CS 6156

Program Instrumentation
(with ASM and AspectJ)

Owolabi Legunsen

Slides based in part on lectures by Klaus Havelund at CalTech
Some logistics

• Reading 3 to be assigned
  • due 11:59pm AoE 3/2

• Feedback on your project proposal was provided on CMSX on 2/14
  • Work on your phase 1, due 3/8
What is instrumentation?

• “By program instrumentation here we mean the process of inserting additional statements into a program for information gathering purposes.”

• “Program instrumentation is a way of learning about the effect individual tests have on a program.”

1J.C. Huang, Detection of Data Flow Anomaly Through Program Instrumentation, TSE 1979
2E. F. Miller, Program Testing, IEEE Computer 1978
Instrumentation in practice

• How do debuggers know what code to step through?

• How does your code coverage tool know what statements, blocks, methods, etc., are covered?

• Did you ever write “printf” statements to know what (parts of) your code does?
Recall: what you’ll learn in CS 6156

• How to instrument code to obtain runtime events?
• Compile-time vs. runtime instrumentation
• Problems and challenges of instrumentation
Why instrumentation in CS6156

• At 57% of student projects will need perform instrumentation

• No instrumentation, no RV
Some instrumentation frameworks

- ASM
- Javassist
- BCEL
- AspectJ, AspectC, AspectWerkz, etc.
- JVMTI
- JMX
- Spring AOP
- ...
Demo

• Maven

• Visitor Pattern

• ASM

• AspectJ
Why AspectJ?

• RV requires instrumentation and specification

• AspectJ can provide both elements\(^3\)

• AspectJ is probably the most popular aspect-oriented programming (AOP) framework

\(^3\)Bodden et al., Collaborative Runtime Verification with Tracematches, RV 2007
JavaMOP syntax extends AspectJ

```
// BNF below is extended with {p} for zero or more and [p] for zero or one repetitions of p

<Specification> ::= {<Modifier>} <Id> <Parameters> "{" 
{<Declaration>}
{<Event>}
{<Property>
{<Property Handler>}
}
"}"

<Modifier> ::= "unsynchronized" | "decentralized" | "perthread" | "suffix"

<Event> ::= "event" <Id> <Event Definition> <Action>

<Property> ::= <Logic Name> ":" <Logic Syntax>

<Property Handler> ::= "@" <Logic State> <Action>

<Event Definition> ::= <Advice Specification> "":"<Extended Pointcut"

>Action ::= "[" <Statements> "]"

<Extended Pointcut> ::= <Pointcut>
| <Extended Pointcut> "&&" <Extended Pointcut>
| "thread" (" <Id> ")
| "condition" (" <Boolean Expression> ")"

<Parameters> ::= "(" [ <Parameter> { "," <Parameter> } ] ")"

<Parameter> ::= <Type Pattern> <Id>

<Type Pattern> ::= - AspectJ Type Pattern

<Id> ::= <!- Java Identifier

<Declaration> ::= <!- Java variable declaration

<Advice Specification> ::= <!- AspectJ AdviceSpec

<Pointcut> ::= <!- AspectJ Pointcut

<Statements> ::= <!- Java statements

<Boolean Expression> ::= <!- Java boolean expressions
```
AspectJ implements AOP

• AOP modularizes programs differently than OOP

• Separates out cross-cutting concerns: code for one *aspect* of the program is collected in one place

• We will not delve into AOP as a paradigm
  • But we briefly explain the more general purpose of AOP
  • Focus: enough AspectJ to understand/write JavaMOP specs
Good modularity

- XML parsing in org.apache.tomcat circa 2009(?)
  - red shows relevant lines of code
  - nicely fits in one box (object)
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 problems AOP tries to solve

**code tangling:**
- one module
- many concerns

**code scattering:**
- one concern
- many modules

**example:** logging
Discuss: what are the effects of tangling and scattering?
The effects of the two problems

• Core logic becomes harder to comprehend when it is tangled with other code

• Scattering similar logic in the code base results in
  • lots of typing, difficult to change code
  • missing the big picture (in one place)
  • increased probability of consistency errors
How AOP solves the two problems

code tangling:
one module
many concerns

code scattering:
one concern
many modules

example: logging
Cross-cutting concerns are common

- logging (tracking program behavior)
- verification (checking program behavior)
- policy enforcement (correcting behavior)
- security management (preventing attacks)
- profiling (exploring where programs spend time)
- memory management
- visualization of program executions
- ...
A very simplified view of AOP

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

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

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)
AspectJ terminology

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

**Pointcut** = **Joinpoint**-set

**Advice** = **Kind** $\times$ **Pointcut** $\times$ **Code**

where **Kind** = \{before, after, around\}

**Aspect** = **Advice**-list
Example code

class Account {
    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;
    }
}
Logger class

class Logger {
    private PrintStream stream;

    Logger() {
        ... create stream
    }

    void log(String message) {
        stream.println(message);
    }
}
Logging without AOP

class Account {
    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 with AOP

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

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

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

Logging code is in exactly one place
Logging in AspectJ

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

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

    before(int amount) :
        call(boolean Account.withdraw(int)) && args(amount) {
            logger.log("withdraw amount : " + amount);
        }
}
```
Primitive pointcuts

• A pointcut is a predicate on join points that:
  • can match or not match any given join point
  • can extract some values at matching join points

Example:

call(void Account.deposit(int))

matches any join point that is a call of a method with this signature
Explaining advice parameters

- Variables are bound by advice declaration
- Pointcuts supply values for variable
- Values are available in the advice body

```java
before(int amount) :
    call(void Account.deposit(int)) && args(amount) {
        logger.log("deposit amount : " + amount);
    }
```
Advice 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 Account.deposit(int)) && args(amount) {
        logger.log("deposit amount : " + amount);
    }
```
Pointcut naming and patterns

```java
aspect Balance {

  pointcut accountChange(Account account) :
    (call(* deposit(..)) || call(* withdraw(..)))
  && target(account);

  after(Account account) : accountChange(account) {
    System.out.println("balance = " + account.balance);
  }
}
```

"after" advice

named pointcut

pointcut patterns

target pointcut
Privileged aspects

• Aspects that can access private fields and methods

privileged aspect Balance {

    pointcut accountChange(Account account) :
        (call(* deposit(..)) || call(* withdraw(..)))
        && target(account);

    after(Account account) : accountChange(account) {
        System.out.println("balance = " + account.balance);
    }
}

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

```java
class Client {
    ... 
    void execute(...) {
        ... 
        account.deposit(500);
        ... 
    }
    ... 
}
```

```java
class Account {
    ... 
    void deposit(int amount) {
        ... 
    }
    ... 
}
```

```java
Object C

before(Client client, Account account, int amount) :
    call(void Account.deposit(int))
    && args(amount) && this(client) && target(account) {...}
```

Object A
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 same type as VariableName.
- exposes target if argument is a variable name

target(Account) :
- matches when target object is of type Account

Account is a type

target(account) :
- matches too, since account is of type Account
- in addition, it binds the target object to account

account is a variable
Parameter data flow again

• Value is ‘pulled’
  • right to left from pointcuts to user-defined pointcuts
  • from pointcuts to advice
  • and then to advice body

```java
pointcut accountChange(Account account) :
    (call(* deposit(..)) || call(* withdraw(..))) && target(account);

after(Account account) : accountChange(account) {
    System.out.println("balance = " + account.balance);
}
```
The proceed “method”

• For each around advice with the signature:

  \[ T \text{ around}(T1 \text{ arg}1, T2 \text{ arg}2, \ldots) \]

• There is a special method with the signature:

  \[ T \text{ proceed}(T1, T2, \ldots) \]

• Calling “proceed” means:

  “run what would have been run if this around advice had not been defined”
Reflexive information available at all joinpoints

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

- **thisJoinPointStaticPart**
  - getKind() : String
  - getSignature() : Signature
  - getSourceLocation() : SourceLocation

Fun activity: implement a code coverage tool in AspectJ
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 Account.set*(*)
- boolean Account.withdraw(int)
- bo* Po*.wi*w(i*)
- !static * *.*(..)
- rover..command.Command+.check(int,..)
Challenges in instrumentation

• Cost: instrumentation can slow programs down

• Heisenbugs\(^4\): slowing program execution can introduce hard-to-debug timing-related bugs

• Can produce hard to read (binary) code

• Instrumentation tools can conflict

\(^4\)Recall “Heisenberg’s” uncertainty principle in physics
Food for thought (take home)

Is AspectJ/AOP the best way to instrument code for RV?
AspectJ Resources

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

---

_AspectJ Quick Reference_

**Aspects**  
_at top-level (or static in types)_

aspect A { ... }  
defines the aspect A

privileged aspect A { ... }  
A can access private fields and methods

aspect A extends B implements I, J { ... }  
B is a class or abstract aspect, I and J are interfaces

aspect A perclawo( call(void Foo.m()) ) { ... }  
an instance of A is instantiated for every control flow through calls to m()

general form:  
[ privileged ] [ Modifiers ] aspect Id  
[ extends Type ] [ implements TypeList ] [ PerClause ]  
{ Body }  
where PerClause is one of  
pertarget ( Pointcut )
What we covered in this class

• Instrumentation is important in many software engineering tasks, including RV

• We learned the basics of two instrumentation tools

• An introduction to aspect-oriented programming

• Hands-on exposure to AspectJ