introduction to AspectJ

CS 119

here viewed as a:
- program instrumentation and
- monitoring framework
why AspectJ?

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

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

• this lesson : introducing the language
• next lesson : monitoring with AspectJ
resources

- [http://www.eclipse.org/aspectj](http://www.eclipse.org/aspectj)

optional reading
AspectJ

- AspectJ, launched 1998 at Xerox PARC
- 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, very mature
- AspectJ works with Eclipse, and other IDEs
- outputs .class files compatible with any JVM
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
code scattering: one concern many modules

element: logging
two central problems
AOP tries to solve

code trangling:
one module
many concerns

code scattering:
one concern
many modules

example:
logging
examples of crosscutting code

- logging (tracking program behavior)
- verification (checking program behavior)
- policy enforcement (correcting behavior)
- security management (preventing attacks)
- profiling (exploring where a program spends its time)
- memory management
- visualization of program executions
- ...
the problem

• the flow of the core logic gets obscured, harder to follow, the core logic is tangled with the new code.

• the new code code gets scattered throughout the code base
  – lots of typing
  – big picture (in one place) is missing
  – difficult to find what is new code and how it works
  – difficult to change new code
  – increases probability of consistency errors
very simplified view of AOP

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

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

informal notation

weaver

program

aspect

while(more())
{
  ...
  check(msg);
  send(msg);
  ...
}
that’s it

except for notation, all the details, usage, …
basic mechanisms

- join points
  - points in a Java program

- three main 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
  - aspect
    - a modular unit of crosscutting behavior
      - normal Java declarations
      - pointcut definitions
      - advice

- inter-type declarations
  add fields, methods to classes
terminology as equations

**Program:**
Joinpoint = well-defined point in the program

**AspectJ:**
Pointcut = Joinpoint-set

Advice = Kind × Pointcut × Code
where Kind = \{before, after, around\}

Aspect = Advice-list
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;
    }
}
logging class

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

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

logging the traditional way
logging the AOP way

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

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

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

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

```java
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);
        }
}
```

the AspectJ way
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

Example:

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

matches any join point that is a:

\texttt{call} of a method with this signature
explaining parameters... of advice

- 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);
    }
```
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);
  }
}

suppose power.balance is a private variable. Then the aspect must be privileged.
**args, this and target pointcuts**

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

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

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

\texttt{target( TypeName | VariableName )}

does two things:
- predicate on join points - any join point at which target object
  is an instance of \texttt{TypeName} or of the same type as \texttt{VariableName}.
  “any join point “ can be:
  \begin{itemize}
  \item method call join points
  \item field get & set join points
  \item ...
  \end{itemize}
- exposes target if argument is a variable name

\texttt{target(Power) :}
  - matches when target object is of type \texttt{Power}

\texttt{target(power) :}
  - ditto, since power is of type \texttt{Power}
  - 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

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 that parameter is valid
- **post-conditions**
  - check that result is correct
- **policy enforcement**
  - check, and correct if check fails
- **invariants**
  - check that some condition on the state "always" holds
pre-condition

```java
boolean withdraw(int amount) {
    pre balance - amount > 50;
    ...
    // implementation:
    ...
}
```
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;
    }
}
boolean withdraw(int amount) {
    pre balance - amount > 50;
    post result == (old(balance) - amount) >= 0 &&
    balance ==
    (result ? old(balance) - amount : old(balance))

    ... // implementation:
    ...
}

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);
    }
}
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:

\[ T \text{ around}(T_1 \text{ arg1}, T_2 \text{ arg2}, \ldots) \]

there is a special method with the signature:

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

calling this method means:

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

boolean withdraw(int amount) {
    pre balance - amount > 50 {
        System.out.println("withdrawal rejected");
        return false;
    }
    ...
    // implementation:
    ...
}
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;
        }
    }
}
invariant checking

class Power {
    int balance;

    invariant balance >= 500;

    void deposit(int amount) {
        ...
    }

    boolean withdraw(int amount) {
        ...
    }
}
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 boundaries
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
... thisJoinPoint
```
checking object creation

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

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

causes check to be performed at compile time
inter-type declarations

• inside an aspect:
  adding declarations to a class \( C \)

```java
aspect A {
  int counter = 0;
  void count() {counter++;}
}

class C {
  ...
  ...
}
```
inter-type declarations

• one must indicate what class:

```java
aspect A {
    int C.counter = 0;
    void C.count() {counter++;}
    ...
}
```

```java
class C {
    ...
    ...
}
```
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 (\texttt{perthis}, \texttt{pertarget})
  - one per control-flow (\texttt{percflow}, \texttt{percflowbelow})

```
aspect <AspectName> <association>(<pointcut>){
  pointcut ... : ...;
  ...
}

<association> ::= 
  perthis | pertarget | 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;
    }
}
```
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);
    }
}
Before: execution(core.ExecuteOnlyOnce())
    After: execution(core.ExecuteOnlyOnce())
    After: initialization(core.ExecuteOnlyOnce())
    After: static initialization(core.ExecuteOnlyOnce.<clinit>)
Before: set(int core.Command.counter)
    After: set(int core.Command.counter)
Before: execution(core.Command(int))
    Before: set(int core.Command.budget)
    After: set(int core.Command.budget)
    After: execution(core.Command(int))
After: initialization(core.Command(int))
Before: initialization(core.PictureCommand(int))
    Before: execution(core.PictureCommand(int))
    After: execution(core.PictureCommand(int))
    After: initialization(core.PictureCommand(int))
    After: call(core.PictureCommand(int))
Before: call(core.TurnCommand(int, int))
    Before: static initialization(core.TurnCommand.<clinit>)
    After: static initialization(core.TurnCommand.<clinit>)
    Before: pre initialization(core.TurnCommand(int, int))
    After: pre initialization(core.TurnCommand(int, int))
    Before: pre initialization(core.Command(int))
After: initialization(core.Command(int))
    Before: set(int core.Command.counter)
    After: set(int core.Command.counter)
Before: execution(core.Command(int))
    Before: set(int core.Command.budget)
    After: set(int core.Command.budget)
    After: execution(core.Command(int))
After: initialization(core.Command(int))
Before: initialization(core.TurnCommand(int, int))
    Before: execution(core.TurnCommand(int, int))
    Before: set(int core.TurnCommand.degrees)
    After: set(int core.TurnCommand.degrees)
abstract pointcuts

• 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 pointcuts
• similar to the way methods can be defined abstract and later defined in sub-classes
abstract Tracing aspect

abstract aspect AbstractTracing {
private int callDepth = -1;

abstract pointcut tracePoint();

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

after() : tracePoint() {
    print("After", thisJoinPoint); callDepth--; System.out.println;
}

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

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

It's a bit like function application:

\[
\text{aspect ConcreteTracing = AbstractTracing(}
    \text{call(* Power.*(..)) || withincode(* Command+.*(..))}
\]

not AspectJ syntax

we just define the pointcut
control flow pointcuts

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

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

\textit{top pointcut not included}
example

introduce check where we need to know the caller
example

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

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

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

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

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

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

pointcut calledWork(Caller c, Worker w):
   cflow(invocation(c)) && workPoint(w);
context-passing aspects

abstract aspect CapabilityChecking {

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

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

  pointcut calledWork(Caller c, Worker w):
    cflow(invocation(c)) && workPoint(w);

  before (Caller c, Worker w): calledWork(c, w) {
    verifyCalledWork(c, w);
  }
}
advice precedence

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

aspect Policy {
    pointcut scope() : !cflow(adviceexecution());
    ...
    before(): call(* ..(.)) && scope() {
        if (!isAllowed(thisJoinPoint))
            error("invalid ");
    }
    declare precedence: Policy, *;
}

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

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

use within, cflow, adviceexecution() to control

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

pointcuts

primitive
call
class execution
class handler
class get set
class initialization
this target args
within withincode
cflow cflowbelow

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);
    }
}
```
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 ) 
  | percfow ( Pointcut ) | percfowbelow ( 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 )
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 , ...)

AspectJ syntax
pointcuts
MethodPat ::= 
[ModifiersPat] TypePat [TypePat . ] IdPat ( TypePat | .., ... )
[ throws ThROWSPat ]

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

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 )
end