**Document for Microbat Framework**

# Goal

Microbat is framework which can dynamically instrument Java bytecode to a target program so that we can keep track of its execution trace. Nevertheless, the instrumentation allows us to both observe and modifying the program runtime behavior. In this stage, we focus more on observing. In the long run, we will move on modifying and optimizing the runtime execution performance.

- For now, we may accidentally change the runtime behavior of the program for calling toString() method. We will discuss it later.

# Agent architecture

The design for microbat\_instrumentator is closed to Observer (Inceptor) Pattern. In this project, we inject agents into the program execution process & listen to favorable events such as when state/execution is reach. To implement this, there are a few types of components need to be identified/defined to do the job. Their relations can be inferred by Fig.1.

- **Observable**: the subject in which we can inject our agents.

- **Injector**: to inject agents into the program execution process.

- **Observer**: There are some observers to listen to different events, but amongst them there is one who does the main job of collecting data from the execution for our specific purpose, let call them Observer Headquarter.

- **Event**: sent from our agents to Observer whenever favorable state/execution is reach, it also implicitly indicates where to inject & why.

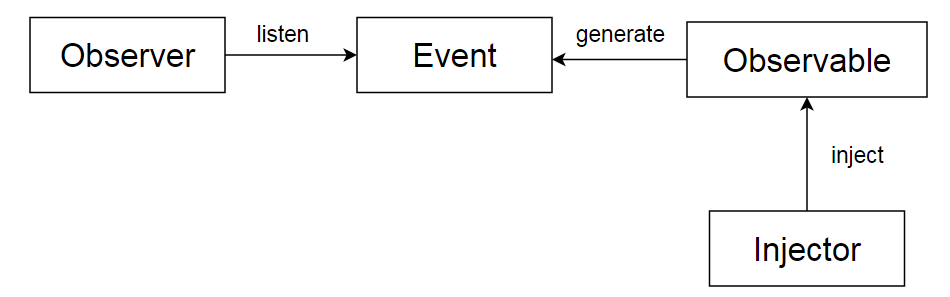


Fig.1. Relation between Four Components

~~There are 3 ways our agents are injected to the execution process:~~

There are 3 levels of events that establish 3 Observer-Event-Observable composites

## JVM-level ~~Instrumentation~~

JVM provides system events allow us to monitor state of program as well as JVM itself, but here we only listen to shutdown event to finalize our tracer.

- Observable: JVM

- Event: vm shutdown

- Injector: Agent

- Observer: Agent

*In class Agent, we register a listener for shutdown event, that start a thread to call stop() which cleans up transformer and further calls specific agent (TracerAgent/PrecheckAgent/..) for finalization process.*

**public** **void** startup(**long** vmStartupTime, **long** agentPreStartup) {

*agent*.startup(vmStartupTime, agentPreStartup);

Runtime.*getRuntime*().addShutdownHook(**new** Thread() {

@Override

**public** **void** run() {

Agent.*stop*();

}

});

}

**public** **static** **synchronized** **void** stop() {

**synchronized** (*shutdowned*) {

**try** {

**if** (!*shutdowned*) {

*instrumentation*.removeTransformer(*agent*.getTransformer());

…

*agent*.shutdown();

}

*shutdowned* = **true**;

} **catch** (Throwable e) {

AgentLogger.*error*(e);

*shutdowned* = **true**;

}

}

}

*Example of TraceAgent.shutdown()*

public void shutdown() throws Exception {

ExecutionTracer.*shutdown*();

/\* collect trace & store \*/

…

IExecutionTracer tracer = ExecutionTracer.*getMainThreadStore*();

Trace trace = ((ExecutionTracer) tracer).getTrace();

…

ExecutionTracer.*dispose*(); // clear cache

…

createVirtualDataRelation(trace);

…

AgentLogger.*debug*("construct ControlDomianceRelation....");

trace.constructControlDomianceRelation();

…

timer.newPoint("Saving trace");

writeOutput(trace);

AgentLogger.*debug*(timer.getResultString());

}

## 2. Library-level (Instrumentation part)

The purpose of this instrumentation part is to modify class bytecodes before it is loaded into jvm for the execution. The instrumentated bytecodes acts as event generator. They are actually instructions to load all valuable data at the current executing state and call our tracer for recording.

- Observable: Jre libs, User Application’s classes & library

- Event: enterMethod, exitMethod, fieldAccess, hitLine,…

- Injector: Instrumentator: AbstractInstrumenter[CoverageInstrumenter, TraceInstrumenter,..]

- Observer: Observer\_Headquater: ITracer [ExecutionTracer, CoverageTracer, TraceMeasurement]

First a **transformer** is registed into JVM

Premain class:

**public** **static** **void** premain(String agentArgs, Instrumentation inst) **throws** Exception {

…

*installBootstrap*(inst);

CommandLine cmd = CommandLine.*parse*(agentArgs);

Class<?>[] retransformableClasses = *getRetransformableClasses*(inst);

…

Agent agent = **new** Agent(cmd, inst);

agent.startup(vmStartupTime, agentPreStartup);

**inst.addTransformer(agent.getTransformer(), true);**

inst.addTransformer(**new** TestRunnerTranformer());

agent.retransformClasses(retransformableClasses);

*debug*("after retransform");

}

A transformer object is returned from TraceAgent:

@Override

**public** TraceTransformer getTransformer() {

**return** **new** TraceTransformer(agentParams);

}

In **TraceTransformer**, we call **TraceInstrumenter** for actual instrumentation:

**public** **class** TraceTransformer **extends** AbstractTransformer **implements** ClassFileTransformer {

**private** TraceInstrumenter instrumenter;

**public** TraceTransformer(AgentParams params) {

instrumenter = **new** TraceInstrumenter(params);

}

@Override

**protected** **byte**[] doTransform(ClassLoader loader, String classFName, Class<?> classBeingRedefined,

ProtectionDomain protectionDomain, **byte**[] classfileBuffer) **throws** IllegalClassFormatException {

/\* filter uninstrumented classes \*/

**if** ((loader == **null**) || (protectionDomain == **null**)) {

**if** (!GlobalFilterChecker.*isTransformable*(classFName, **null**, **true**)) {

**return** **null**;

}

}

**…**

/\* do instrumentation \*/

**try** {

**return** instrumenter.instrument(classFName, classfileBuffer);

} **catch** (Exception e) {

e.printStackTrace();

}

**return** **null**;

}

}

TraceInstrumenter:

An example of injecting bytecode for recording hitline using BCEL library:

For each method, we call instrumentMethod() for modifying method bytecodes. New class bytecode is presented as in a JavaClass object newJC. Finally, we call newJC.getBytes(); to return modified bytes array which will be used to replace original class bytes array content.

**public** **class** TraceInstrumenter **extends** AbstractInstrumenter {

**protected** **static** **final** String ***TRACER\_VAR\_NAME*** = "$tracer"; // local var

**private** **static** **final** String ***TEMP\_VAR\_NAME*** = "$tempVar"; // local var

**private** **int** tempVarIdx = 0;

@Override

**protected** **byte**[] instrument(String classFName, String className, JavaClass jc) {

ClassGen classGen = **new** ClassGen(jc);

ConstantPoolGen constPool = classGen.getConstantPool();

JavaClass newJC = **null**;

**for** (Method method : jc.getMethods()) {

**try** {

MethodGen methodGen = **new** MethodGen(method, classFName, constPool);

**boolean** change = instrumentMethod(classGen, constPool, methodGen, method);

**if** (change) {

// All changes made, so finish off the method:

InstructionList instructionList = methodGen.getInstructionList();

instructionList.setPositions();

methodGen.setMaxStack();

methodGen.setMaxLocals();

classGen.replaceMethod(method, methodGen.getMethod());

}

newJC = classGen.getJavaClass();

newJC.setConstantPool(constPool.getFinalConstantPool());

} **catch** (Exception e) {

AgentLogger.*error*(e);

}

}

**if** (newJC != **null**) {

**byte**[] data = newJC.getBytes();

**return** data;

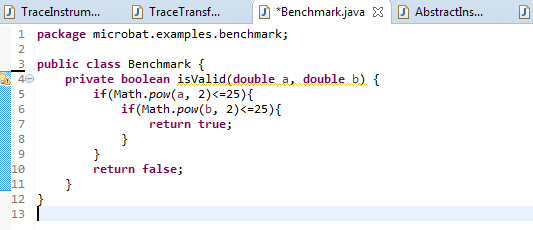
}

**return** **null**;

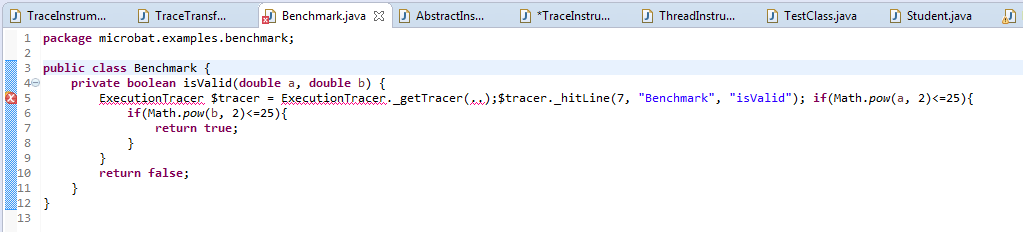
}

TraceInstrumenter.instrumentMethod()

For example: if our program is like this



The instrumented code to trace line number for code at line 5 will be like this:



In instrumentMethod(), first we append generated bytecode for calling method hitline() at beginning of each LineInstruction, then we append generated bytecode for Tracer initialization at the very beginning of the method. This order of instrumentation code makes sure the bytecode for Tracer initialization is always at the very top of instructions list.

/\*\*

\* **@return** whether method is changed

\*/

**protected** **boolean** instrumentMethod(ClassGen classGen, ConstantPoolGen constPool, MethodGen methodGen, Method method) {

tempVarIdx = 0;

InstructionList insnList = methodGen.getInstructionList();

InstructionHandle startInsn = insnList.getStart();

**if** (startInsn == **null**) {

// empty method

**return** **false**;

}

/\* fill up missing variables in localVariableTable \*/

LocalVariableSupporter.*fillUpVariableTable*(methodGen, method, constPool);

List<LineInstructionInfo> lineInsnInfos = LineInstructionInfo.*buildLineInstructionInfos*(classGen, constPool, methodGen, method, insnList);

LocalVariableGen classNameVar = createLocalVariable(***CLASS\_NAME***, methodGen, constPool);

LocalVariableGen methodSigVar = createLocalVariable(***METHOD\_SIGNATURE***, methodGen, constPool);

LocalVariableGen tracerVar = methodGen.addLocalVariable(***TRACER\_VAR\_NAME***, Type.*getType*(IExecutionTracer.**class**), insnList.getStart(), insnList.getEnd());

**for** (LineInstructionInfo lineInfo : lineInsnInfos) {

injectCodeTracerHitLine(insnList, constPool, tracerVar, lineInfo.getLine(), lineInfo.getLineNumberInsn(), classNameVar, methodSigVar, lineInfo.hasExceptionTarget(), lineInfo.getReadWriteInsnTotal(**false**), lineInfo.getReadWriteInsnTotal(**true**));

}

injectCodeInitTracer(methodGen, constPool, classNameVar,methodSigVar, tracerVar);

**return** **true**;

}

Example code for instrument bytecode for calling tracer.hitline() method:

First, we load tracerVar, then other values for tracer.\_hitline() method arguments including lineNumber, current executed class name, method signature. And then append instruction to invoke \_hitLine() method.

**protected** **void** injectCodeTracerHitLine(InstructionList insnList, ConstantPoolGen constPool, LocalVariableGen tracerVar, **int** line, InstructionHandle lineNumberInsn, LocalVariableGen classNameVar, LocalVariableGen methodSigVar, **boolean** isExceptionTarget, **int** readVars, **int** writtenVars) {

TracerMethods tracerMethod = TracerMethods.***HIT\_LINE***;

InstructionList newInsns = **new** InstructionList();

newInsns.append(**new** ALOAD(tracerVar.getIndex()));

newInsns.append(**new** PUSH(constPool, line));

newInsns.append(**new** ALOAD(classNameVar.getIndex()));

newInsns.append(**new** ALOAD(methodSigVar.getIndex()));

appendTracerMethodInvoke(newInsns, tracerMethod, constPool);

insertInsnHandler(insnList, newInsns, lineNumberInsn);

newInsns.dispose();

}

**protected** **void** appendTracerMethodInvoke(InstructionList newInsns, TracerMethods method, ConstantPoolGen constPool) {

**if** (method.isInterfaceMethod()) {

**int** index = constPool.addInterfaceMethodref(method.getDeclareClass(), method.getMethodName(), method.getMethodSign());

newInsns.append(**new** INVOKEINTERFACE(index, method.getArgNo()));

} **else** {

**int** index = constPool.addMethodref(method.getDeclareClass(), method.getMethodName(),

method.getMethodSign());

newInsns.append(**new** INVOKESTATIC(index));

}

}

After instrumenting user class bytecode, at runtime, it will call our ExecutionTracer.hitLine() . In this case of ExecutionTracer, whenever \_hitLine is called, a TraceNode will be added into our current Trace.

**public** **class** ExecutionTracer **implements** IExecutionTracer, ITracer {

**private** **static** ExecutionTracerStore *rtStore* = **new** ExecutionTracerStore();

@Override

**public** **void** \_hitLine(**int** line, String className, String methodSignature, **int** numOfReadVars, **int** numOfWrittenVars) {

**boolean** isLocked = locker.isLock();

locker.lock();

**try** {

**…**

TraceNode latestNode = trace.getLatestNode();

**if** (latestNode != **null** && latestNode.getBreakPoint().getClassCanonicalName().equals(className) && latestNode.getBreakPoint().getLineNumber() == line) {

locker.unLock(isLocked);

**return**;

}

**int** order = trace.size() + 1;

**if** (order > *stepLimit*) {

*shutdown*();

Agent.*\_exitProgram*("fail;Trace is over long!");

}

**if** (order > *tolerantExpectedSteps*) {

*shutdown*();

Agent.*\_exitProgram*("fail;Trace size exceeds expected\_steps!");

}

BreakPoint bkp = **new** BreakPoint(className, methodSignature, line);

TraceNode currentNode = **new** TraceNode(bkp, **null**, order, trace, numOfReadVars, numOfWrittenVars);

trace.addTraceNode(currentNode);

AgentLogger.*printProgress*(order, *expectedSteps*);

**if**(!methodCallStack.isEmpty()){

TraceNode caller = methodCallStack.peek();

caller.addInvocationChild(currentNode);

currentNode.setInvocationParent(caller);

}

} **catch** (Throwable t) {

handleException(t);

}

locker.unLock(isLocked);

}

## 3. use our engine which can act as notifier to trigger the execution of user application.

Here, to start User Application testcase we use our own TestRunner which defines some methods instrumented to trigger Obervers during runtime.

- Observable: MicroBatTestRunner, SavJunitRunner,..

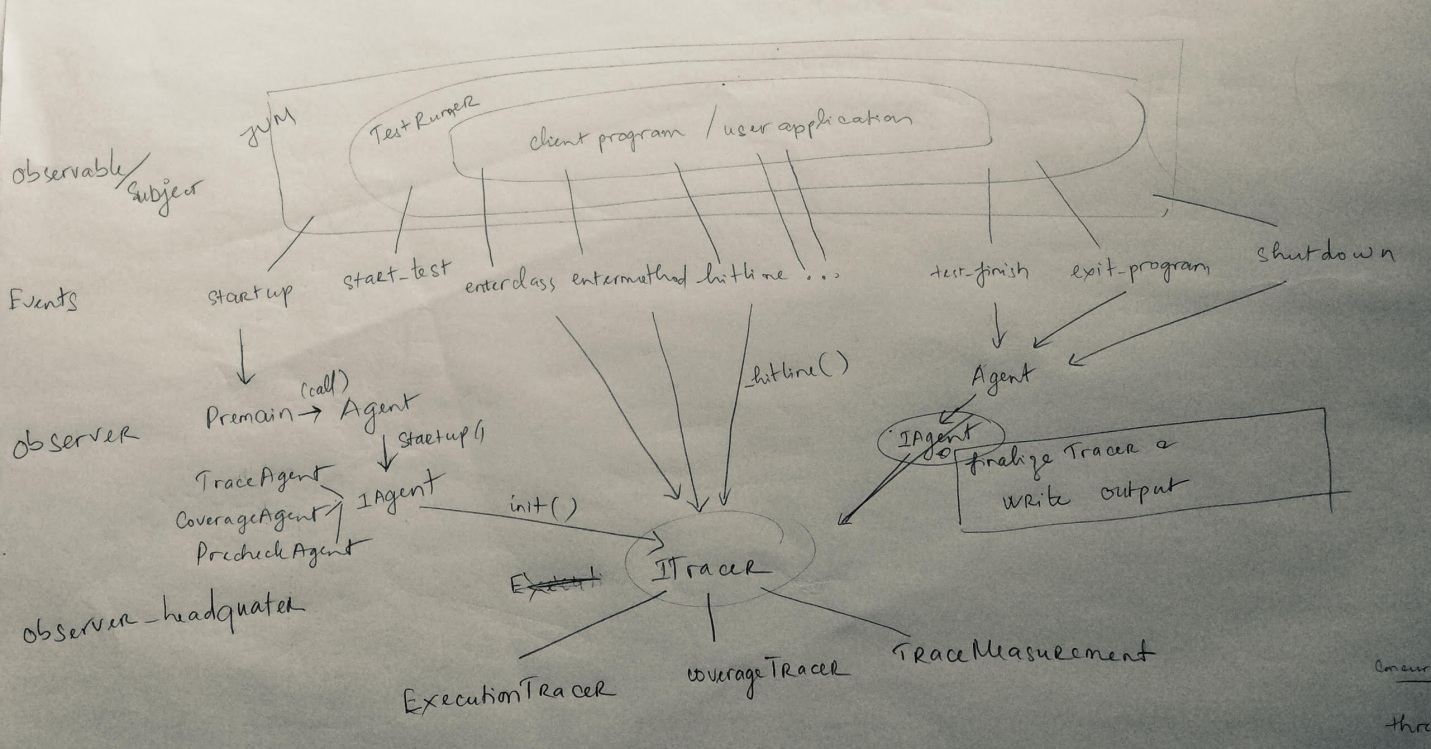
- Injector: TestRunnerTranformer

- Event: start\_test, exit\_test, exit\_program

- Oberver: Agent

*TODO: code example in the system.*

So overall, this is how our agent looks like:



# Tracer

# Implement a new Tracer or modify the existing one

To keep the original design idea not broken, for a new requirement, identify the events that you need to take into account, then from that, indicate corresponding Observable, Injector & Observer to work on.

And… you are in, Welcome on board! ☺

Example:

To extend ExecutionTracer to record concurrency program:

New event: thread start, thread end

->new Observable: Thread

Injector be added: ThreadInstrumentator

Observer to adapt:

Agent: to collect thread\_id to start a new Tracer with thread \_id, update output recording.

ExecutionTracer: make sure correct tracer is used to record the execution for specific thread.