**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:

## JVM-level Instrumentation

Such instrumentation deals with JVM & ask it to notify us for some of its events

- Observable: JVM

- Injector: Premain

- Event: before\_launching\_app /startup, vmShutdown/shutdown

- Observer:

Premain {startup}: add bycode transformer,…

IAgent [CoverageAgent, TraceAgent..]:

{startup()} init Observer\_Headquater.

{shutdown} finalize job of Observer\_Headquater.

*TODO: code example in the system.*

2. Library-level Instrumentation

Such instrumentation modifies bytecodes before it is loaded into jvm for the execution.

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

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

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

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

*TODO: code example in the system.*

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,..

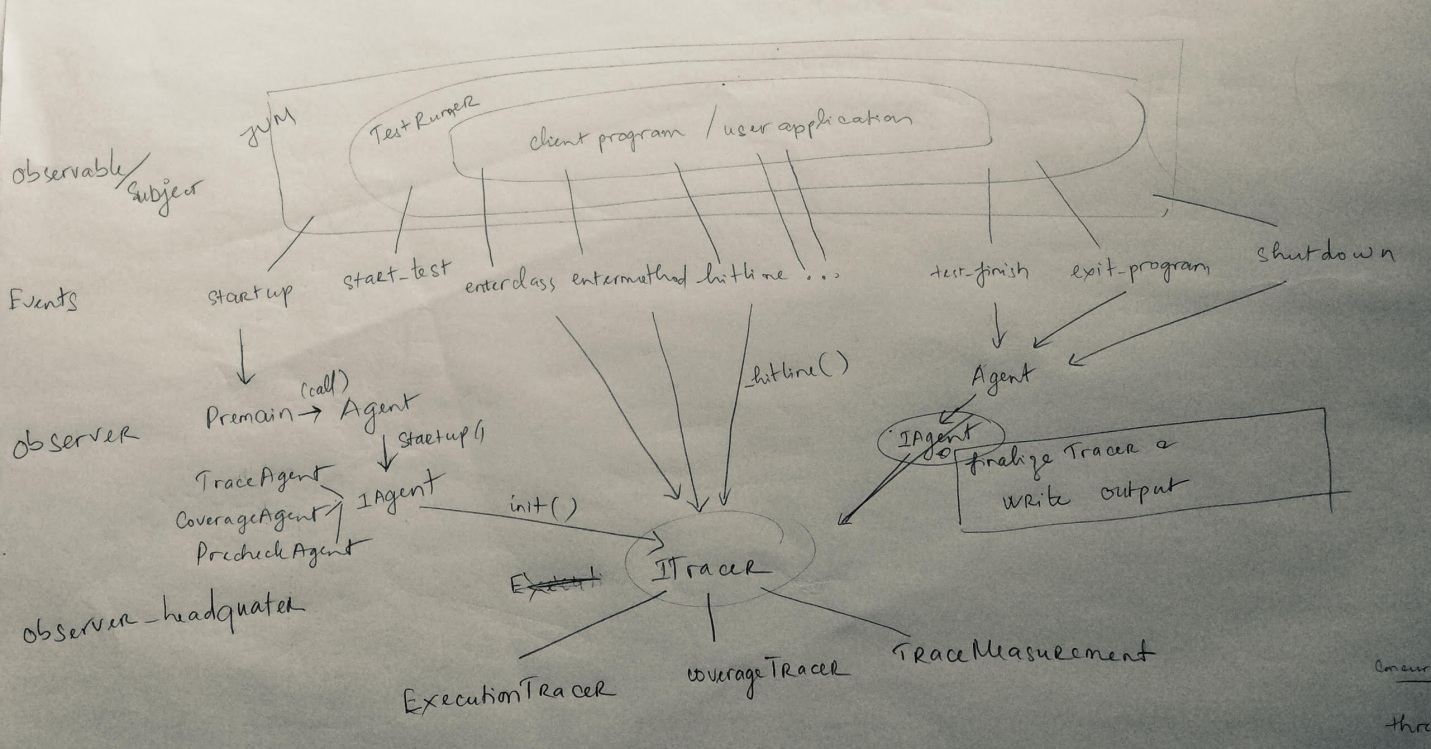
- Injector: TestRunnerTranformer

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

- Oberver: Agent

*TODO: code example in the system.*

So overall, this is how our agent looks like:



# 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.