-passing aspects

```java
pointcut invocations(Caller c):
  this(c) && call(void Service.doService(String));
```
context-passing aspects

pointcut invocations(Caller c):
    this(c) && call(void Service.doService(String));

pointcut workPoints(Worker w):
    target(w) && call(void Worker.doTask(Task));
context-passing aspects

pointcut invocations(Caller c):
    this(c) && call(void Service.doService(String));

pointcut workPoints(Worker w):
    target(w) && call(void Worker.doTask(Task));

pointcut perCallerWork(Caller c, Worker w):
    cflow(invocations(c)) && workPoints(w);
abstract aspect CapabilityChecking {

  pointcut invocations(Caller c):
      this(c) && call(void Service.doService(String));

  pointcut workPoints(Worker w):
      target(w) && call(void Worker.doTask(Task));

  pointcut perCallerWork(Caller c, Worker w):
      cflow(invocations(c)) && workPoints(w);

  before (Caller c, Worker w): perCallerWork(c, w) {
      w.checkCapabilities(c);
  }
}
advice precedence

what happens if two pieces of advice apply to the same join point?

```java
aspect Policy {
    pointcut scope() : !cflow(adviceexecution());

    before(): call(* .*(..)) && scope() {
        if (!isAllowed(thisJoinPoint))
            error("invalid ");
    }

    declare precedence: Policy, *;
}
```

order undefined unless:
- in same aspect,
- in sub-aspect, or
- using declare precedence...

```java
aspect LogIt {
    pointcut scope() : !cflow(adviceexecution());

    before(): call(* .*(..)) && scope() {
        System.out.println("Entering " + thisJoinPoint);
    }
}
```
advice precedence rules

assume that aspect L has lower priority than aspect H (L < H) and consider a particular joinpoint

• H executes its before advice before L’s before advice
• H executes its after advice after L’s after advice
• H’s around advice encloses L’s around advice
beginner mistake
not controlling circularity of advice

pointcuts sometimes match more than expected

```java
aspect A {
    before(): call(String toString()) {
        System.err.println(thisJoinPoint);
    }
}
```

use within, cflow, adviceexecution() to control

```java
aspect A {
    before(): call(String toString()) && !within(A) {
        System.err.println(thisJoinPoint);
    }
}
```
summary

pointcuts

primitive
  call
  execution
  handler
  get set
  initialization
  this target args
  within within code
  cflow cflow below

user-defined
  pointcut

advice
  before
  after
  around

inter-type decls
  Type.field
  Type.method()

declare
  error
  parents
  precedence

reflection
  thisJoinPoint
  thisJoinPointStaticPart
class Rover {
    Power power = new Power();
    boolean error = false;

    void execute(Command[] plan) {
        power.deposit(500);
        for (Command cmd : plan) {
            if (power.withdraw(cmd.getBudget())) {
                try {
                    cmd.execute();
                } catch (ExecException e) {
                    e.printStackTrace();
                }
            } else {
                error = true;
                System.out.println("terminating");
                break;
            }
        }
    }
}
an aspect that gets “around”

```java
public aspect Monitor {
    static boolean tracingOn = true;

    pointcut scope() : if(tracingOn) && !cflow(adviceexecution());

    pointcut handlethrow(ExecException e) : handler(ExecException) && args(e);
    before(ExecException e) : handlethrow(e) && scope() {
        print("*** bad luck: " + e);
    }

    after() returning (Power power) : call(Power.new()) && scope() {
        print("power object created " + power);
    }

    before(int amount) : call(void deposit(int)) && args(amount) && scope() {
        print("depositing: " + amount);
    }

    after(int amount) returning (boolean success):
        call(boolean Power.withdraw(int)) && args(amount) && scope() {
        print("withdrawing " + amount + ":" + success);
    }
}
```
```java
void around(Command[] plan) {
    execution(void Rover.execute(Command[])) && args(plan) && scope() {
        if (!validatePlan(plan))
            proceed(correctPlan(plan));
        else
            proceed(plan);
    }
}

after() returning(Power power):
    get(Power Rover.power) && within(Rover) && scope() {
        print("reading power " + power);
    }

before(boolean value):
    set(boolean Rover.error) && args(value) && if(value)
        && withincode(* Rover.execute(..)) && scope() {
        print("error flag being set to " + value);
    }

before(): call(* .*(..)) && cflow(call(* Power.withdraw(..))) && scope() {
    print("function call " + thisJoinPointStaticPart.getSignature());
}

before(Rover rover, Command command):
    call(* Command.execute()) &&
    this(rover) && target(command) && scope() {
        print("Rover " + rover + " executing command " + command);
    }
```
abstract syntax for AspectJ

• contains most elements of language
• look at quick guide
• look at examples
AspectDecl ::= 
  [ privileged ] [ Modifiers ] aspect Id 
  [ extends Type ] [ implements TypeList ] 
  [ PerClause ] 
  { BodyDecl* } 

PerClause ::= 
  pertarget ( Pointcut ) | perthis ( Pointcut ) 
  | percflow ( Pointcut ) | percflowbelow ( Pointcut ) | issingleton () 

BodyDecl ::= 
  JavaBodyDecl 
  | IntertypeDecl 
  | PointcutDecl 
  | AdviceDecl
InterTypeDecl ::=  
    [ Modifiers ] Type Type . Id ( Formals ) [ throws TypeList ] { Body }  
| [ Modifiers ] Type . new ( Formals ) [ throws TypeList ] { Body }  
| [ Modifiers ] Type Type . Id [ = Expression ] ;  
| declare warning : Pointcut : String ;  
| declare error : Pointcut : String ;  
| declare precedence : TypePatList ;
PointcutDecl ::= 
  abstract [Modifiers] pointcut Id ( Formals ) ; 
  | [Modifiers] pointcut Id ( Formals ) : Pointcut ;

AdviceDecl ::= 
  AdviceSpec [ throws TypeList ] : Pointcut { Body }

AdviceSpec ::= 
  before ( Formals )
  after ( Formals )
  after ( Formals ) returning [ ( Formal ) ]
  after ( Formals ) throwing [ ( Formal ) ]
  Type around ( Formals )

AspectJ syntax
pointcut and advice declarations
Pointcut ::=  

  call(MethodPat) | call(ConstructorPat) 
  | execution(MethodPat) | execution(ConstructorPat) 
  | initialization(ConstructorPat) | preinitialization(ConstructorPat) 
  | staticinitialization(TypePat) 
  | get(FieldPat) | set(FieldPat) 
  | handler(TypePat) 
  | adviceexecution() 
  | within(TypePat) | withincode(MethodPat) | withincode(ConstructorPat) 
  | cflow(Pointcut) | cflowbelow(Pointcut) 
  | if(Expression) 
  | this(Type | Var) | target(Type | Var) | args(Type | Var , ...)
MethodPat ::=  
  [ModifiersPat] TypePat [TypePat . ] IdPat ( TypePat | .. , … )  
  [ throws ThroesPat ]

ConstructorPat ::=  
  [ModifiersPat ] [TypePat . ] new ( TypePat | .. , … )  
  [ throws ThroesPat ]

FieldPat ::= [ModifiersPat] TypePat [TypePat . ] IdPat

TypePat ::=  
  IdPat [ + ] [ [] … ]  
  | ! TypePat  
  | TypePat && TypePat  
  | TypePat || TypePat  
  | ( TypePat )  
  | IdPat ::=  
  | Java id with `*`'s mixed in
Expression ::= 
    thisJoinPoint 
  | thisJoinPointStaticPart 
  | thisEnclosingJoinPointStaticPart

StatementExpression ::= 
    proceed ( Arguments )

AspectJ syntax
special expressions and statements
end