A Framework for Automatic Diagnosis of Performance Problems

—

User Guide

Alexander Wert
alexander.wert@kit.edu
Am Fasanengarten 5
Software Design and Quality
Karlsruhe Institute of Technology

Denis Knöpfle
denis.knoepfle@sap.com
Vincenz-Prießnitz-Straße 1
SAP AG

Contributors

Alexander Wert, Christoph Heger, Roozbeh Farahbod, Denis Knöpfle, Peter Merkert, Marius Oehler, Henning Schulz

http://sopeco.github.io/DynamicSpotter

September 21, 2015

Contents

1 The Approach behind DynamicSpotter

DynamicSpotter is a framework for automatic detection of performance problems in Java-based enterprise software systems. Therefore, DynamicSpotter combines the concepts of software performance anti-patterns with systematic experimentation.

Software performance anti-patterns (SPAs) [SW00, SW02, SW03a, SW03b] describe common, recurring design or implementation mistakes leading to impaired software performance. As a big portion of software performance problems exhibit recurring nature, the concept of SPAs is a means to identify such problems in different contexts (different target-systems, environments, etc.) by searching for the generic patterns, or characteristics, different SPAs exhibit.

If the system under test (SUT) is implemented and can be executed, performance tests allow to gain insights on its performance behaviour. There are three important things to know about performance tests:

1. Usually, a load driver (such as JMeter™[Apa14] or HP LoadRunner [Hew14]) is used to generate a set of virtual users processing a script, which describes the work of single users.

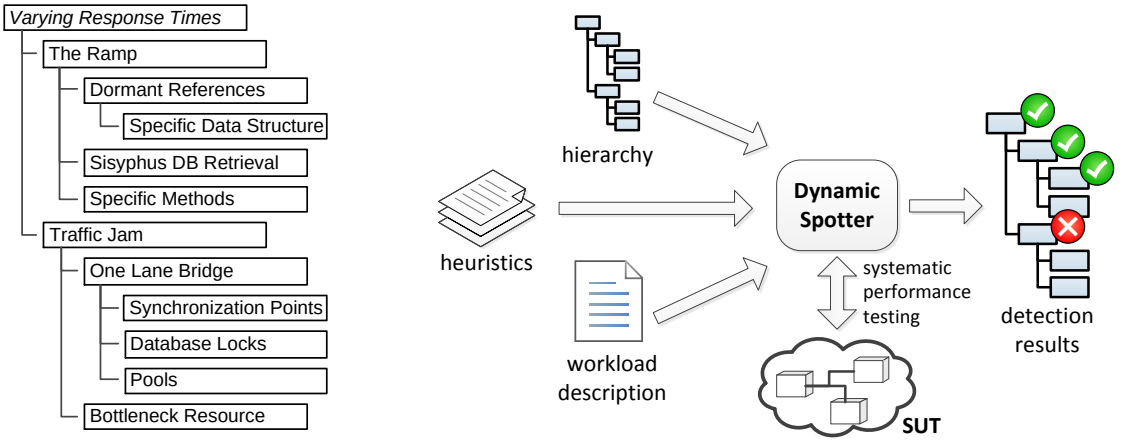
2. During load generation, performance metrics are retrieved from the SUT using instrumentation and monitoring techniques.

3. The gathered measurement data is analyzed to provide insights on particular performance engineering tasks.

In our context, the gathered measurement data can be mapped to the generic characteristics defined by individual SPAs. If measurement data matches certain pattern of an SPA, there is a high chance that the SUT contains the corresponding anti-pattern. However, the following circumstances render the detection of different SPAs with a single performance test impractical. Different SPAs refer to different parts, performance metrics and granularity levels of the SUT. Thus, in order to investigate different SPAs in a SUT as part of a single performance test, detailed and excessive instrumentation of the SUT is required. However, as each instrumentation instruction comes with a performance overhead, excessive instrumentation yields a high overhead which distorts measurement data, rendering analysis on this data useless. Hence, an experiment-based process is needed in which individual SPAs are investigated as part of individual performance tests (in the following called *experiments*). In this way, during each experiment the SUT is instrumented selectively for the corresponding SPA, providing detailed measurement data while keeping the performance overhead of the instrumentation low.

Though, the experiment-based process significantly mitigates the problem of the performance overhead introduced by the instrumentation, it increases the manual effort required to configure, execute and analyze each single experiment referring to an SPA. Moreover, the larger the set of investigated SPAs the more experiments are required, and with that the time required to execute the experiments grows. However, this problem can be mitigated by using an appropriate structure of the SPAs. Though there is a large set of different recurring performance problems, many of those problems share common characteristics and symptoms. Hence, performance problems can be structured in a systematic way, yielding a hierarchy from high level symptoms to specific root causes of the problems. An example of such a hierarchy is depicted in Figure 1(a).

Varying Response Times is a symptom. *The Ramp* [SW03b] and *Traffic Jam* [SW02] are potential causes of *Varying Response Times*. *The Ramp* occurs if response times of an application increase during operation. Such a behaviour can, for example, occur if the application contains *Dormant References* [RM07], i.e., the memory consumption of the application is growing over time. The root cause is *Specific Data Structure*s which are growing during operation or which are not properly disposed. The *Traffic Jam* performance antipattern constitutes another cause of *Varying Response Times*. A *Traffic Jam* occurs if many concurrent threads or processes are waiting for the same passive resource ((like semaphores or mutexes)) or active resource (like CPU or hard disk). In the first case, we have a typical *One Lane Bridge* [SW00] whose critical resource needs to be identified. We focus on *Synchronization Points*, *Database Locks*, and *Pools* as potential root causes. In the case of limited physical resources, the root cause can only be a specific *Bottleneck Resource*. Further examples on a performance problem hierarchy can be found in [WHH13, WOHF14, Wer13].



(a) An exemplary hierarchy of performance problems (from [WHH13])

(b) Overview on the DynamicSpotter approach (graphic from [WOHF14])

Figure 1: DynamicSpotter approach

A hierarchy as depicted in Figure 1(a) can be utilized as a search tree, in order to systematically search for performance problems by conducting a depth-first search on the tree. In particular, a branch or sub-tree of the hierarchy does not need to be investigated if the problem corresponding to the sub-tree's root node is not present in the SUT. In our example, possible root causes for *The Ramp* do not need to be investigated if *The Ramp* itself has not been detected in the SUT. In this way, many unnecessary experiments can be avoided, saving experimentation time and effort.

DynamicSpotter utilizes the concept of hierarchically structuring performance problems (respectively SPAs) in order to automate the search for recurring performance problems (SPAs) by systematically executing measurement experiments. The high level approach behind DynamicSpotter is depicted in Figure 1(b). DynamicSpotter takes a performance problem hierarchy (as described before) as input. For each node of that hierarchy, performance experts define a heuristic responsible to decide on the existence of the corresponding problem. To this end, a heuristic executes a series of experiments, observes certain performance metrics and analyzes them to make a decision. (Note: Both, the hierarchy and the corresponding heuristics are generic artifacts which do not depend on the SUT under test and, thus, can be reused in other contexts.) For instance, in order to detect a *One Lane Bridge*, the corresponding heuristic may execute a set of experiments with different load intensities, while observing end-to-end response times. If response times significantly increase with the load, while none of the hardware resources is utilized to capacity, the SUT contains an OLB. For execution of experiments, a load script (*workload description*) specifies the work of single virtual users. Traversing the hierarchy and applying corresponding heuristics for each performance problem of the hierarchy, DynamicSpotter generates a detection result report. The report states for each node in the hierarchy whether the corresponding problem exists in the SUT and, where appropriate, points to the root cause and location in the SUT of a detected problem.

2 Architecture of DynamicSpotter

As described in Section 1, DynamicSpotter is a framework for automatic detection of performance problems. The architecture of DynamicSpotter is depicted in Figure 2. The main component of DynamicSpotter (`DynamicSpotter Core`) realizes the logic for automating performance tests and data analysis. In particular, `DynamicSpotter Core` is responsible for coordinating the instrumentation of the system under test (SUT), the monitoring process, gathering and pre-processing measurement data, as well as analyzing data. Furthermore, `DynamicSpotter Core` implements the high level process of iterating a performance problem hierarchy (cf. Component `Process Controller` in Figure 2), as described in Section 1. The `Experiment Execution` component is responsible for automating experiment execution, while the `Data Analysis` component conducts the data analysis task for individual performance problems according to the detection rules defined for corresponding performance problems in form of *Heuristics*. For each sub-task, DynamicSpotter provides extension points (denoted with ≪`EP`≫ in Figure 2) allowing to provide adapters for specific implementations or tools used for instrumentation, monitoring, workload generation, and data analysis:

- **≪EP≫ Load Driver Adapter:** The `Load Driver Adapter` extension point provides means to use existing load driver tools like JMeter, Faban or HP LoadRunner for workload generation. In particular, a `Load Driver Adapter` needs to provide means to control the execution of performance tests programmatically. Furthermore, the adapter realizes a remote communication between DynamicSpotter and the remotely running load generation tool.
- **≪EP≫ Instrumentation Adapter:** The `Instrumentation Adapter` extension point allows to provide adapters for instrumentation tools like DiSL, Kieker, our own instrumentation tool AIM (Adaptable Instrumentation and Monitoring), etc. Thereby, DynamicSpotter uses a generic instrumentation description model (IDM) to decouple the instrumentation description from the tool realizing it. Thus, an instrumentation adapter for a specific instrumentation tool has to realize the transformation from the IDM instances to the tool-specific representation of the instrumentation description.
- **≪EP≫ Measurement Adapter:** While an instrumentation adapter is solely responsible for instrumenting the SUT, a `Measurement Adapter` is used to enable and disable data collection, as well as to transform and transfer data from the monitoring tool to DynamicSpotter in a common data representation format. Though a specific instrumentation adapter and a measurement adapter are often realized within one external tool, they represent conceptually different tasks. In such a case, adapters for both extensions points are required, even if only one tool realizes both tasks.
- **≪EP≫ Detection Heuristic:** As described in Section 1, DynamicSpotter uses a performance problem hierarchy and corresponding heuristics to guide the detection process. Both artifacts need to be provided by performance experts. For each performance problem defined in the hierarchy, a performance expert can provide a `Detection Heuristic` extension which is then responsible to define the experiments and analyse the corresponding measurement data for the corresponding performance problem.

Depending on the size of the SUT, DynamicSpotter may use several `Instrumentation Adapters`, `Measurement Adapters` and `Load Driver Adapters`. A set of adapters to be used in a specific application context of DynamicSpotter determines the *Measurement Environment* for DynamicSpotter . In order to use DynamicSpotter , a user has to specify the *Measurement Environment* in a `configuration`. Thereby, a user either can start a headless DynamicSpotter process (cf. DynamicSpotter Runner in Figure 2) providing the `configuration` as input, or, the user harnesses the `DynamicSpotter Eclipse-Plugin` to build a `DynamicSpotter configuration` in an interactive way, trigger the automatic detection process, and finally, examine the detection results in the graphical user interface. The `DynamicSpotter Eclipse-Plugin` communicates with a `DynamicSpotter Service` layer running on top of the `DynamicSpotter Core`. Utilizing the description of the *Measurement Environment*, DynamicSpotter identifies required extensions and establishes connections to the corresponding external instrumentation, measurement and load generation tools. DynamicSpotter is then able to automatically run performance tests and analyze measurement data as defined in

the heuristics of the individual performance problems from the performance problem hierarchy. As a result, DynamicSpotter provides a report to the user stating which performance problems have been detected.

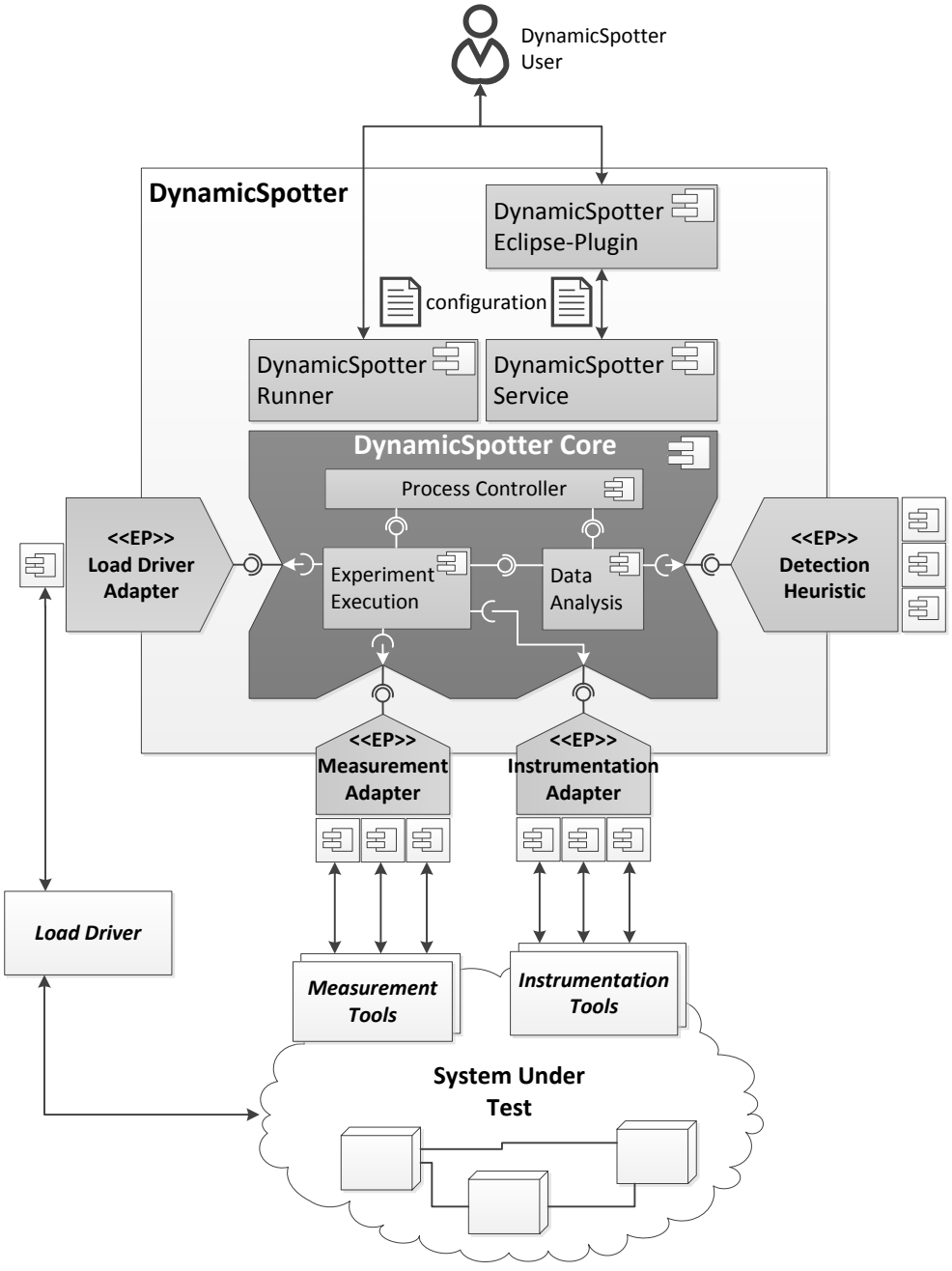


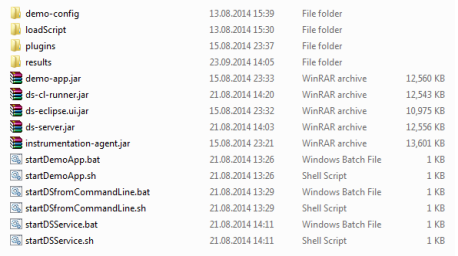
Figure 2: DynamicSpotter architecture

3 DynamicSpotter - Quick Start

To get a demo for DynamicSpotter running, follow the following steps.
Note: The following screenshots show the execution in Windows and work similarly in Linux.

1. First, you need a Java JDK 1.7 to be installed.
2. Download the Demo-Bundle from the following URLs:

   Windows: https://github.com/sopeco/DynamicSpotter-Demo/releases/download/v1.0/demo-all-in-one.zip
   Linux: https://github.com/sopeco/DynamicSpotter-Demo/releases/download/v1.0/demo-all-in-one.tar.gz
3. Unpack the archive to an arbitrary directory. Lets call it `DEMO_HOME`:



4. Open a command line window (e.g. `cmd` in Windows) and go to the `DEMO_HOME` directory with `cd PATH_TO_DEMO_HOME`.
   Linux only: make the script files executable with: `chmod u+x ./*.sh`
5. Start the target application (System under Test) by executing the following command line script:

   Windows: `startDemoApp.bat`
   Linux: `sh ./startDemoApp.sh`

```
C:\Users\c5170547\Desktop\DEMO_HOME>
C:\Users\c5170547\Desktop\DEMO_HOME>startDemoApp.bat_
```

   If you see a line ending with
   `...- Web-Server started on port 8081`
   the target application has been started successfully:

   IMPORTANT: Do not stop the application!
6. Open a *second* command line window and again go to the `DEMO_HOME` directory
7. Start DynamicSpotter analysis by executing the following command line script:

   Windows: `startDSfromCommandLine.bat`
   Linux: `sh ./startDSfromCommandLine.sh`

   If DynamicSpotter has been started successfully you should see progress report in the command line:
8. In the demo scenario DynamicSpotter runs about 15 minutes to execute all required experiments and analyze the measurement data.

```
Sep 24, 2014 10:41:08 AM com.sun.jersey.api.core.PackagesResourceConfig init
INFO: Scanning for root resource and provider classes in the packages:
  org.codehaus.jackson.jaxrs
  org.spotter.ext.demo.app
  org.lpe.common.util.web
Sep 24, 2014 10:41:10 AM com.sun.jersey.api.core.ScanningResourceConfig logClasses
INFO: Root resource classes found:
  class org.spotter.ext.demo.app.DummyApp
  class org.lpe.common.util.web.ShutdownService
Sep 24, 2014 10:41:10 AM com.sun.jersey.api.core.ScanningResourceConfig logClasses
INFO: Provider classes found:
  class org.codehaus.jackson.jaxrs.JacksonJsonProvider
  class org.codehaus.jackson.jaxrs.JacksonJaxbJsonProvider
  class org.codehaus.jackson.jaxrs.JsonMappingExceptionMapper
  class org.codehaus.jackson.jaxrs.JsonParseExceptionMapper
Sep 24, 2014 10:41:10 AM com.sun.jersey.server.impl.application.WebApplicationImpl _initiate
INFO: Initiating Jersey application, version 'Jersey: 1.17.1 02/28/2013 03:28 PM'
Sep 24, 2014 10:41:12 AM org.glassfish.grizzly.http.server.NetworkListener start
INFO: Started listener bound to [0.0.0.0:8081]
Sep 24, 2014 10:41:12 AM org.glassfish.grizzly.http.server.HttpServer start
INFO: [HttpServer-1] Started.
10:41:12.270 [main] INFO  org.lpe.common.util.web.WebServer - Web-Server started on port 8081
```

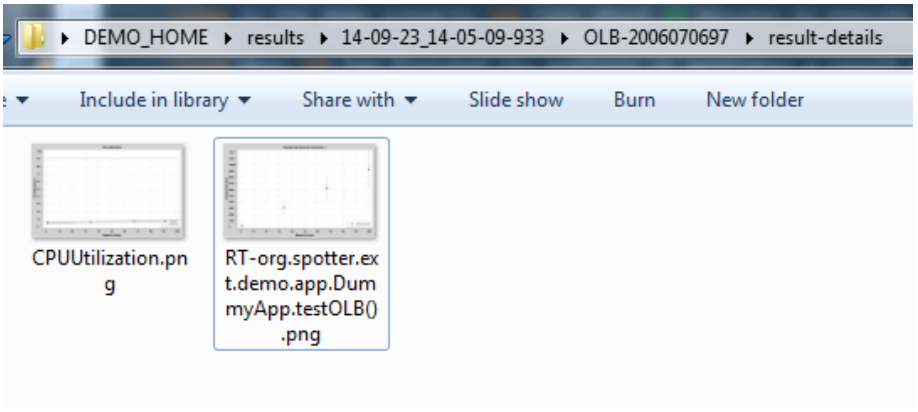
```
C:\Users\c5170547>cd Desktop\DEMO_HOME

C:\Users\c5170547\Desktop\DEMO_HOME>startDSfromCommandLine.bat_
```

```
10:47:57.914 [main] DEBUG o.l.c.extension.ExtensionRegistry - Loading extension org.aim.artifacts
10:47:57.915 [main] DEBUG o.l.c.extension.ExtensionRegistry - Loading extension measurement.satel
10:47:57.917 [main] DEBUG o.l.c.extension.ExtensionRegistry - Loading extension org.aim.artifacts
10:47:59.058 [pool-1-thread-1] INFO  org.spotter.core.ProgressManager - Progress - OLB - 00,0% -
10:47:59.060 [main] DEBUG o.s.e.w.simple.SimpleWorkloadDriver - waiting for finished load ...
10:47:59.061 [pool-1-thread-2] INFO  o.s.e.w.simple.SimpleWorkloadDriver - starting 1 vUsers ...
10:47:59.061 [main] DEBUG o.s.e.w.simple.SimpleWorkloadDriver - activeUsers: 0
10:47:59.063 [main] DEBUG o.s.e.w.simple.SimpleWorkloadDriver - warmupPhaseFinished: false
10:47:59.063 [main] DEBUG o.s.e.w.simple.SimpleWorkloadDriver - experimentPhaseFinished: false
10:47:59.168 [main] DEBUG o.s.e.w.simple.SimpleWorkloadDriver - activeUsers: 1
10:47:59.168 [main] DEBUG o.s.e.w.simple.SimpleWorkloadDriver - warmupPhaseFinished: true
10:47:59.169 [main] DEBUG o.s.e.w.simple.SimpleWorkloadDriver - experimentPhaseFinished: false
10:48:00.058 [pool-1-thread-1] INFO  org.spotter.core.ProgressManager - Progress - OLB - 00,1% -
10:48:01.059 [pool-1-thread-1] INFO  org.spotter.core.ProgressManager - Progress - OLB - 00,2% -
10:48:02.059 [pool-1-thread-1] INFO  org.spotter.core.ProgressManager - Progress - OLB - 00,3% -
10:48:03.059 [pool-1-thread-1] INFO  org.spotter.core.ProgressManager - Progress - OLB - 00,5% -
10:48:04.059 [pool-1-thread-1] INFO  org.spotter.core.ProgressManager - Progress - OLB - 00,6% -
10:48:05.059 [pool-1-thread-1] INFO  org.spotter.core.ProgressManager - Progress - OLB - 00,7% -
```

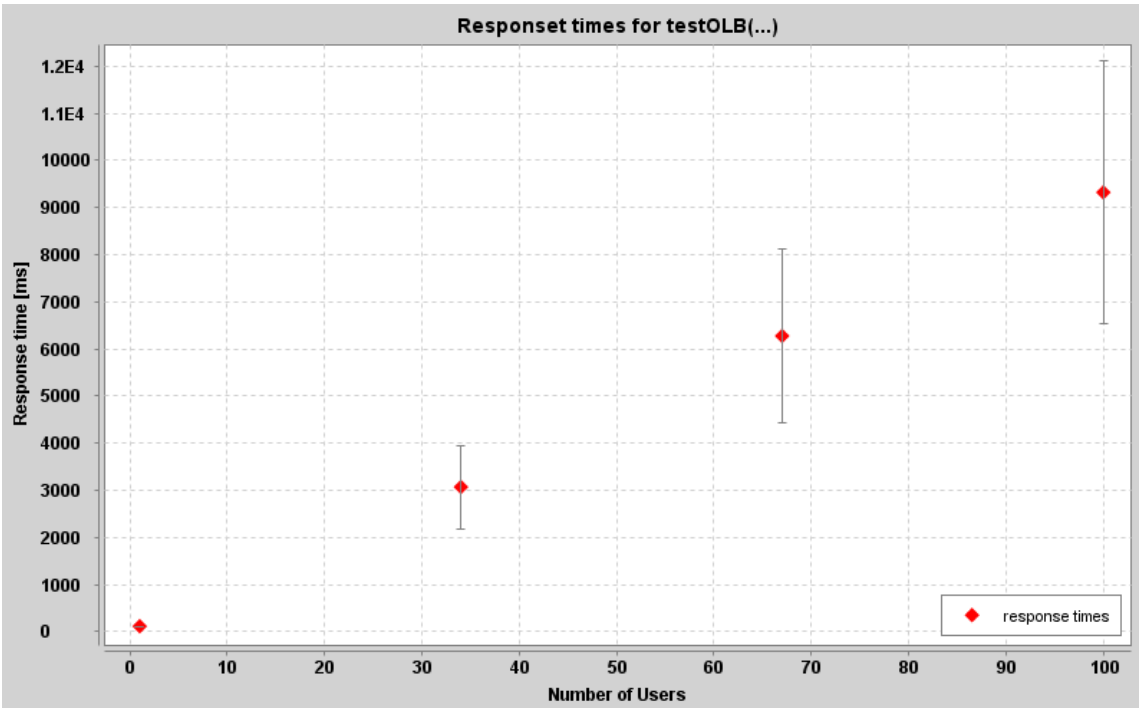
9. When DynamicSpotter has finished its analysis it generates a result report in the `DEMO_HOME/results` directory. There DynamicSpotter creates a folder with the latest timestamp as folder name. Go to that directory.

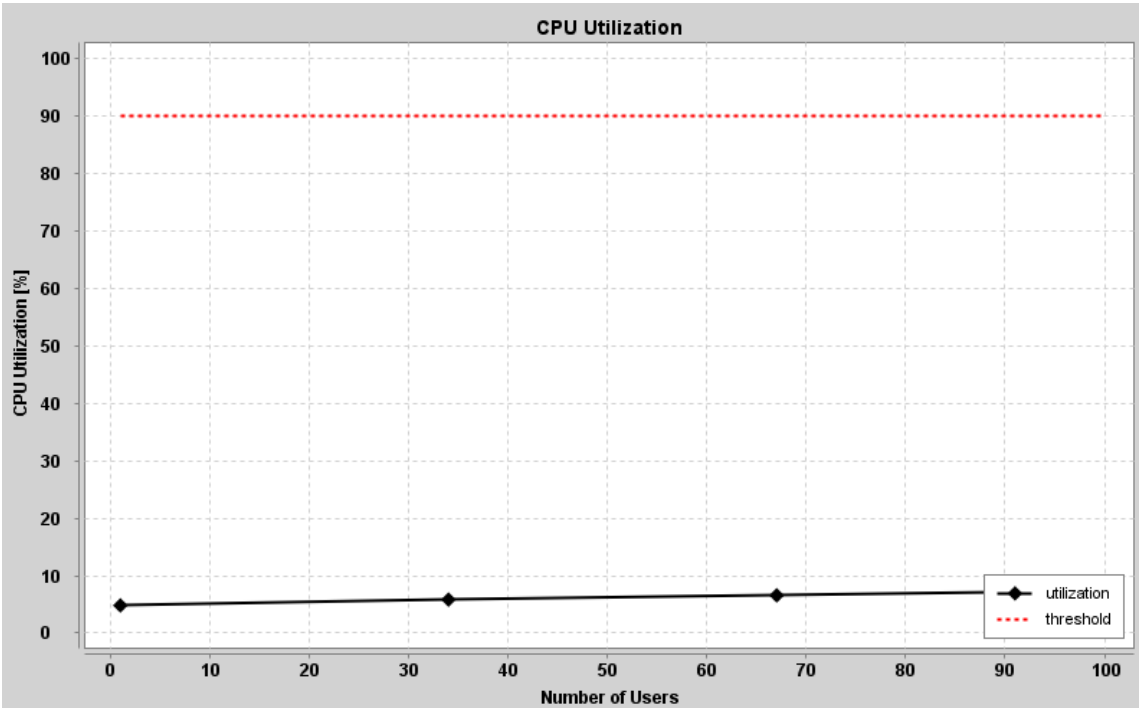
10. The `SpotterReport.txt` file contains a textual description of which problems have been detected or not.

11. If you further go to the individual directories of the anti-patterns (e.g. OLB_2006070899), there you find the raw measurement data in the `csv` folder and (if available) some graphs in the `result-details` folder illustrating the problem:



The following graphs for instance show that the response times of the *testOLB()* service grow with the load, while the CPU utilization stays low.





Please read the next chapter to get more details on the Demo scenario and to see how to configure DynamicSpotter using an Eclipse plugin.

4 DynamicSpotter - Getting Started

In this section, we demonstrate the usage of DynamicSpotter on a very simple scenario. Some of the DynamicSpotter plugins we use in this scenario are just simple examples (e.g. the plugin for load generation) to demonstrate the DynamicSpotter framework. In real scenarios more sophisticated DynamicSpotter plugins would be used, such as the JMeter plugin for load generation.

We show how to configure the target application, set-up DynamicSpotter , run DynamicSpotter diagnosis and view diagnosis results.

4.1 Requirements

In order to run through the demo example in this section, the following is required:

- JDK (tested with JDK 1.7)
  http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html
- An Eclipse standard installation (tested with Eclipse Kepler)
  http://www.eclipse.org/downloads/packages/eclipse-standard-432/keplersr2

4.2 All-In-One Demo Example

All binaries required for executing the demo scenario are available in the `All-In-One` archive which you can download from the following URLs:

Windows: https://github.com/sopeco/DynamicSpotter-Demo/releases/download/v1.0/demo-all-in-one.zip
Linux: https://github.com/sopeco/DynamicSpotter-Demo/releases/download/v1.0/demo-all-in-one.tar.gz

Unpack the archive in any directory. The content of the archive is the following:

```
/
├── demo-app.jar
├── instrumentation-agent.jar
├── ds-server.jar
├── plugins
│   ├── instrumentation-plugin.jar
│   ├── measurement-plugin.jar
│   ├── loadgeneration-plugin.jar
│   ├── detection-olb-plugin.jar
│   └── detection-ramp-plugin.jar
├── ds-eclipse-ui.jar
├── loadScript
│   └── org
│       └── spotter
│           └── ext
│               └── demo
│                   └── load
│                       └── VUser.class
├── ds-cl-runner.jar
└── demo-config
    ├── hierarchy.xml
    ├── mEnv.xml
    └── spotter.conf
```

The `demo-app.jar` comprises the target demo application used in the following scenario as the system under test (SUT). In order to be able to retrieve some performance data from the target application, we provide an `instrumentation-agent.jar` which allows to instrument the

target application and gather measurement data. Thereby, `instrumentation-agent.jar` contains the Java-agent from our own instrumentation tool AIM (adaptable instrumentation and monitoring) also available on GitHub (cf. Section 7). The instrumentation agent needs to be started with the target application as an additional JVM parameter, which we will explain later in more detail. The DynamicSpotter framework is bundled in the `ds-server.jar`. The `plugins` directory contains a set of extensions for DynamicSpotter which we need for this demo scenario. In particular (according to the architecture described in Section 2), we need an instrumentation (`instrumentation-plugin.jar`) and measurement adapter (`measurement-plugin.jar`) to connect to the instrumentation agent, a load generation plugin (`loadgeneration-plugin.jar`) to be able to submit load on the target application, and some detection heuristics (`detection-olb-plugin.jar` and `detection-ramp-plugin.jar`) for the analysis of certain performance problems. For simplicity, in this scenario we use only two detection heuristics, one for detecting a One Lane Bridge anti-pattern (software bottleneck) and a heuristic for the detection of the Ramp anti-pattern (slowly decreasing software performance). The `ds-eclipse-ui.jar` is a plugin for Eclipse comprising the graphical user interface for DynamicSpotter . Finally, the `loadScript` directory contains a load scenario for the target application in form of a Java class (including the package-directory structure). In order to run DynamicSpotter in a batch mode from the command line one can use the `ds-cl-runner.jar`. The demo-config folder contains demo configuration for the execution of DynamicSpotter from command line.

4.3 Demo Application

For our demonstration scenario we use a dummy application providing two services. A service containing a One Lane Bridge (OLB) performance anti-pattern and a second service without any performance problems. Listing 1 shows the code of the target dummy application. The `DummyApp` class comes with two REST services: `testOLB` and `fibonacci`. The first service contains an OLB anti-pattern manifested in a call to a synchronized method (`OLB.olbMethod()`) becoming a software bottleneck. This dummy application runs on a Web server such that users of that application can access the services via a REST interface.

Listing 1: Code of the Demo Application

```
public class DummyApp {
   // A dummy service containing a One Lane Bridge anti-pattern
   @GET @Path("testOLB")
   public String testOLB()
    throws InterruptedException {
      System.out.println("OLB called");
      OLB.getInstnace().olbMethod();
      return "Hello from OLB Test Method!";
   }

   // Another dummy service containing no performance problems.
   @GET @Path("fibonacci")
   public String fibonacci() {
      fibonacci(4);
      return "Hello from Fibonacci Method!";
   }

   // Recursive calculation of fibonacci.
   private int fibonacci(int n) {
      if (n <= 1)
         return 1;
       else
         return fibonacci(n - 2) + fibonacci(n - 1);
   }
}
```

```
public class OLB {
    ...
    // Method leading to a One Lane Bridge.
    public synchronized void olbMethod() throws InterruptedException {
        Thread.sleep(100); // sleep 100 ms
    }
}
```

In order to execute the demo application execute the following command in the directory you have unpacked the all-in-one archive to:

```
java -javaagent:instrumentation-agent.jar -jar demo-app.jar start
```

With `-jar demo-app.jar start` we instruct the demo application to start. Additionally we provide a `-javaagent` JVM argument pointing to the `instrumentation-agent.jar`. This Java agent is later used to dynamically instrument the bytecode of the target application in order to retrieve different types of measurement data. The Java agent starts a REST service on port `8888`.

4.4 Load Script

As described in Section 2, for the performance tests executed by DynamicSpotter , a load script is required which describes the behaviour of the target application users. Depending on the tool used for load generation, the load script has a different representation. For instance, HP LoadRunner and Apache JMeter each have proprietary representations of the load script. For the sake of simplicity, in this demo scenario we use a very minimalistic load generator which repeatedly executes a Runnable for each emulated user. Thus, the load script is defined by a Java class implementing a certain interface. Listing 2 shows the load script we use in this demo scenario. It is the Java code the `org.spotter.ext.demo.load.VUser` class which you can find in the `loadScript` directory of the all-in-one drop.

Listing 2: Load Script

```
import java.util.Random;
import javax.ws.rs.core.MediaType;

import org.lpe.common.util.web.LpeWebUtils;
import org.spotter.ext.workload.simple.ISimpleVUser;

import com.sun.jersey.api.client.Client;
import com.sun.jersey.api.client.WebResource;

public class VUser implements ISimpleVUser {

    private static final int THINK_TIME_MIN = 100;
    private static final int THINK_TIME_MAX = 200;
    private static final Random random = new Random(System.nanoTime());
    private final WebResource webResource;

    public VUser() {
        Client client = LpeWebUtils.getWebClient();
        webResource = client.resource("http://localhost:8081/");
    }

    @Override
```

```
    public void executeIteration() {
        try {
            // call OLB service
            webResource.path("demo").path("testOLB").
                accept(MediaType.APPLICATION_JSON).get(String.class);

            Thread.sleep(getNextThinkTime());

            // call Fibonacci service
            webResource.path("demo").path("fibonacci").
                accept(MediaType.APPLICATION_JSON).get(String.class);

            Thread.sleep(getNextThinkTime());
        } catch (Throwable e) {
            // ignoring exception
            e.printStackTrace();
        }
    }

    private long getNextThinkTime() {
        int r = random.nextInt(THINK_TIME_MAX - THINK_TIME_MIN);
        return THINK_TIME_MIN + r;
    }

}
```

Assuming a closed workload, the `executeIteration()` method of the `VUser` class defines the actions of an emulated user in one iteration of the loop. In this particular example, the emulated users first call the `testOLB` service of the Demo application and then call the `fibonacci` service. Inbetween the actions, the users idle for a think time between 0.1 and 0.2 seconds. Later when configuring DynamicSpotter , we will have to reference this class in order to specify the usage behaviour submitted to the target application.

4.5 Using the DynamicSpotter Eclipse UI

DynamicSpotter can be executed either as a service allowing an Eclipse-Plugin to connect to the DynamicSpotter service, or DynamicSpotter can be executed in a batch mode from the command line. In this section we show how to use DynamicSpotter from the Eclipse UI.

Starting DynamicSpotter Service

As shown in Figure 2, the DynamicSpotter Eclipse-Plugin is decoupled from the core of DynamicSpotter . The DynamicSpotter Eclipse-Plugin communicates with the DynamicSpotter Service over a REST interface. Thus, in order to use DynamicSpotter with the Eclipse UI, first, we need to start the DynamicSpotter Service. From the command line you can execute the following command to start the DynamicSpotter Service (execute from the directory you unpacked the all-in-one drop to):

```
java -jar ds-server.jar start
```

This will start the DynamicSpotter Service on the default port `8080`. In order to load all available plugins, DynamicSpotter looks for a `plugins` folder in the directory you have executed DynamicSpotter from. From that directory, DynamicSpotter dynamically loads all jars representing DynamicSpotter extensions.

In order to use another port, or another root directory for the `plugins` folder, the following program arguments can be used to specify the port and the root directory:

```
java -jar ds-server.jar start port=<PORT> rootDir=<ROOT_DIR>
```

Starting Eclipse with the DynamicSpotter Plugin

If you do not have an Eclipse installation yet, download and install Eclipse:

http://www.eclipse.org/downloads/packages/eclipse-standard-432/keplersr2

Drop the `ds-eclipse-ui.jar` into the plugins directory of your Eclipse installation and start Eclipse. Under `Window` → `Open Perspective` → `Other` change to the perspective `DynamicSpotter` (cf. Figure3).

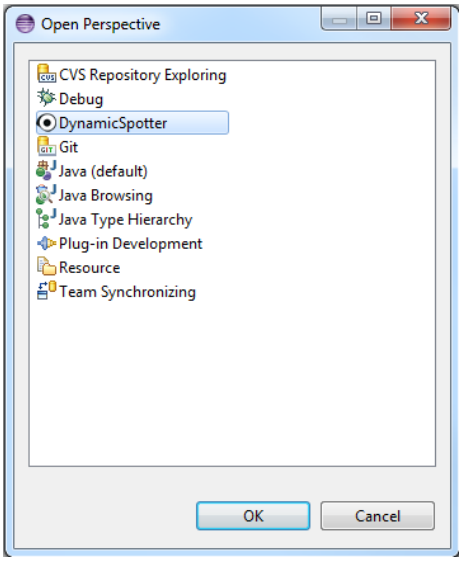
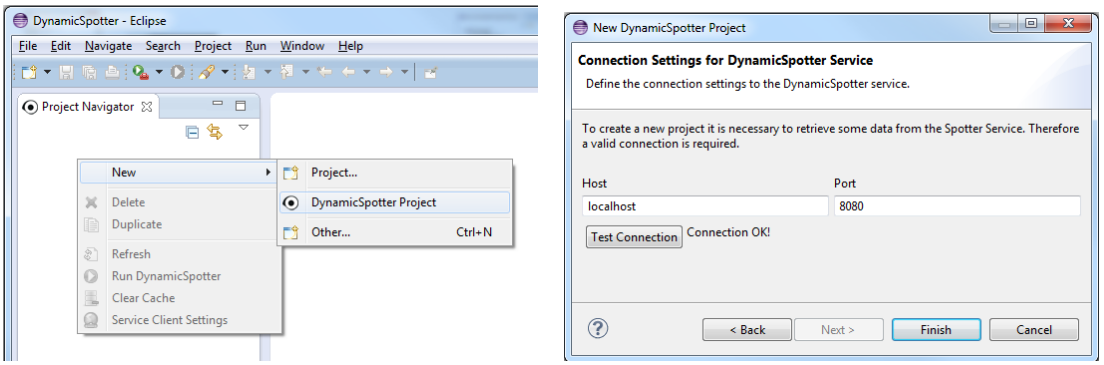


Figure 3: Open DynamicSpotter perspective

Creating and Configuring a DynamicSpotter Project

The DynamicSpotter Eclipse-Plugin provides the notion of DynamicSpotter Project. A DynamicSpotter Project covers the configuration, execution and results of DynamicSpotter for a specific scenario (i.e. system under test). For our demo application scenario we first need to create a DynamicSpotter Project. Therefore, select `New` → `DynamicSpotter Project` in the context menu of DynamicSpotters's Project Navigator (cf. Figure4(a)). In the project wizard, a name for the project and the configuration of the connection to the DynamicSpotter Service are required (cf. Figure 4(b)). The new project should appear in the Project Navigator (cf. Figure 5).



(a) Context Menu

(b) DynamicSpotter Project Wizard

Figure 4: Creating DynamicSpotter Project

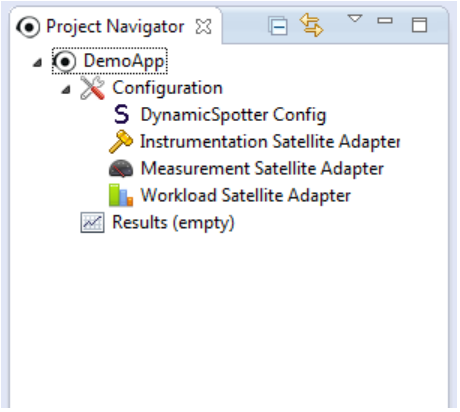


Figure 5: Project Structure

A DynamicSpotter Project has two main nodes: a `Configuration` part and a `Results` part. In the `Configuration` part, one has to specify the scenario-specific properties as well as domain specific configuration parameters for DynamicSpotter . The `DynamicSpotter Config` contains global configurations and domain specific information required for the execution of DynamicSpotter on the specific system under test. The `Satellite Adapter` configurations allow to specify the measurement environment of the scenario including configuration of instrumentation, measurement and workload adapters. The `Results` part provides a history of results for DynamicSpotter diagnosis runs executed for that DynamicSpotter Project.

In the following, we explain how to configure DynamicSpotter for our demo scenario. First, we edit the `DynamicSpotter Config` by opening (double-click on `DynamicSpotter Config` in the Project Navigator) the `DynamicSpotter Config Editor` (cf. Figure 6). We can edit the individual

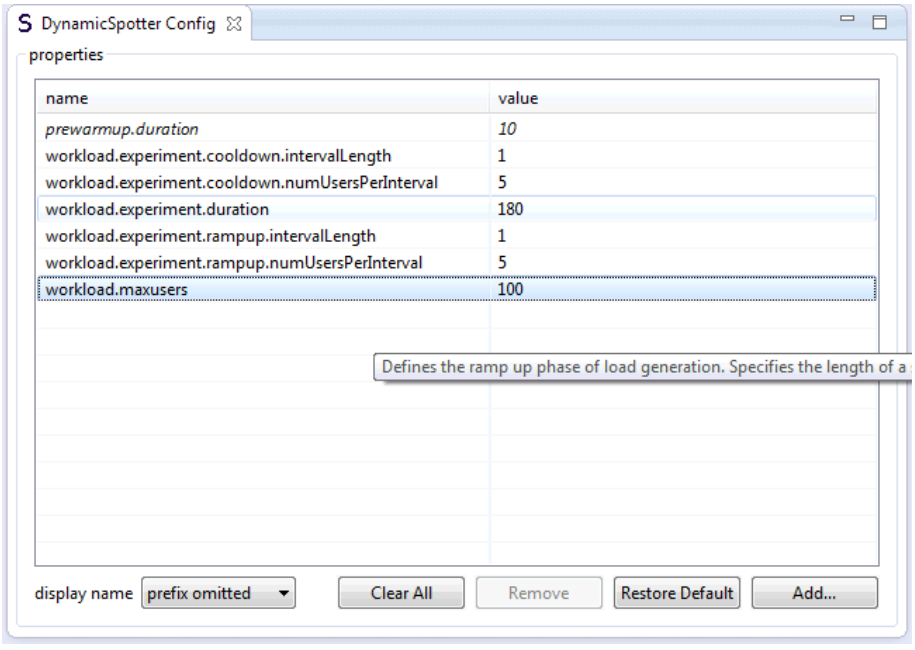


Figure 6: DynamicSpotter Config Editor

properties like experiment duration, maximal expected number of users, etc. The tool tip on each property shows the description of that property. DynamicSpotter requires six parameters specifying the workload to be mandatory configured for a certain scenario (parameters starting with `workload.*`). Additionally to the mandatory properties, one can configure optional properties. Therefore, add an optional property by clicking on the `Add...` button at the bottom right, and select a parame-

ter you would like to edit. For our demo scenario we use a configuration as depicted in Figure 6: We set the duration of the initial warm-up phase (`prewarmup.duration`) of the system under test to 10 seconds. Each experiment should be executed for 180 seconds (`workload.experiment.duration`), and for the ramp-up and cool-down behaviour of each experiment we define a user entry rate of that 5 users per second. Furthermore, we expect a maximum number of 100 users for our demo scenario (`workload.maxusers`). Finally, we save that configuration.

As next step we specify the measurement environment. As you may remember, in Section 4.3 we started the demo application with an instrumentation agent. That agent covers the functionality of two DynamicSpotter Satellite Adapter types: it is responsible for instrumenting the target application (`Instrumentation Satellite Adapter`) and in order to retrieve the measurement data generated by the instrumentation code that agent provides an interface for a `Measurement Satellite Adapter`. Furthermore, besides that two adapters we will need to configure a `Workload Satellite Adapter` using the load script described in Section 4.4.

We start with specifying the instrumentation adapter. Therefore, we need to open the editor for `Instrumentation Satellite Adapters`. There, we add an instrumentation adapter of type `instrumentation.satellite.adapter.default`(`Add...` button on the upper part of the editor). We then see a instrumentation adapter entry in the list with a red cross (cf. Figure 7). The red

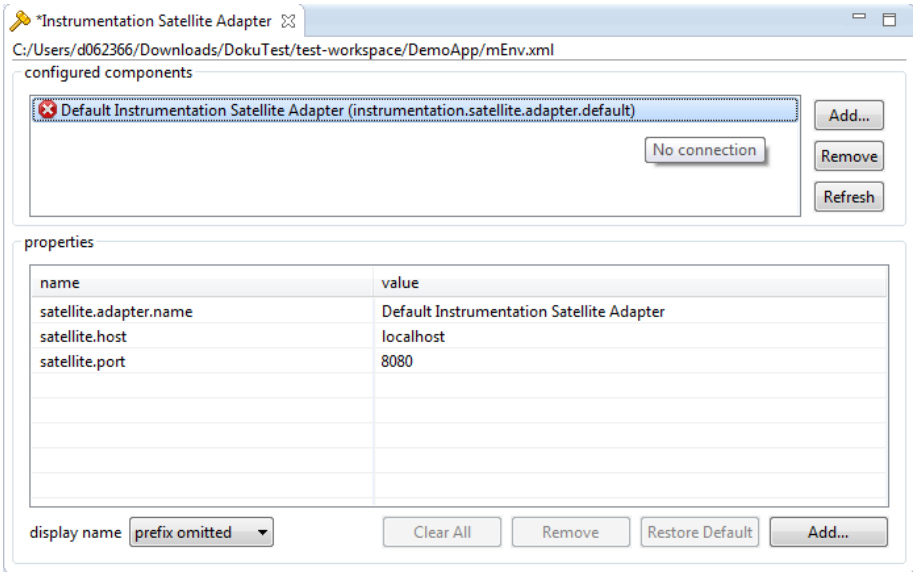


Figure 7: DynamicSpotter Instrumentation Satellite Adapter Editor

cross means that no connection to the satellite could be established. As mentioned in Section 4.3, the instrumentation agent starts per default its REST service on port `8888`. Thus, we need to change the port in the properties view of the instrumentation adapter (`satellite.port`) from `8080` to `8888`. This should turn the red cross to a green check mark. Again, we need to save the configuration. Analogously to the configuration of the instrumentation adapter we have to create a measurement adapter in the `Measurement Satellite Adapter` editor. Thereby, we need a measurement adapter of type `measurement.satellite.adapter.instrumentation` in order to connect to the measurement interface of the instrumentation agent.

Next, we have to configure a workload adapter. Therefore, we open the `Workload Satellite Adapter Editor` and add a satellite of the type `workload.satellite.adapter.customized` (cf. Figure 8). This type of workload adapter is a simple load generator which runs within the DynamicSpotter process. Therefore, this adapter does not have properties like host or port and is per default connected to DynamicSpotter (green check mark). The customized workload adapter uses a Java class extending the interface `ISimpleVUser` (cf. Section 4.4) as a load script for a virtual user. Thus, we need to specify the full qualified class name of that load script class (`workload.simple.userScriptClassName`) and the path (`workload.simple.userScriptPath`) to the directory containing the Java package structure (the `loadScript` directory of the all-in-one-drop

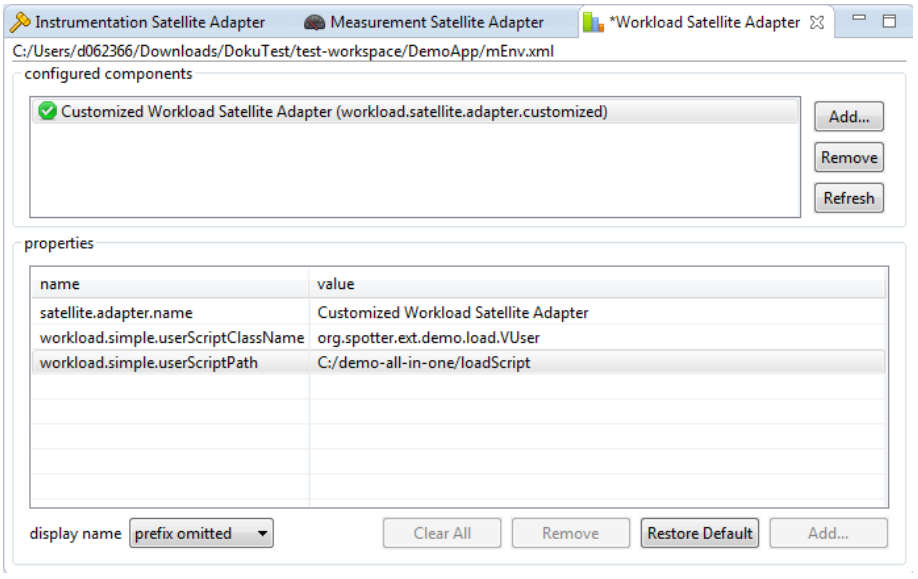
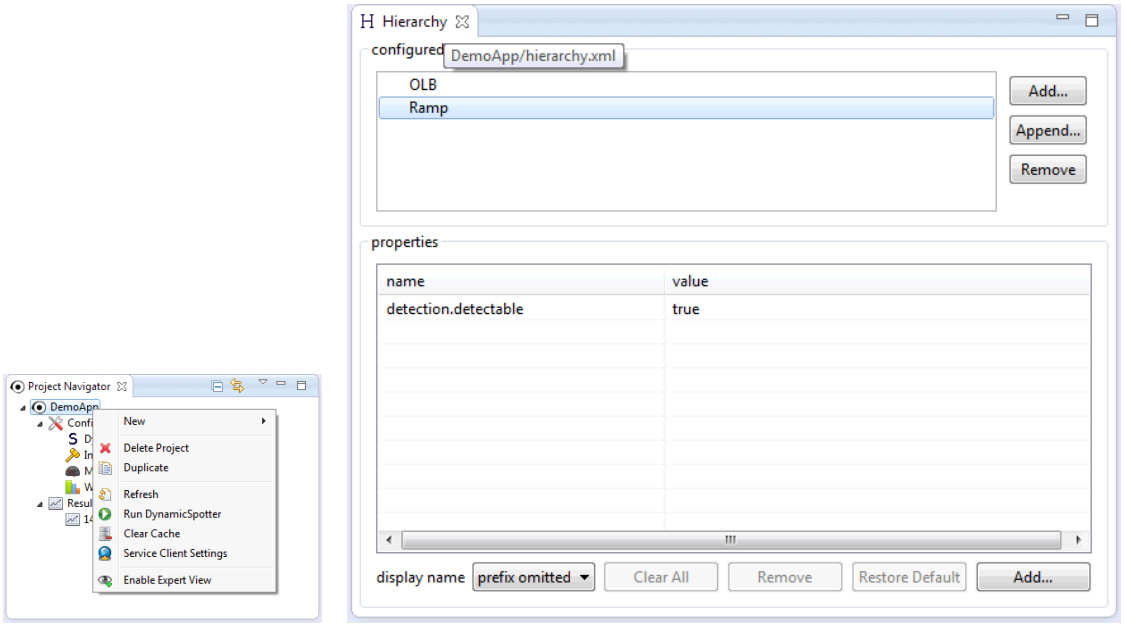


Figure 8: Configuration of the Workload Adapter

archive).

Finally, we have to tell DynamicSpotter which performance anti-patterns should be analyzed in the target application. Configuring the performance problem hierarchy is a task for performance experts. Therefore, we first have to activate the expert view by selecting `Enable Expert View` in the context menu of our DynamicSpotter Project (cf. Figure 9(a)). Now, an additional node



(a) Enabling Expert View

(b) Hierarchy Editor

Figure 9: Editing the Performance Problem Hierarchy

called `Hierarchy` should appear in the project navigator. Double click on that node to edit the performance problem hierarchy. For sake of simplicity, in our demo scenario we consider only two performance anti-patterns to be analyzed: the One Lane Bridge (OLB) and the Ramp anti-pattern. Add (flat, not appending) both anti-patterns to the hierarchy and save the changes (cf. Figure 9(b)).

Having configured the global properties and specified the measurement environment, our DynamicSpotter Project is ready to be executed. Therefore select the `DemoApp` project in the DynamicSpotter Project Navigator and click on the `Run` button (or `Run DynamicSpotter` in the context menu of the Project Navigator). In general, depending on the complexity of the scenario the execution of DynamicSpotter may take hours. The execution of our demo scenario may take up to 30 minutes.

Analyzing DynamicSpotter Results

The results of a run are stored in the project's `results` folder. For each run a separate folder is created. Inside this folder all relevant data of the run are stored, e.g. also additional resources that were generated. Each of these folders are represented as a sub-node under the *Results* node in the DynamicSpotter project navigator.

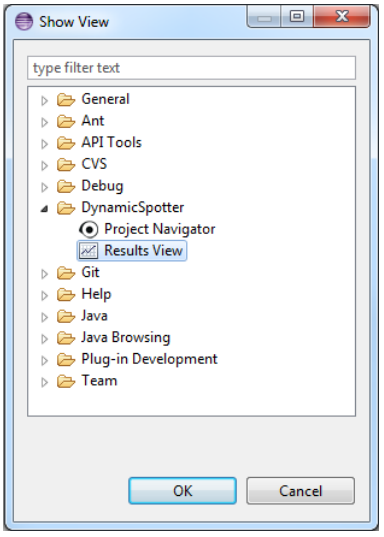


Figure 10: Show Results View

If the *Results View* has not been opened yet, it can be opened via `Window` → `Show View` → `Other...` under the category *DynamicSpotter* (cf. Figure 10), or simply by double-clicking on any of the results sub-nodes.

The *Results View* displays the content corresponding to the selected run node in the navigator (cf. Figure 11). The view contains two tabs, one showing the hierarchy and one showing the report text. In the upper part of the hierarchy tab there is the hierarchy tree that was used during the diagnosis. Each detected problem is marked with a red flag. Problems that have not been detected are marked with a green flag. An exclamation mark signals that there has been an error during the diagnosis of this problem.

In the lower part the details of the currently selected problem including its related resources are shown. If resources are available, clicking on one of the items in the list will show a little preview next to it which can be enlarged by clicking on the preview image.

In our demo example, the Ramp anti-pattern has not been detected, however, DynamicSpotter detected a One Lane Bridge (OLB) in method `org.spotter.ext.demo.app.DummyApp.testOLB()` (cf. Figure 11). The additional resources (graphs) show that the response times grow steadily with the load while the CPU utilization does not increase significantly.

4.6 Executing DynamicSpotter from Command Line

When using the Eclipse-Plugin to configure a DynamicSpotter Project, the Eclipse-Plugin creates configuration files for DynamicSpotter in the background (which you can find in the project directory in your Eclipse workspace). In order to execute DynamicSpotter from the command line, DynamicSpotter configuration files need to be passed to DynamicSpotter directly instead of

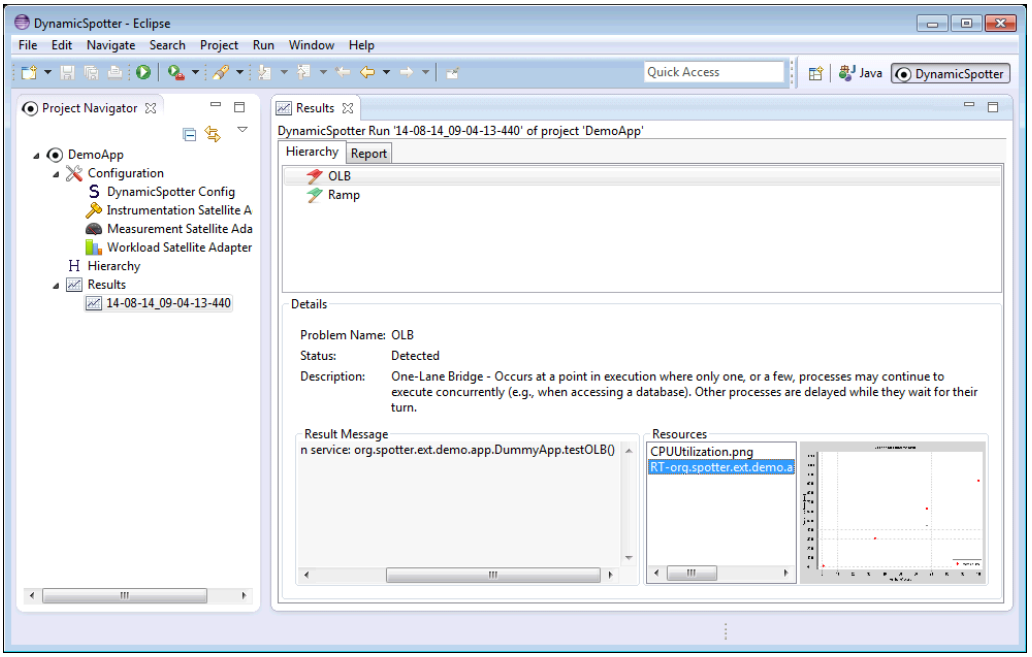


Figure 11: Results View

executing the steps from Section 4.5. First, adapt the paths in the `demo-config/spotter.conf` of the all-in-one-drop. In particular, set the paths to the hierarchy XML, the measurement environment XML and the result directory:

```
# path to the XML file describing the problem hierarchy
org.spotter.conf.problemHierarchyFile = ...

# path to the XML file describing all measurement satellites
# and their configurations
org.spotter.measurement.environmentDescriptionFile = ...

# path to the directory containing the results
org.spotter.resultDir = ...
```

Furthermore, you need to adapt the path to the workload file in the `demo-config/spotter.conf`. Therefore, for the property `org.spotter.workload.simple.userScriptPath` set the value to the *absolute path* of the `all-in-one-drop/loadScript` directory.

```
...
<workloadAdapter>

    <extensionName>workload.satellite.adapter.customized</extensionName>

    <config key="org.spotter.satellite.adapter.name"
      value="Customized␣Workload␣Satellite␣Adapter"/>

    <config key="org.spotter.workload.simple.userScriptClassName"
      value="org.spotter.ext.demo.load.VUser"/>

    <config key="org.spotter.workload.simple.userScriptPath"
      value="..."/>

</workloadAdapter>
```

```
...
```

Having adapted the configuration files use the following command to start DynamicSpotter from the command line:

```
java -jar ds-cl-runner.jar <PATH_TO_SPOTTER_CONF>
```

Thereby, the `<PATH_TO_SPOTTER_CONF>` points to the DynamicSpotter configuration as provided with the all-in-one-drop (`demo-config/spotter.conf`). During execution, DynamicSpotter writes the measurement and detection results to the specified result directory.

5 Building DynamicSpotter

In order to build the DynamicSpotter framework, the following tools need to be installed:

- JDK (tested with JDK 1.7)
  http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html
- Git client
  http://git-scm.com/downloads
- Maven 3
  http://maven.apache.org/download.cgi

In Section 7, we provide some useful Web links including the link to the DynamicSpotter framework repository. Use a Git client to clone the repository from the DynamicSpotter GitHub repository:

https://github.com/sopeco/DynamicSpotter/tree/ver-1.0.0

The DynamicSpotter framework comprises seven sub-projects:

Project Name	**Description**
org.spotter.parent	Maven parent project for the DynamicSpotter framework
org.spotter.core	Comprises the core of the DynamicSpotter framework
org.spotter.runner	Wraps a command line executor around DynamicSpotter core
org.spotter.service	Wraps a Web server around DynamicSpotter core extending it with a REST service layer.
org.spotter.client	The DynamicSpotter client is used to conveniently consume the DynamicSpotter REST service from other applications using the org.spotter.client library.
org.spotter.shared	Comprises classes and artifacts shared between DynamicSpotter core, DynamicSpotter runners and DynamicSpotter client
org.spotter.eclipse.ui	The Eclipse-Plugin provides a graphical user interface for configuring, executing DynamicSpotter and analyzing its detection results. This project uses the DynamicSpotter client to communicate with the DynamicSpotter REST service.

Table 1: DynamicSpotter Project Structure

In order to build dynamic spotter use the command line and switch to the `org.spotter.parent` directory. There, execute the following Maven build command:

```
mvn clean install
```

If executed successfully, Maven creates in the `target` directories of the individual sub-projects the corresponding JARs. The executable JAR for the DynamicSpotter Service is located in the `org.spotter.service/target` directory.

The Eclipse Plugin JAR is located in the `org.spotter.eclipse.ui/target` directory.

6 Writing Extensions for DynamicSpotter

As mentioned in Section 2, DynamicSpotter provides four different types of extensions which are required to run DynamicSpotter . These types are explained in Section 2. A collection of some extensions for DynamicSpotter is available under the following GitHub repository:

https://github.com/sopeco/DynamicSpotter-Extensions

In this section, we explain how to write different types of extensions for DynamicSpotter .

6.1 General Structure of a DynamicSpotter Extension

DynamicSpotter extensions can be either bundled in separate JARs or in aggregated JARs each containing several extensions. A single extension at least comprises a Java class providing meta-information about the extension (with typical suffix `___Extension`), a Java class representing the actual extension artifact, and an entry in a text file declaring the extension. The extension class has to implement at least three methods (cf. Listing 3).

Listing 3: Methods to be implemented by an Extension Class

```
...

/**
 * Returns the name of the extension which is expected
 * to be unique in the framework.
 *
 * The name is expected to be specific to the extension
 * that is provided.
 */
String getName();

/**
 * Returns a set of configuration parameter descriptions.
 */
Set<ConfigParameterDescription> getConfigParameters();

/**
 * Creates a new artifact for this extension.
 */
EA createExtensionArtifact();

...
```

The `getName` method specifies the name of the extensions, the `getConfigParameters` method allows to define a set of extension-specific configuration parameters, and the `createExtensionsArtifacts` method creates the actual extension artifact. The extension artifact needs to pass an instance of the extension class as the extension provider to the super constructor (cf. Listing 4).

Listing 4: A constructor of an Extension Artifact

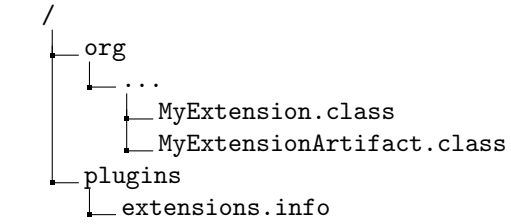
```
/**
 * Constructor.
 */
public ExtArtifact(IExtension<IDetectionController> provider) {
   super(provider);
}
```

Finally, we have to declare the extension in the `extensions.info` file. The `extensions.info` file contains per extension a line with the full qualified name of the corresponding extension (provider) class (cf. Listing 5).

Listing 5: Declaring an extension in the `extensions.info` file

```
...
full.qualified.name.of.ExtensionClass
...
```

The `extensions.info` file needs to be located in a directory called `plugins`. Assuming that we bundle each extension in a separate JAR, despite for additional dependencies the JAR structure would look like as follows:



In the following, we describe for each extension type what needs to implemented to provide an extension for the corresponding type.

6.2 Writing an Instrumentation Extension

As explained in Section 6.1, for an extension we need an *extension artifact class* implementing the functionality of the extension type and an *extension class* providing meta information on the extension. Except for the class to inherit, the extension class has the same structure for the extension types *Instrumentation, Measurement and Load Generation* (cf. Listing 6).

Listing 6: Extension class for an instrumentation extension

```
public class MyInstExtension extends AbstractInstrumentationExtension {

    @Override
    public String getName() {
        return "MyInstrumentation";
    }

    @Override
    protected void initializeConfigurationParameters() {
        addConfigParameter(
            ConfigParameterDescription.createExtensionDescription(
                "This text provides a description on the extension."));

        ConfigParameterDescription aConfigParameter
            = new ConfigParameterDescription(
                "parameterName", LpeSupportedTypes.Long);
        aConfigParameter.setDefaultValue(10);
        aConfigParameter.setDescription(
            "This text describes the configuration parameter.");
        addConfigParameter(aConfigParameter);
    }

    @Override
    public boolean isRemoteExtension() {
        return true;
    }
```

```
    @Override
    public boolean testConnection(String host, String port) {
        // test connection to remote satellite
    }

    @Override
    public IInstrumentationAdapter createExtensionArtifact() {
        return new MyInstExtensionArtifact(this);
    }
}
```

An instrumentation extension needs to inherit the class `AbstractInstrumentationExtension`. `getName` returns the unique name of the extension. In the `initializeConfigurationParameters` method we can provide a textual description of the extension as well as a set of extension-specific configuration parameters. The `isRemoteExtension` method indicates whether the corresponding extension is an adapter to a remote satellite. If that is the case, DynamicSpotter later uses the `testConnection` method to check whether a connection to the remote satellite can be established. Finally, the method `createExtensionArtifact` creates a new instance of the actual instrumentation extension artifact (here: `MyInstExtensionArtifact`). In this example the class `MyInstExtensionArtifact` implements the actual instrumentation adapter, and as such has to inherit the class `AbstractInstrumentationAdapter` (cf. Listing 7).

Listing 7: Extension Artifact class for an instrumentation extension

```
public class MyInstExtensionArtifact
    extends AbstractInstrumentationAdapter {

    public MyInstExtensionArtifact(IExtension<?> provider) {
        super(provider);
    }

    @Override
    public void initialize() throws InstrumentationException {
        // initialization of the instrumentation adapter
    }

    @Override
    public void instrument(InstrumentationDescription description)
        throws InstrumentationException {
        // instrument the target application as specified
        // in the instrumentation description
    }

    @Override
    public void uninstrument() throws InstrumentationException {
        // revert instrumentation
    }

}
```

An instrumentation extension artifact has to implement three methods: `initialize`, `instrument` and `uninstrument`. The `initialize` method takes care of conducting initialization tasks required before the specific instrumentation tool can be used. The `instrument` method triggers the instrumentation, realizing the instrumentation instructions wrapped by the instrumentation description. Finally, the `uninstrument` method reverts the instrumentation. Depending on the tool used for instrumentation the implementation of this instrumentation adapter may look completely

different. For instance, if we are using our own instrumentation tool (Adaptable Instrumentation and Monitoring (AIM)), the adapter just passes the instrumentation description to the remote AIM instrumentation engine to realize the instrumentation. An instrumentation extension for Kieker or DiSL would require to translate the instrumentation description into Kieker- or DiSL-specific instrumentation instructions. Furthermore, if the instrumentation tool does not support dynamically adaptable instrumentation, the instrumentation extension would need to take care of restarting the target application to enable adaptation of the instrumentation state. Regardless of which instrumentation tool is used behind an instrumentation extension, DynamicSpotter only assumes that after calling the `instrument` method, the target application is instrumented accordingly. Analogously for the `uninstrument` method.

6.3 Writing a Measurement Extension

A measurement extension class has to inherit the class `AbstractMeasurmentExtension`. Apart from that, the extension class of a measurement extension has the same structure as shown in Listing 6. A measurement extension artifact has to inherit the class `AbstractMeasurmentAdapter` exhibiting a structure as depicted in Listing 8.

Listing 8: Extension Artifact class for a measurement extension

```
public class MyMeasurementExtensionArtifact
   extends AbstractMeasurementAdapter {

   public MyMeasurementExtensionArtifact(IExtension<?> provider) {
      super(provider);
   }

   @Override
   public void initialize()
      throws MeasurementException {
      // initialization
   }

   @Override
   public void enableMonitoring()
      throws MeasurementException {
      // activate data gathering / resets previously gathered data
   }

   @Override
   public void disableMonitoring()
      throws MeasurementException {
      // deactivate data gathering
   }

   @Override
   public MeasurementData getMeasurementData()
      throws MeasurementException {
      // collect and return measured data
   }

   @Override
   public void pipeToOutputStream(OutputStream oStream)
      throws MeasurementException {
      // pipe measured data to the given oStream
   }
```

```
    @Override
    public long getCurrentTime() {
        // returns the current timestamp of the
        // node the measurement tool is running on.
        // This is required for synchronization of
        // distributed measurement records.
    }
}
```

Analogously to the instrumentation adapter, a measurement adapter can be initialized. The `enableMonitoring` and `disableMonitoring` activate data gathering on the target system. The methods `getMeasurementData` and `pipeToOutputStream` allow to retrieve the data measured since the last activation of monitoring. While `getMeasurementData` is a blocking call, `pipeToOutputStream` allows to take advantage of data pipelining. Finally, `getCurrentTime` returns the current timestamp of the monitoring tool. This timestamp is required to align the timestamps of all records which may come from distributed system nodes. Analogously to instrumentation extensions, a measurement extension serves as an adapter. For instance, if we want to use Kieker as monitoring tool, the corresponding measurement extension has to take care of translating Kieker monitoring records into the DynamicSpotter-specific representation of measurement data. In this way, we enable the usage of diverse tools via the adapters while keeping a consistent data representation within DynamicSpotter .

6.4 Writing a Load Generation Extension

A load generation extension class has to inherit the class `AbstractWorkloadExtension`. Apart from that, the extension class of a measurement extension has the same structure as shown in Listing 6. A load generation extension artifact has to inherit the class `AbstractWorkloadAdapter` yielding a class structure as depicted in Listing 9.

Listing 9: Extension Artifact class for a load generation extension

```
public class MyWorkloadExtensionArtifact
    extends AbstractWorkloadAdapter {

    public MyWorkloadExtensionArtifact(IExtension<?> provider) {
        super(provider);
    }

    @Override
    public void initialize()
        throws WorkloadException {

        // initialization
    }

    @Override
    public void startLoad(LoadConfig loadConfig)
        throws WorkloadException {
        // asynchronously start generation of workload
    }

    @Override
    public void waitForWarmupPhaseTermination()
        throws WorkloadException {
        // blocks until the warm-up phase terminates
    }
```

```
    @Override
    public void waitForExperimentPhaseTermination()
        throws WorkloadException {
        // blocks until the stable experiment phase terminates
    }

    @Override
    public void waitForFinishedLoad() throws WorkloadException {
        // blocks until the cool down phase terminates
        // and no load is submitted to the target application anymore
    }
}
```

Same as for the instrumentation and measurement adapters, a workload adapter provides a method for initialization tasks. By calling the `startLoad` method, DynamicSpotter tells a workload adapter to start generating load *asynchronously*. The three `wait___` methods block until the corresponding experiment phase has been reached. In our extensions repository (cf. Section 7), we provide load generation adapters for Apache JMeter, HP LoadRunner, and a simple custom workload generator used in the Demo example in Section 4.3.

6.5 Writing a Detection Heuristic

In the DynamicSpotter framework the detection heuristics for individual performance problems are managed as extensions, as well. The extension class of a detection heuristic extension has to extend the class `AbstractDetectionExtension`. The three methods which need to be implemented we already know from the general structure shown in Listing 3. The class implementing the actual detection logic has to extend the `AbstractDetectionController` class. This inheritance requires the detection class to implement four methods as shown in Listing 10.

Listing 10: Detection controller class for a detection heuristic extension

```
public class MyDetection extends AbstractDetectionController {

    private int numExperimentSteps;

    public MyDetection(IExtension<IDetectionController> provider) {
        super(provider);
    }

    @Override
    public void loadProperties() {
        // load heuristic-specific configuration properties
        numExperimentSteps = Integer.parseInt(
            getProblemDetectionConfiguration().
                getProperty("myDetection.numSteps"));
    }

    @Override
    protected void executeExperiments()
        throws InstrumentationException, MeasurementException,
            WorkloadException {
        // create a problem specific instrumentation description
        InstrumentationDescription instDescr = ...;
        // trigger default experiment series
        executeDefaultExperimentSeries(MyDetection.class,
            numExperimentSteps, instDescr);
    }
```

```
    @Override
    protected SpotterResult analyze(DatasetCollection data) {
        // analyze measurement data
        // ...
        SpotterResult result = new SpotterResult();
        result.setDetected(true);
        result.addMessage("Detection run finished successfully!");
        return result;
    }

    @Override
    public long getExperimentSeriesDuration() {
        // estimate duration of the experiment series to execute
    }
}
```

The `loadProperties` method is called by DynamicSpotter directly after the instantiation of the detection class. This method can be used to load heuristic-specific properties as defined in the `getConfigParameters` method of the extension class (cf. Listing 3). The detection process of an individual heuristic comprises two phases: an experiment execution phase and an data analysis phase. While the former is triggered by the `executeExperiments` method, the `analyze` method is responsible for analysing the measured data and return a detection result. For the experiment phase, a detection heuristic has to specify a problem-specific instrumentation of the target application by providing an instrumentation description. (Note: this instrumentation description should be system independent in order to be reused in other systems as well.) Using the instrumentation description the detection heuristic can tell the DynamicSpotter to execute a series of experiments (cf. `executeDefaultExperimentSeries`) using the corresponding instrumentation, measurement and load generation adapters. `defaultExperimentSeries` means that a set of `n` experiments is executed, whereby the load is increased from one experiment to the next between a load of 1 user and the specified maximum number of users. When the experimentation phase has finished, DynamicSpotter calls the `analyze` method passing the measured data as input to that method. It is now up to the analyze method to apply certain detection rules in order to provide a detection result. A detection result has to state whether the corresponding analyzed performance problem has been detected or not. Finally, the `getExperimentSeriesDuration` method is used by DynamicSpotter to estimate the duration of the experiments. In this method, the developer of a detection heuristic has to provide the estimated duration (in seconds) of the experiments executed for that heuristic.

7 Useful Links

Link & Description
Code Repositories
https://github.com/sopeco/DynamicSpotter/tree/ver-1.0.0 DynamicSpotter framework repository
https://github.com/sopeco/DynamicSpotter-Extensions Extensions repository for DynamicSpotter
https://github.com/sopeco/LPE-Common Repository containing utility libraries for performance measurements, data analysis, load generation, etc.
Documentation
http://sopeco.github.io/DynamicSpotter Documentation on DynamicSpotter
https://github.com/sopeco/DynamicSpotter-Demo/releases/download/v1.0/demo-all-in-one.zip https://github.com/sopeco/DynamicSpotter-Demo/releases/download/v1.0/demo-all-in-one.tar.gz All-in-One Drop (as .zip or .tar.gz file) containing the demo example for DynamicSpotter

Table 2: Useful Links

Bibliography

[Apa14] Apache Software Foundation, "Apache jmeter homepage," April 2014. [Online]. Available: jmeter.apache.org

[Hew14] Hewlett-Packard Development Company, L.P., "Hp loadrunner homepage," April 2014. [Online]. Available: www.hp.com/LoadRunner

[RM07] D. Rayside and L. Mendel, "Object ownership profiling: a technique for finding and fixing memory leaks," in *ASE*. ACM, 2007, pp. 194–203.

[SW00] C. Smith and L. Williams, "Software performance antipatterns," in *WOSP*. ACM, 2000, pp. 127–136.

[SW02] ——, "Software performance antipatterns; common performance problems and their solutions," in *CMG-CONFERENCE-*, vol. 2, 2002, pp. 797–806.

[SW03a] ——, "More new software performance antipatterns: Even more ways to shoot yourself in the foot," in *CMG-CONFERENCE-*, 2003, pp. 717–725.

[SW03b] ——, "New software performance antipatterns: More ways to shoot yourself in the foot," in *CMG-CONFERENCE-*, vol. 2, 2003, pp. 667–674.

[Wer13] A. Wert, "Performance problem diagnostics by systematic experimentation," in *Proc. WCOP*. ACM, 2013, pp. 1–6.

[WHH13] A. Wert, J. Happe, and L. Happe, "Supporting swift reaction: automatically uncovering performance problems by systematic experiments," in *Proc. ICSE*. IEEE Press, 2013, pp. 552–561.

[WOHF14] A. Wert, M. Oehler, C. Heger, and R. Farahbod, "Automatic Detection of Performance Anti-patterns in Inter-component Communications," in *Proceedings of the 10th International Conference on Quality of Software Architecture*, ser. QoSA '14, 2014.