Java Concurrency Lab Manual

Lab #3 – Building the Sensor implementation

Interfaces

Sensor

This is the interface to create a Sensor. You will be writing an implementation of this interface, that your SensorContoller will be invoking in a later lab. The SensorController will require two threads to work with a Sensor object:

1. The Thread that creates the Sensor, sets the Reporter implementation for the Sensor to use, stops the Sensor reporting and shutdown the Sensor
2. A Thread that startsReporting, the Thread that starts the reporting will go into a loop until the SensorStatus is changed, and this Thread is interrupted.

The SensorStatus value is used to control the Sensor operations, and will result in a Data Race condition if it is not designed, coded documented and tested properly.

SensorReporter

This is the interface that will be implemented by the SensorController to receive the SensorData being pushed by the Sensor.

SensorSimulator

The generator for the simulated values for the Sensor. There is a implementation of this interface provided in the starter code for your use.

Enumerations

SensorStatus

The enum holding the status values for a Sensor

Lab Assignment

You are to create a Sensor implementation. You must document and test your code. You will need to avoid a Data Race condition between the Thread that starts the recording thus setting the SensorStatus, and the Thread that stops the recording thus changing the SensorStatus.

You can choose any synchronization technique you wish.

You will need to prove you do not have a Data Race condition via a JUnit test with VMLens.

You will need to mock the SensorReporter interface for testing, you should consider mocking the SensorSimulator interface also.

Logging Threads

Using Log4J

To log thread activity using Log4j, you will need:

- The following Maven dependencies (you already have this in the maven-build-master):
  - org.apache.logging.log4j/log4j-api
  - org.apache.logging.log4j/log4j-core
- A log4j2.xml config file in one of these locations:
  - src/main/resources
  - src/test/resources
- A <PatternLayout> element with a pattern containing:
  - %T or %tid → Thread Id
  - %t or %threadName → Thread Name
  - %tp or %threadPriority → Thread Priority

Code Review

The Maven-Build-Master

The maven-build-master project is the home for all the common dependencies and plug-ins for the projects in the course. This project is defined as a POM deployment, and will be the Maven parent project for most of the projects in your workspace. This project has no Java source code, however it does have a number of configuration files including:

1. Custom checkstyle style rules → config/StyleRules.xml
2. Custom spotbugs-security configuration files → config/spotbugs-security-exclude.xml
   → config/spotbugs-security-include.xml
3. Custom dependency-check suppressions → config/suppressions.xml

The maven-build-master must be built before it can be used as a parent for another project. To build any Maven project: Right click the project in the Eclipse Explorer → Run As… → Maven install.

The maven-build-master has a <reporting> section that will be used for automated Code Review. To start the Code Review of your code: Right click your application project in the Eclipse Explorer → Run As… → Maven build… → Name the new run configuration: Code Review and set the Goal as: site. You can now rerun Code Review on this project by Right click your application project in the Eclipse Explorer → Run As… → Maven build (and selecting the configuration named: site)

The Code Review Report

After running the Maven site command on your project, you will need to Refresh (F5) the target folder in the project and open target → site → index.html with a Web Browser (Right click the index.html and select Open with… → Web Browser)

Project Reports

The Site Build will generate a standard set of reports, and our customized Code Review has included a number of additional reports.

- Project Information
  - The default Maven site reports customized by the contents of the pom.file
- Project Reports
  - Surefire Report → Report on the test results of the project
  - Javadoc → the generated Javadoc API documentation
  - SpotBugs → Generates a source code report with the SpotBugs Library
  - JaCoCo → JUnit Code Coverage Report
  - Checkstyle → Report on coding style conventions
  - CPD → Copy Paste Detection report
  - PMD → Source code analyzer for programming errors
  - Source Xref → HTML based, cross-reference version of Java source code
  - Test Source Xref → HTML based, cross-reference version of Java test source code
  - Dependency Check → Lists published vulnerabilities within project dependencies
  - DSM Report → Design Structure Matrix report of the project
  - Tag List → Lists the TODO, FIXME, @todo, or @depracated tags in the project

To use the automated Code Review, start with the Surefire report to confirm the JUnit tests succeeded. Next review the generated JavaDocs, followed by the JaCaCo Code Coverage report. Any code less than 80% code coverage should be examined. Next look at the Checkstyle report, the style rules are in the maven-master-build project, and you can choose to ignore style warnings. Next check the CPD and PDM reports for copy/paste issues and performance issues. Finally check the Dependency Check report of any project dependencies with reports security vulnerabilities.

Maven

Maven Update

Anytime you edit the Maven pom file (or the parent project is rebuilt) you should run Maven Update to inform Eclipse about the change. Right click project in an Eclipse explorer and select Maven → Update Project… .

Maven Clean

Running a Maven class will remove all the compiled source code, both main and test

Maven Test

Running a Maven test will recompile the source and test code if needed and run all the JUnit tests

Maven Build

To run a “custom” Maven build, select Run As → Maven build… and enter the Maven Goal(s)

For example, I can create a reusable custom Maven build by:

1) Naming the configuration: Clean-Test

2) Setting the Goal to: clean test

3) I will now have a Run Configuration in Eclipse I can reuse anytime by selecting

Run As → Maven Build → Selecting my named configuration

If there is only 1 build configuration for this project, I will not have to select it

I can edit/delete Maven run configuration via the Eclipse Run menu

VMLens

The open-source concurrency testing tool VMLens (https://vmlens.com) allows you to write a JUnit test for Data Races and Deadlocks.

A VMLens JUnit test looks like:

```
@Test
public void runConcurrentSensorVMLens() throws InterruptedException
{
	try (AllInterleavings testReporting =
		AllInterleavings.builder("TestSensorReporting")
			.showStatementsWhenSingleThreaded()
			.maximumRuns(10)
			 .build();)
	{
		while(testReporting.hasNext())
		{
			Sensor testSensorThread = new SensorImpl(simulatorMock);
			testSensorThread.setReporter(reporterMock);
			Thread runner1 = new Thread(() -> {
				testSensorThread.startReporting();
			}, "runner1");
			runner1.start();
			Thread.sleep(1000);
			testSensorThread.stopReporting();
			runner1.join();
		}
		testSensor.shutdown();
	}
}
```

This test will show a Data Race when the startReporting() and stopReporting() methods attempt to access a SensorStatus field at the same time without an explicit lock.

VMLens runs as an Eclipse plug-in (right click the JUnit test and select Run As → JUnit traced with vmlens. Or as part of a Maven build. The Code Review maven build will create the VMLens reports.

The report shows you the three potential outcomes of the interleave loop:

A successful loop, the symbol: 

An interrupted loop, the symbol: 

And a loop containing a data race, the symbol:  .

VMLens can be used to test remote applications also,

see: https://vmlens.com/help/manual/#all-other-applications for details.