Byteman
Simple, Flexible, Dynamic Advice Injection for Java

Andrew Dinn
Principal Software Engineer
Red Hat UK
• Why Byteman?
• Using Byteman
• Byteman Rule Language
• Extending Byteman
• Byteman Extension Packages
• Summary
• Questions
Why Byteman?
What Is Byteman?

• **Byte(code) Man(ipulation) Agent**
  – Java agent (*java.lang.instrument*)
• **Injects Java advice into Java code**
  – like JBoss AOP, AOXXX
• **Developed to support**
  – testing
    • inject faults, engineer/validate test scenarios
  – tracing
    • in live installations and at all stages of testing
  – monitoring & stats
    • in live installations and in later stages of testing
Why Use Advice?

• **Testing**
  – less need for mocks
    • make real code misbehave
  – less need for scaffolding
    • make real code enter abnormal scenarios
  – less need to instrument your app
    • injected code can “access all areas”

• **Tracing/Monitoring**
  – no clunky global on/off switch
    • with consequent changes to footprint and timings
  – no 100% foresight
    • decide what to trace/monitor at run time
Why Byteman?

• Already have JBoss AOP
  – used by many JBoss projects

• Already allows you to inject code for
  – testing
  – tracing
  – monitoring & stats

• The issue is *How* not *Why*
Why Not AOP? (1)

- **Pointcut algebra is very un-Java**
  - developers may understand AOP pointcuts
    - maintainers? support staff? customers?

- **It's too powerful**
  - mostly want simple pointcuts
    - specific, known locations for trace, synch or fault
  - rarely want bulk transforms
    - in whatever code gets thrown at us

- **It's not powerful enough**
  - runtime injection must be efficient
    - complex pointcut algebras endanger this
Why Not AOP? (2)

- Advice model is not “normal” Java
  - interceptor abstraction is powerful but *indirect*
  - “do this here!” is *direct, hands-on*

- Reflective API means more indirection
  - programming with boxing gloves
  - discards type checking
    - in the eyeball as well as in the runtime

- Java code should link direct into target
  - “inject y.print() into X.process(Y y)”
    - clear to any Java programmer who knows X, Y
    - developers, maintainers, support staff, customers
Why Not AOP? (3)

• **Advice deployment**
  – we need it to be rapid and flexible
    • edit, (re)deploy
    • edit, *configure?*, *compile?*, *package?*, (re)deploy

• **Interactive, investigative use**
  – “inject this here”
  – “remove that and re-inject this”

• **Return to status quo ante**
  – “uninject all” should restore normal running
• Using Byteman
Tracing Example

RULE trace inactive transaction at commit
CLASS TransactionIImpl
METHOD commit()
AT ENTRY
BIND status : int = $0.getStatus()
IF status != javax.transaction.Status.STATUS_ACTIVE
DO traceStack("inactive commit " + $this +
               " status=" + status, 15);
ENDRULE

• Event Condition Action Rules
  – simple minimal structure for injected code

• Very Java-oriented
  – in fact it is Java, mostly!
E(B)CA Rules

• **Event**
  – CLASS/INTERFACE, METHOD, AT
    • identifies trigger point(s) in code base
    • package, signature, return type optional

• **Binding**
  – introduces and initializes rule variables

• **Condition**
  – any Java boolean expression

• **Action**
  – any Java expressions
    • may end with THROW or RETURN
Fault Injection Example

RULE simulate exception from Executor
INTERFACE ^java.util.Executor
METHOD execute
AT ENTRY
IF callerEquals("ServiceInstanceImpl.execute", true)
DO traceln(“Throwing exception in execute”);
   THROW new
       java.util.concurrent.RejectedExecutionException();
ENDRULE

• inject through interface into implementors
• inject into overriding implementations
  – AbstractExecutor implements Executor
  – ThreadPoolExecutor extends AbstractExecutor
• THROW/RETURN
  – short-circuits trigger method
  – must not break method contract
Driving Byteman

• From command line
  
  java -javaagent:byteman.jar=script:myRules.btm
  bmjava -l myRules.btm -cp my.jar MyClass myargs

• Into running program
  
  bminstall org.jboss.Main
  bmsubmit myRules.btm
  bmsubmit -u myRules.btm

• Agent transformer
  – injects rules into existing/newly loaded classes

• Agent port listener
  – adds/removes rules to/from current rule set
  – triggers retransform of affected classes
• Byteman Rule Language
Location Clauses

AT ENTRY
AT EXIT
AT/AFTER READ [[package.]class.]field [count]
AT/AFTER WRITE [[package.]class.]field [count]
AT/AFTER READ $localvar [count]
AT/AFTER WRITE $localvar [count]
AT/AFTER CALL [[package.]type.]method [(Types)] [count]
AT THROW [count]
AT LINE number

public check(Symbol sym) throws BadSym, BadType
{
    String s = "";
    if (badSym(sym)) {
        s = munge(sym.name);
        throw new BadSym(s);
    } else if (badType(sym.type)) {
        s = munge(sym.type.name);
        throw new BadType(s);
    }
    ...

Location Clauses

AT ENTRY
AT EXIT
AT/AFTER READ [[package.]class.]field [count]
AT/AFTER WRITE [[package.]class.]field [count]
AT/AFTER READ $localvar [count]
AT/AFTER WRITE $localvar [count]
AT/AFTER CALL [[package.]type.]method [(Types)] [count]
AT THROW [count]
AT LINE number

public check(Symbol sym) throws BadSym, BadType
{
    String s = ""; // AFTER WRITE $s
    if (badSym(sym)) {
        s = munge(sym.name);
        throw new BadSym(s);
    } else if (badType(sym.type)) {
        s = munge(sym.type.name);
        throw new BadType(s);
    }
    ...

Location Clauses

AT ENTRY
AT EXIT
AT/AFTER READ [[package.]class.]field [count]
AT/AFTER WRITE [[package.]class.]field [count]
AT/AFTER READ $localvar [count]
AT/AFTER WRITE $localvar [count]
AT/AFTER CALL [[package.]type.]method [(Types)] [count]
AT THROW [count]
AT LINE number

public check(Symbol sym) throws BadSym, BadType
{
    String s = "";              // AFTER WRITE $s
    if (badSym(sym)) {
        s = munge(sym.name);      // AT WRITE $s 2
        throw new BadSym(s);
    } else if (badType(sym.type)) {
        s = munge(sym.type.name);
        throw new BadType(s);
    }
}
Location Clauses

AT ENTRY
AT EXIT
AT/AFTER READ [[package.]class.]field [count]
AT/AFTER WRITE [[package.]class.]field [count]
AT/AFTER READ $localvar [count]
AT/AFTER WRITE $localvar [count]
AT/AFTER CALL [[package.]type.]method [(Types)] [count]
AT THROW [count]
AT LINE number

```java
public check(Symbol sym) throws BadSym, BadType {
    String s = "";              // AFTER WRITE $s
    if (badSym(sym)) {
        s = munge(sym.name);      // AT WRITE $s 2
        throw new BadSym(s);
    } else if (badType(sym.type)) {
        s = munge(sym.type.name); // AFTER WRITE $s 3
        throw new BadType(s);
    }
}
```
Location Clauses

AT ENTRY
AT EXIT
AT/AFTER READ [[package.]class.]field [count]
AT/AFTER WRITE [[package.]class.]field [count]
AT/AFTER READ $localvar [count]
AT/AFTER WRITE $localvar [count]
AT/AFTER CALL [[package.]type.]method [(Types)] [count]
AT THROW [count]
AT LINE number

public check(Symbol sym) throws BadSym, BadType
{
    String s = "";
    if (badSym(sym)) {
        s = munge(sym.name);      // AT READ name
        throw new BadSym(s);
    } else if (badType(sym.type)) {
        // AT READ name 2
        s = munge(sym.type.name); // AT READ Type.name
        throw new BadType(s);
    }
}
Location Clauses

AT ENTRY
AT EXIT
AT/AFTER READ [[package.]class.]field [count]
AT/AFTER WRITE [[package.]class.]field [count]
AT/AFTER READ $localvar [count]
AT/AFTER WRITE $localvar [count]
AT/AFTER CALL [[package.]type.]method [(Types)] [count]
AT THROW [count]
AT LINE number

public check(Sym sym) throws BadSym, BadType {
    String s = "";
    if (badSym(sym)) {
        // AT CALL munge
        s = munge(sym.name);
        // AT CALL munge(SymName)
        throw new BadSym(s);
    } else if (badType(sym.type)) {
        // AFTER CALL munge 2
        s = munge(sym.type.name);
        // AFTER CALL munge(TypeName)
        throw new BadType(s);
    }
}
Location Clauses

AT ENTRY
AT EXIT
AT/AFTER READ [[package.]class.]field [count]
AT/AFTER WRITE [[package.]class.]field [count]
AT/AFTER READ $localvar [count]
AT/AFTER WRITE $localvar [count]
AT/AFTER CALL [[package.]type.]method [(Types)] [count]
AT THROW [count]
AT LINE number

public check(Sym sym) throws BadSym, BadType
{
    String s = "";
    if (badSym(sym))) {
        s = munge(sym.name);
        throw new BadSym(s); // AT THROW, AT THROW ALL
    } else if (badType(sym.type)) {
        s = munge(sym.type.name);
        throw new BadType(s); // AT THROW 2, AT THROW ALL
    }
}
Rule Expressions

• Parameter, Local and Bind Variables
  – $0, $1, $this, $myArg, $loopvar, status

• Special Variables
  – $*, $# trigger method target/args and count
    • Object[] $*; int $#;
  – $! stacked return val AT EXIT/AFTER CALL
    • type same as trigger method/called method
  – $@ method target/args AT CALL
    • Object[] $@;
  – $^ stacked throwable AT THROW
    • Throwable $^;
Rule Expressions (2)

• All the usual Java operations
  – operators + - */& | , && | | ! , == <, new, =, etc
  – instance/static field access and method calls
  – built-in methods (any call with no target object)
  – no control structures (just IF DO)

• $X = \ldots$ changes trigger method state
  – $1 = "Andrew"$
  – $\text{loopvar} = \text{loopvar} + 1$
  – $! = 3$
Built-in Methods

• **Tracing**
  – traceOpen, traceClose, traceLn, traceStack

• **Shared Rule State**
  – flag, clear, flagged, countDown, incrementCounter

• **Synchronization**
  – waitFor, signalWake, rendezvous, delay

• **Timing**
  – createTimer, getElapsedTime, resetTimer

• **Call Stack Checking**
  – callerEquals, callerMatches

• **Recursive Triggering**
  – disable/enableTriggering
Multi-Rule Example

• Test Scenario
  – XTS Coordinator Service
    • negotiates 2 phase commit with remote Web Service Participants
    • sends PREPARE waits for PREPARED
    • logs participant details
    • sends COMMIT expects COMMITTED
  – crash recovery test
    • kill JVM just after logging then reboot
    • drop COMMITTED messages during recovery
    • exercise coordinator retry and participant resend
Multi-Rule Example 2

- Use Countdown to coordinate actions
  - drop message while counting down

RULE drop committed message
CLASS CoordinatorEngine
METHOD committed(Notification, MAP, ArjunaContext)
AT ENTRY
BIND engine:CoordinatorEngine = $0,
   identifier:String = engine.getId()
IF getCountDown(identifier)
DO RETURN
ENDRULE

- create Countdown during reboot
- decrement at commit attempt
RULE add coordinator engine countdown
CLASS CoordinatorEngine
METHOD <init>(String, boolean, 
    EndpointReference, boolean, State)
AT EXIT
BIND engine:CoordinatorEngine = $0, 
    identifier:String = engine.getId()
IF engine.recovered
DO createCountDown(identifier, 2)
ENDRULE

RULE countdown at commit
CLASS CoordinatorEngine
METHOD commit
AFTER WRITE status
BIND engine:CoordinatorEngine = $0 
    identifier:String = engine.getId()
IF engine.recovered && countDown(identifier)
DO traceln("countdown completed for " + identifier)
ENDRULE
• Extending Byteman
Helper Classes

- Built-ins are public methods of a POJO
  - org.jboss.byteman.rule.Helper

- You can use any POJO you like

  class DBHelper {
      public void dbtrace(String msg, DBRecord rec) {
          ...  
      }

  }

  RULE use my own trace method
  CLASS org.my.db.DBManager
  METHOD update(DBRecord)
  AT CALL setName(String)
  HELPER org.my.db.bmutil.DBHelper
  IF $@[1] == "Andrew"
  DO dbtrace("found interesting update", $1)
  ENDRULE
Helper Classes (2)

- Built-ins are public methods of a POJO
  - org.jboss.byteman.rule.Helper
- You can use any POJO you like

```java
class DBHelper extends Helper {
    public void dbtrace(String msg, DBRecord rec) {
        ...
    }
}
```

RULE use my own trace method
CLASS org.my.db.DBManager
METHOD update(DBRecord)
AT CALL setName(String)
HELPER org.my.db.bmutil.DBHelper
IF flagged($1) && $@[1] == "Andrew"
DO dbtrace("found interesting update", $1)
ENDRULE
Helper Lifecycles

- Run code as rules are loaded/unloaded
  - static methods of helper class
    - first/last rule using helper
      - activated()/deactivated()
    - per rule
      - installed(String),uninstalled(String)

- Allows for setup/teardown

```java
public class JMXHelper extends Helper {
    static void activated() {
        bgThread = new JMXHelperThread();
        bgThread.start();
    }
    static void deactivated() { bgThread.end(); }
    ...
}
```
Monitoring with JMXHelper

• Script injects counting rules into JVM

RULE track thread entry
CLASS ^Thread
METHOD run()
HELPER org.jboss.byteman.sample.helper.JMXHelper
IF TRUE
DO incrementCounter("thread runs")
ENDRULE

• Helper thread samples & presents stats
  – present via standard Java dynamic MXBean
  – configured using injected code
    • sample period
    • which counters/what format
Monitoring with JMXHelper (2)

• Configuring the helper thread

RULE return key info
CLASS JMXHelperThread
METHOD keyInfo()
BIND KeyInfo keyInfo =
    new KeyInfo("JVM stats in an MX bean")
IF TRUE
DO keyInfo.addKey("thread runs",
    KeyInfo.KEY_TYPE_CUMULATIVE,
    "Thread.run() - total calls");
keyInfo.addKey("thread runs",
    KeyInfo.KEY_TYPE_RATE,
    "Thread.run() - calls per second");
    ...
    RETURN keyInfo;
ENDRULE
• Byteman Extension Packages
Byteman Runtime API

```
org.jboss.byteman.agent.install.Install
install(String pid, ...) // "0" == this JVM
VMInfo[] availableVMs()
```

```
org.jboss.byteman.agent.submit.Submit
addRulesFromFile(List<String> files)
addScripts(List<ScriptText> scripts)
```

- Enables automated advice injection
  - e.g. based on annotations (BMUnit)
  - e.g. based on reflection (DTest)
BMUnit

• Integrates Byteman into JUnit/TestNG
• Annotate test classes and test methods
  – @BMScript identifies script to load
  – @BMRule provides rule text in annotation
  – test class annotation
    • load before running test methods, unload after
  – test method annotation
    • load before running test method, unload after
• Test package extension point
  – @RunWith(BMUNitRunner.class)
  – class DBTests1 extends BMNGRunner
BMUnit (2)

- **JUnit Byteman example**
  ```java
  package org.my.dbtests;
  @RunWith(BMUnitRunner.class)
  @BMScript(value="traceRules", dir="scripts")
  class DBTests1 {
      @Test
      @BMRule(className="FileOutputStream",
              methodName="\<init\>(File)",
              condition="\$1.getName().contains("Andrew")",
              action="THROW new FileNotFoundException")
      public void testDBFileHandler() {
          ...
      }
  }
  
  - **BMUnitRunner**
  - ensures agent installed in current VM
  - submits and un submits rules at the right time
DTest

• Client instruments remote server JVM
  – either methods, classes or packages

Instrumentor = new Instrumentor();
InstrumentedClass results;
Set String methods = new HashSet<String>();
methods.add("dbCreate");
methods.add("dbUpdate");
methods.add("dbDelete");
results =
  inst.instrument(DBManager.class, methods);
inst.injectFault(DBManager.class,
  "dbUpdate", "RuntimeException",
  new Object{ "throw by Byteman!" });
<invoke service >
DTest (2)

- Helper logs call trace via network
  - allowing client to validate execution

```java
results.assertMethodCalled("dbCreate", 2);
results.assertMethodNotCalled("dbDelete");
...```

• Summary
Summary

• Flexible, dynamic advice injection for
  – Testing
  – Tracing
  – Monitoring & Stats

• Easily understood by Java users

• Minimally invasive at runtime

• Supports complex advice suites

• Easily customised and extended in Java

• Java APIs allow sophisticated automation
• Questions
Questions

• **Byteman Project Page at JBoss**
  – http://www.jboss.org/byteman/

• **Downloads**
  – http://www.jboss.org/byteman/downloads
  – also in JBoss maven repo (groupid:org.jboss.byteman)
    • latest release 1.5.1.

• **Documentation**
  – http://www.jboss.org/byteman/documentation
    Programmers Guide (pdf)
  – contrib packages
    contrib/xxx/README.txt

• **User and Developer Forums**
  – follow link from project page

• **SVN Repository**
  – http://anonsvn.jboss.org/repos/byteman