Übung 03

Arbeitsaufwand insgesamt: 17h

Inhaltsverzeichnis

Teil 1 – Server-Profiling-Dashboard

In dieser Übung geht es darum, Metriken eines Servers zu überwachen.

Lösungsidee:

Die Kernkomponenten „Components“ und „Profiler“ lassen sich direkt aus dem Text erkennen. Zusätzlich dazu wird durch das viele verwenden des Stichworts „überwachen“ die Verwendung des Observer-Patterns impliziert, weshalb ich mich auch für die Verwendung dieses Patterns entschieden habe.

Die Komponenten des Servers sind dabei die Subjekte und die Profiler die Observer. Komponenten ändern deren Metriken durchgehend und benachrichtigen jede Sekunde die zuständigen Profiler, die sich an dieser Komponente interessieren. Damit die Benachrichtigung nicht den normalen Programmablauf blockiert, würde ich jede Komponente in einem eigenen Thread laufen lassen. Um realistische Werte bereitzustellen, würde ich anfangs für jede Komponente die verfügbaren Metriken angeben (Name) und diese dann jedes Mal kurz vor der Benachrichtigung an den Profiler zufällig berechnen lassen.

Wenn der Profiler benachrichtigt wird, dann gibt er die neu erhaltenen Metriken in eine Map, speichert diese zwischen und sammelt so im Laufe des Programms immer mehr Daten an. Weil die Komponenten eigene Threads abbilden, ist es hierbei wichtig, dass synchronisierte Methoden und Maps verwendet werden. Dadurch, dass der Profiler immer die aktuellen Werte abfängt, bietet es sich hier an Warnung für Über- oder Unterschreitung gewisser Grenzwerte auszugeben. Diese wiederum könnten wieder in einer Map als Kombination „Metrikname“ und Grenzwert gespeichert werden.

Um später Daten ausgeben zu können würde ich noch eine Klasse „Protokoll“ implementieren, welche die Daten zusätzlich filtert und den Zugriff auf spezifische Zeiträume erlaubt. Somit ist es möglich, die Metriken auf eine gewisse Komponente und Zeitraum zu begrenzen. Vor der Ausgabe können noch wichtige statistische Daten wie Minimum, Maximum, Mittelwert oder Median berechnet werden. Als Ausgabeart würde ich CSV und die Konsole definieren. CSV ist insofern sinnvoll, weil dadurch leicht Diagramme erzeugt werden können.

Nachtrag:

Es wurde zudem das Singleton Muster verwendet, um „Unterprotokolle“ für jede Komponente zu definieren. Dieses Protokoll soll nur einmal existieren, da es sich nicht verändert und so sehr einfach in Sets verwaltet werden kann, da es sich immer um dasselbe Element handelt.

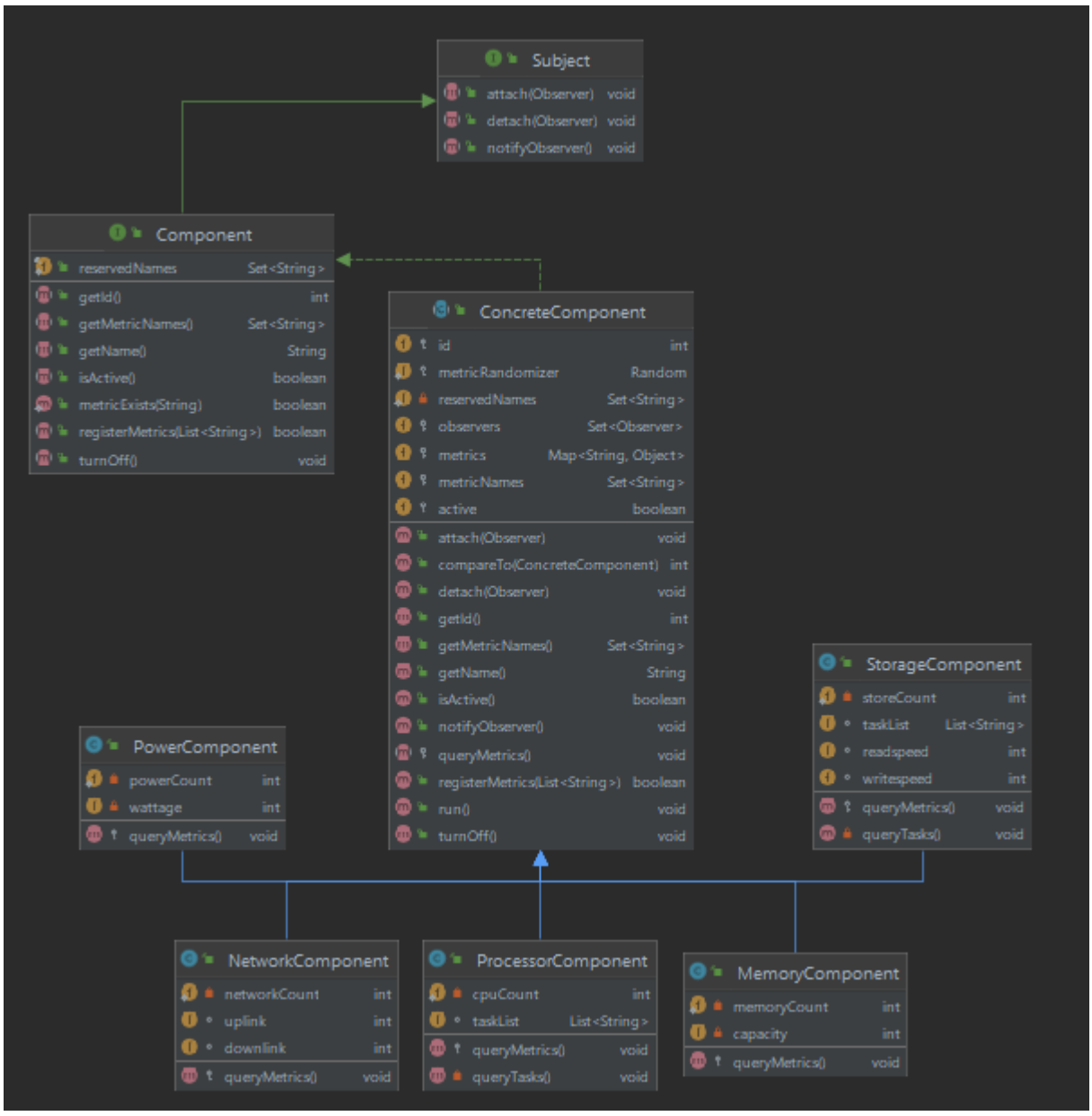
Zusätzlich dazu wurden die Datenstrukturen innerhalb der Komponenten abgeflacht, um die Datenhaltung zu erleichtern. Anstatt von verschachtelten Maps wurde jetzt eine einzige Map verwendet, deren Keys jetzt jedoch nicht mehr atomar sind. Als Beispiel die Prozessor-Komponente:

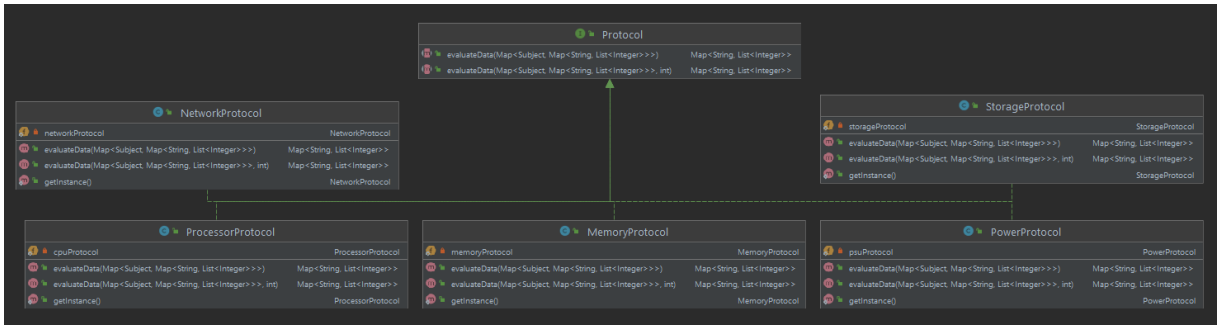
- Vorher: unterste Map hat zwei Knoten „Gesamtauslastung“ und „Per-Prozess“. „Per-Prozess“ ist wiederum eine Map und verbindet Prozessnamen mit deren Auslastung).
- Nachher: Key = „Per-Prozess;IntelliJ“, Value = 10%

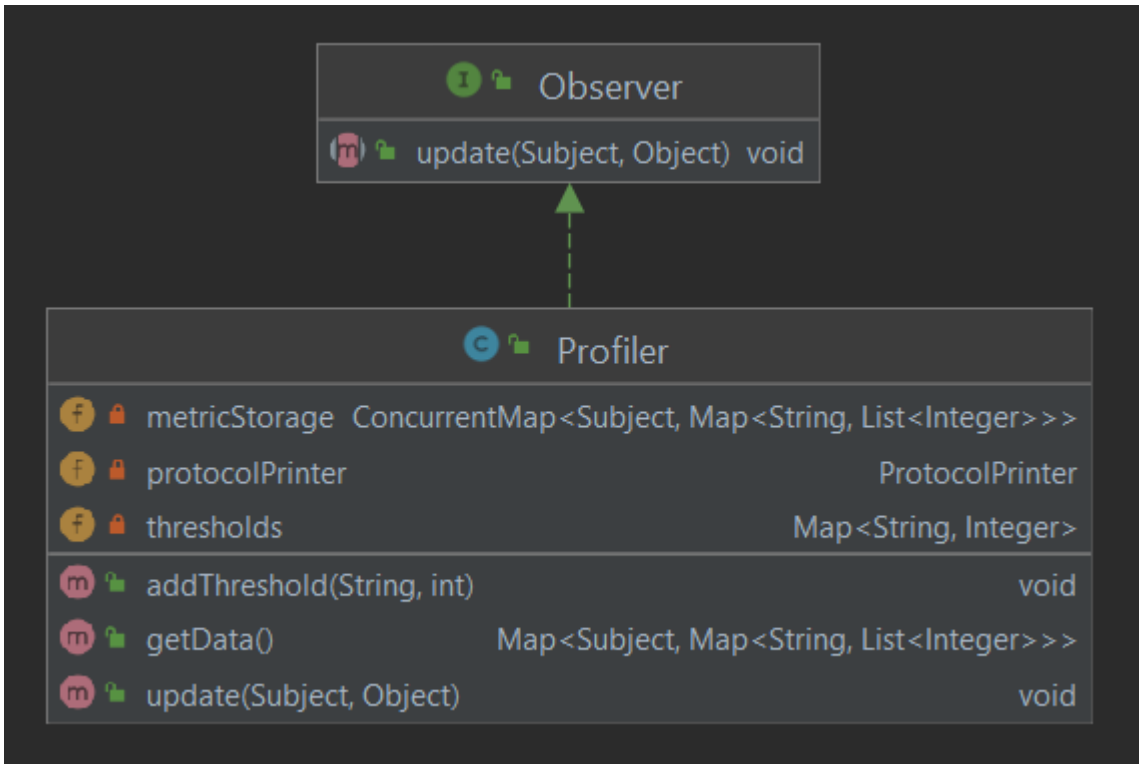
Lösung:

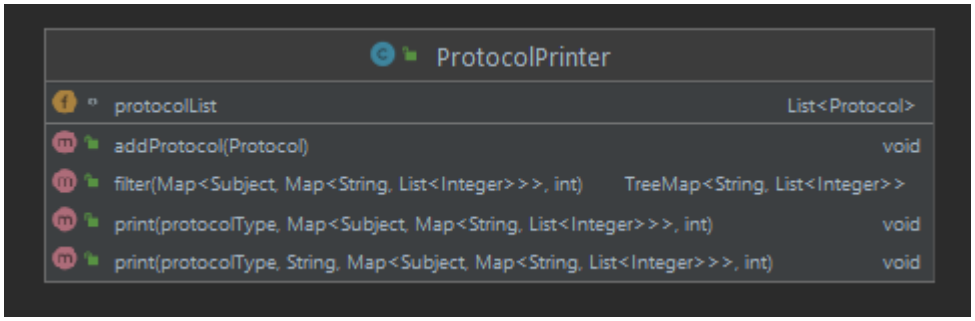
Als IDEA Projekt im Archiv
Nachfolgender Source-Code

UML


Subject
attach(Observer) void
detach(Observer) void
notifyObserver() void
Component
reservedNames Set<String>
getId() int
getMetricNames() Set<String>
getName() String
isActive() boolean
metricExists(String) boolean
registerMetrics(List<String>) boolean
turnOff() void
ConcreteComponent
id int
metricRandomizer Random
reservedNames Set<String>
observers Set<Observer>
metrics Map<String, Object>
metricNames Set<String>
active boolean
attach(Observer) void
compareTo(ConcreteComponent) int
detach(Observer) void
getId() int
getMetricNames() Set<String>
getName() String
isActive() boolean
notifyObserver() void
queryMetrics() void
registerMetrics(List<String>) boolean
run() void
turnOff() void
StorageComponent
storeCount int
taskList List<String>
readspeed int
writespeed int
queryMetrics() void
queryTasks() void
PowerComponent
powerCount int
wattage int
queryMetrics() void
NetworkComponent
networkCount int
uplink int
downlink int
queryMetrics() void
ProcessorComponent
cpuCount int
taskList List<String>
queryMetrics() void
queryTasks() void
MemoryComponent
memoryCount int
capacity int
queryMetrics() void


Protocol
NetworkProtocol
StorageProtocol
ProcessorProtocol
MemoryProtocol
PowerProtocol


Observer
update(Subject, Object) void
Profiler
metricStorage ConcurrentMap<Subject, Map<String, List<Integer>>>
protocolPrinter ProtocolPrinter
thresholds Map<String, Integer>
addThreshold(String, int) void
getData() Map<Subject, Map<String, List<Integer>>>
update(Subject, Object) void


ProtocolPrinter
protocolList List<Protocol>
addProtocol(Protocol) void
filter(Map<Subject, Map<String, List<Integer>>>, int) TreeMap<String, List<Integer>>
print(protocolType, Map<Subject, Map<String, List<Integer>>>, int) void
print(protocolType, String, Map<Subject, Map<String, List<Integer>>>, int) void

Source-Code:

DashboardTest.java

```
package swp4.ue03.test;

import swp4.ue03.components.impl.*;
import swp4.ue03.profiling.impl.Profiler;
import swp4.ue03.protocol.Protocol;
import swp4.ue03.protocol.impl.*;

public class DashboardTest {

    private static void test_example() {
        // Create all the components with customizable
        MemoryComponent ramComp = new MemoryComponent(32);
        NetworkComponent netComp = new NetworkComponent(150,40);
        ProcessorComponent cpuComp = new ProcessorComponent();
        PowerComponent psuComp = new PowerComponent(750);
        StorageComponent storeComp = new StorageComponent(250, 80);

        // Create and configure the profilers
        Profiler p1 = new Profiler();
        Profiler p2 = new Profiler();
        p2.addThreshold("cpu-utilization",90);

        // Tell the profilers which components to monitor.
        ramComp.attach(p1);
        netComp.attach(p1);
        psuComp.attach(p1);

        storeComp.attach(p2);
        cpuComp.attach(p2);


        // "Start" all the components
        Thread ramThread = new Thread(ramComp);
        Thread netThread = new Thread(netComp);
        Thread psuThread = new Thread(psuComp);
        Thread cpuThread = new Thread(cpuComp);
        Thread storeThread = new Thread(storeComp);
        ramThread.start();
        netThread.start();
        psuThread.start();
        cpuThread.start();
        storeThread.start();

        ProcessorComponent cpuComp2 = new ProcessorComponent();
        cpuComp2.attach(p2);
        Thread cpuThread2 = new Thread(cpuComp2);
        cpuThread2.start();

        // wait 10 seconds for generation of data
        try {
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
```

```
        // Print protocols to console and CSV
        ProtocolPrinter pp1 = new ProtocolPrinter();
        pp1.addProtocol(MemoryProtocol.getInstance());
        pp1.addProtocol(NetworkProtocol.getInstance());
        pp1.addProtocol(PowerProtocol.getInstance());

        ProtocolPrinter pp2 = new ProtocolPrinter();
        pp2.addProtocol(ProcessorProtocol.getInstance());
        pp2.addProtocol(StorageProtocol.getInstance());
        pp1.print(ProtocolPrinter.protocolType.NUMBER, p1.getData(), 5);
        pp2.print(ProtocolPrinter.protocolType.CSV, "cpu_info.csv",
p2.getData(), 10 );

        // shut down all the components (and their threads)
        ramComp.turnOff();
        netComp.turnOff();
        psuComp.turnOff();
        cpuComp.turnOff();
        storeComp.turnOff();

        cpuComp2.turnOff();
    }
    private static void test_unstarted() {
        ProcessorComponent cpuComp = new ProcessorComponent();
        Profiler p = new Profiler();
        cpuComp.attach(p);

        ProtocolPrinter pp = new ProtocolPrinter();
        pp.addProtocol(ProcessorProtocol.getInstance());

        pp.print(ProtocolPrinter.protocolType.NUMBER, p.getData(), 10);
    }
    private static void test_no_subprotocols() {
        ProcessorComponent cpuComp = new ProcessorComponent();
        Profiler p = new Profiler();
        cpuComp.attach(p);
        Thread cpuT = new Thread(cpuComp);
        cpuT.start();

        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // protocolprinter without specified protocols
        ProtocolPrinter pp = new ProtocolPrinter();

        pp.print(ProtocolPrinter.protocolType.NUMBER, p.getData(), 10);
        cpuComp.turnOff();
    }

    public static void main(String[] args) {
        test_example();
        //test_unstarted();
        //test_no_subprotocols();
    }
}
```

Component.java

```java
package swp4.ue03.components;

import swp4.ue03.profiling.Subject;

import java.util.*;

public interface Component extends Subject, Runnable {

    // saves all metrics for all components
    Set<String> reservedNames = new HashSet<>();

    // every component has an ID and a name which they can be identified with
    int getId();
    String getName();

    // every component has an activity status and can be turned off to stop
data generation
    void turnOff();
    boolean isActive();

    // if users want to find out which metrics
    Set<String> getMetricNames();

    // When writing new components, they have to register their metrics to
avoid duplicates between components
    boolean registerMetrics(List<String> names);

    // check if a metric exists
    static boolean metricExists(String name) {
        return reservedNames.contains(name);
    }
}
```

ConcreteComponent.java

```java
package swp4.ue03.components;

import swp4.ue03.profiling.Observer;

import java.util.*;

public abstract class ConcreteComponent implements Component,
Comparable<ConcreteComponent> {

    protected int id = 0;
    // Avoid having own Random generator in each component
    protected static Random metricRandomizer = new Random();
    private static Set<String> reservedNames = new HashSet<>();

    // save all observers for each component
    protected Set<Observer> observers = new HashSet<>();

    // a list of all metrics of a component and their current value
    protected Map<String, Object> metrics = new HashMap<>();
    protected Set<String> metricNames = new HashSet<>();
    protected boolean active = false;

    // Updates all values of the current components metrics
    protected abstract void queryMetrics();

    @Override
    public void turnOff() {
        active = false;
    }

    @Override
    public int getId() {
        return id;
    }

    @Override
    public String getName() {
        String[] classPath = this.getClass().getName().split("\\.");
        return classPath[classPath.length-1]+id;
    }

    @Override
    public boolean isActive() {
        return active;
    }

    @Override
    public Set<String> getMetricNames() {
        return metrics.keySet();
    }
```

```
    @Override
    public boolean registerMetrics(List<String> names) {
        for(String name : names) {
            if (!Component.metricExists(name)) {
                metrics.put(name, 0);
                reservedNames.add(name);
            } else {
                System.err.println("Failed to initialize" + this + ".");
                metrics.clear();
                return false;
            }
        }
        return true;
    }

    // Notify all observers with the current values
    @Override
    public void notifyObserver() {
        for(Observer obs : observers) {
            obs.update(this, metrics);
        }
    }
    @Override
    public void run() {
        while(active) {
            try {
                // generate new values and send them to the observers every 1
seconds
                queryMetrics();
                notifyObserver();
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    @Override
    public void attach(Observer observer) {
        observers.add(observer);
    }

    @Override
    public void detach(Observer observer) {
        observers.remove(observer);
    }

    // Comparable in order to use TreeMaps for sorting later
    @Override
    public int compareTo(ConcreteComponent o) {
        return this.getName().compareTo(o.getName());
    }
}
```

MemoryComponent.java

```java
package swp4.ue03.components.impl;

import swp4.ue03.components.ConcreteComponent;
import java.util.Arrays;

public class MemoryComponent extends ConcreteComponent {

    private static int memoryCount = 0;
    private int capacity;

    public MemoryComponent(int capacity) {
        // A memory component has two metrics used and free (in megabytes)
        if (registerMetrics(Arrays.asList("used","free"))) {
            this.capacity = capacity * 1024;
            active = true;
            id = ++memoryCount;
        }
    }

    protected void queryMetrics() {
        if(isActive()) {
            metrics.put("used", metricRandomizer.nextInt(capacity + 1));
            metrics.put("free", capacity - (int) metrics.get("used"));
        }
    }

}
```

NetworkComponent.java

```java
package swp4.ue03.components.impl;

import swp4.ue03.components.Component;
import swp4.ue03.components.ConcreteComponent;
import java.util.ArrayList;
import java.util.Arrays;

public class NetworkComponent extends ConcreteComponent {
    private static int networkCount = 0;
    int uplink = 0;
    int downlink = 0;

    // network component has downlink and uplink in mbits
    public NetworkComponent(int downlink, int uplink) {
        if (registerMetrics(Arrays.asList("received","sent"))) {
            active = true;
            id = ++networkCount;
            this.downlink = (int)(downlink / 8.0 * 1024 * 1024);
            this.uplink = (int)(uplink / 8.0 * 1024 * 1024);
        }
    }
    protected void queryMetrics() {
        if(isActive()) {
            metrics.put("received", metricRandomizer.nextInt(downlink + 1));
            metrics.put("sent", metricRandomizer.nextInt(uplink+1));
        }
    }
}
```

PowerComponent.java

```
package swp4.ue03.components.impl;

import swp4.ue03.components.ConcreteComponent;
import java.util.ArrayList;
import java.util.Arrays;

public class PowerComponent extends ConcreteComponent {
    private static int powerCount = 0;
    private int wattage = 0;

    // power component has wattage in watts
    public PowerComponent(int wattage) {
        if (registerMetrics(Arrays.asList("powerconsumption"))) {
            active = true;
            id = ++powerCount;
            this.wattage = wattage;
        }
    }

    protected void queryMetrics() {
        if(isActive()) {
            metrics.put("powerconsumption",
metricRandomizer.nextInt(wattage+1));
        }
    }
}
```

ProcessorComponent.java

```
package swp4.ue03.components.impl;

import swp4.ue03.components.ConcreteComponent;

import java.util.*;

public class ProcessorComponent extends ConcreteComponent {

    private static int cpuCount = 0;
    List<String> taskList = new
ArrayList(Arrays.asList("FileZilla","OpenSSH","apache2","TensorFlow","suspicio
usCryptoMiner.exe.gz"));

    // a processor has a main utilization as well as subtasks and their
utilization
    public ProcessorComponent() {
        if (registerMetrics(Arrays.asList("cpu-utilization"))) {
            active = true;
            id = ++cpuCount;
        }
    }

    // Query all processes, so they are randomized and add up to the total
utilization
    private void queryTasks() {
        int i = 0;
        int sum = 0;
        for(String task : taskList) {
            int value = metricRandomizer.nextInt(((int)metrics.get("cpu-
utilization") - sum) / (taskList.size() - i++)) + 1;
            metrics.put("cpu-processes;"+task,value);
            sum += value;
        }
    }

    protected void queryMetrics() {
        if(isActive()) {
            int total = metricRandomizer.nextInt(51) + 50;
            metrics.put("cpu-utilization", total);
            queryTasks();
        }
    }
}
```

StorageComponent.java

```
package swp4.ue03.components.impl;

import swp4.ue03.components.ConcreteComponent;

import java.util.*;

public class StorageComponent extends ConcreteComponent {
    private static int storeCount = 0;
    List<String> taskList = new
ArrayList(Arrays.asList("FileZilla","OpenSSH","apache2","TensorFlow","suspicio
usCryptoMiner.exe.gz"));

    int readspeed = 0;
    int writespeed = 0;

    // speeds in MB/s
    public StorageComponent(int readspeed, int writespeed) {
        if (registerMetrics(Arrays.asList("storage-access"))) {
            active = true;
            id = ++storeCount;
            this.readspeed = readspeed * 1024 * 1024;
            this.writespeed = writespeed * 1024 * 1024;
        }
    }

    private void queryTasks() {
        int i = 0;
        int sum = 0;
        for(String task : taskList) {
            metrics.put("storage-processes;"+task+";read",
metricRandomizer.nextInt(readspeed+1));
            metrics.put("storage-processes;"+task+";write",
metricRandomizer.nextInt(writespeed+1));
        }
    }

    protected void queryMetrics() {
        if(isActive()) {
            int total = metricRandomizer.nextInt(5000);
            metrics.put("storage-access", total);
            queryTasks();
        }
    }
}
```

Subject.java

```java
package swp4.ue03.profiling;

public interface Subject {

    void attach(Observer observer);
    void detach(Observer observer);
    void notifyObserver();
}
```

Observer.java

```java
package swp4.ue03.profiling;

public interface Observer {

    void update(Subject source, Object argument);

}
```

Profiler.java

```java
package swp4.ue03.profiling.impl;

import swp4.ue03.profiling.Observer;
import swp4.ue03.profiling.Subject;
import swp4.ue03.protocol.impl.ProtocolPrinter;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.ConcurrentMap;

public class Profiler implements Observer {

    // Profilers save EVERYTHING the components send them.
    private ConcurrentMap<Subject, Map<String, List<Integer>>> metricStorage =
new ConcurrentHashMap<>();
    //  It is the ProtocolPrinters job to filter the data and print it.
    private ProtocolPrinter protocolPrinter = new ProtocolPrinter();
    // The user can set thresholds for specific metrics, so warnings will be
sent
    private Map<String, Integer> thresholds = new ConcurrentHashMap<>();

    public Map<Subject, Map<String, List<Integer>>> getData() {
        return metricStorage;
    }

    public void addThreshold(String name, int thresh) {
        thresholds.put(name,thresh);
    }

    // Everytime the Profiler receives an update, he runs through all metrics
and adds them to the metricStorage
    @Override
```

```
    public synchronized void update(Subject source, Object argument) {
        if(argument != null && source != null) {
            // if the profiler has never received something from this
component, add it to the map
            metricStorage.computeIfAbsent(source, tmp -> new HashMap<>());

            // go through all the metrics per subject and add them to the
storage
            for (String metric : ((Map<String, List<Integer>>)
argument).keySet()) {
                int value = (int)((Map) argument).get(metric);
                metricStorage.get(source).computeIfAbsent((String) metric, tmp
-> new ArrayList<>());
                metricStorage.get(source).get(metric).add(value);

                // additionally, if thresholds have been set, check if they
have been exceeded and print a warning
                for(String importantMetric : thresholds.keySet()) {
                    if(metric.contains(importantMetric) && value >=
thresholds.get(importantMetric)) {
                        System.out.println("WARNING: Sensor "+metric+"
exceeded threshold! ("+value+")");
                    }
                }
            }

            //System.out.println(this + " profiler received update from " +
source + ": " + argument);
        }
    }
}
```

Protocol.java

```
package swp4.ue03.protocol;

import swp4.ue03.profiling.Subject;

import java.util.List;
import java.util.Map;

public interface Protocol {

    // every protocol needs to know how to evaluate data
    Map<String, List<Integer>> evaluateData(Map<Subject, Map<String,
List<Integer>>> storedData, int seconds);
    Map<String, List<Integer>> evaluateData(Map<Subject, Map<String,
List<Integer>>> storedData);

}
```

MemoryProtocol.java

```
package swp4.ue03.protocol.impl;

import swp4.ue03.components.impl.MemoryComponent;
import swp4.ue03.profiling.Subject;
import swp4.ue03.protocol.Protocol;

import java.util.*;

public class MemoryProtocol implements Protocol {

    private static MemoryProtocol memoryProtocol = null;

    private MemoryProtocol() { }

    public static MemoryProtocol getInstance() {
        if(memoryProtocol == null) {
            memoryProtocol = new MemoryProtocol();
        }
        return memoryProtocol;
    }


    @Override
    public Map<String, List<Integer>> evaluateData(Map<Subject, Map<String,
List<Integer>>> storedData, int seconds) {
        Map<String, List<Integer>> tmp = new HashMap<>();

        // Search for the right components
        for(Subject subject : storedData.keySet()) {
            if (subject instanceof MemoryComponent) {
                // iterate through all the metrics for the right component and
filter for time
                for (String metric : storedData.get(subject).keySet()) {
                    ArrayList<Integer> current = (ArrayList<Integer>)
storedData.get(subject).get(metric);
                    int begin = seconds == 0 ? 0 : Math.max(current.size() -
seconds, 0);
                    tmp.put(((MemoryComponent)subject).getName()+";"+metric,
current.subList(begin, current.size()));
                }
            }
        }

        return tmp;
    }

    @Override
    public Map<String, List<Integer>> evaluateData(Map<Subject, Map<String,
List<Integer>>> storedData) {
        return evaluateData(storedData, 0);
    }
}
```

NetworkProtocol.java

```
package swp4.ue03.protocol.impl;

import swp4.ue03.components.impl.MemoryComponent;
import swp4.ue03.components.impl.NetworkComponent;
import swp4.ue03.profiling.Subject;
import swp4.ue03.protocol.Protocol;

import java.util.*;

public class NetworkProtocol implements Protocol {
    private static NetworkProtocol networkProtocol = null;

    private NetworkProtocol() { }

    public static NetworkProtocol getInstance() {
        if(networkProtocol == null) {
            networkProtocol = new NetworkProtocol();
        }
        return networkProtocol;
    }

    @Override
    public Map<String, List<Integer>> evaluateData(Map<Subject, Map<String,
List<Integer>>> storedData, int seconds) {
        Map<String, List<Integer>> tmp = new HashMap<>();

        // Search for the right components
        for(Subject subject : storedData.keySet()) {
            if (subject instanceof NetworkComponent) {
                // iterate through all the metrics for the right component and
filter for time
                for (String key : storedData.get(subject).keySet()) {
                    ArrayList<Integer> current = (ArrayList<Integer>)
storedData.get(subject).get(key);
                    int begin = seconds == 0 ? 0 : Math.max(current.size() -
seconds, 0);
                    tmp.put(((NetworkComponent)subject).getName()+";"+key,
current.subList(begin, current.size()));
                }
            }
        }

        return tmp;
    }

    @Override
    public Map<String, List<Integer>> evaluateData(Map<Subject, Map<String,
List<Integer>>> storedData) {
        return evaluateData(storedData, 0);
    }
}
```

PowerProtocol.java

```
package swp4.ue03.protocol.impl;

import swp4.ue03.components.impl.MemoryComponent;
import swp4.ue03.components.impl.NetworkComponent;
import swp4.ue03.components.impl.PowerComponent;
import swp4.ue03.profiling.Subject;
import swp4.ue03.protocol.Protocol;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class PowerProtocol implements Protocol {

    private static PowerProtocol psuProtocol = null;

    private PowerProtocol() { }

    public static PowerProtocol getInstance() {
        if(psuProtocol == null) {
            psuProtocol = new PowerProtocol();
        }
        return psuProtocol;
    }

    @Override
    public Map<String, List<Integer>> evaluateData(Map<Subject, Map<String,
List<Integer>>> storedData, int seconds) {
        Map<String, List<Integer>> tmp = new HashMap<>();

        // Search for the right components
        for(Subject subject : storedData.keySet()) {
            if (subject instanceof PowerComponent) {
                // iterate through all the metrics for the right component and
filter for time
                for (String metric : storedData.get(subject).keySet()) {
                    ArrayList<Integer> current = (ArrayList<Integer>)
storedData.get(subject).get(metric);
                    int begin = seconds == 0 ? 0 : Math.max(current.size() -
seconds, 0);
                    tmp.put(((PowerComponent)subject).getName()+";"+metric,
current.subList(begin, current.size()));
                }
            }
        }

        return tmp;
    }

    @Override
    public Map<String, List<Integer>> evaluateData(Map<Subject, Map<String,
List<Integer>>> storedData) {
        return evaluateData(storedData, 0);
    }
}
```

ProcessorProtocol.java

```
package swp4.ue03.protocol.impl;

import swp4.ue03.components.impl.MemoryComponent;
import swp4.ue03.components.impl.ProcessorComponent;
import swp4.ue03.profiling.Subject;
import swp4.ue03.protocol.Protocol;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class ProcessorProtocol implements Protocol {

    private static ProcessorProtocol cpuProtocol = null;

    private ProcessorProtocol() { }

    public static ProcessorProtocol getInstance() {
        if(cpuProtocol == null) {
            cpuProtocol = new ProcessorProtocol();
        }
        return cpuProtocol;
    }

    @Override
    public Map<String, List<Integer>> evaluateData(Map<Subject, Map<String,
List<Integer>>> storedData, int seconds) {
        Map<String, List<Integer>> tmp = new HashMap<>();

        // Search for the right components
        for(Subject subject : storedData.keySet()) {
            if (subject instanceof ProcessorComponent) {
                // iterate through all the metrics for the right component and
filter for time
                for (String metric : storedData.get(subject).keySet()) {
                    ArrayList<Integer> current = (ArrayList<Integer>)
storedData.get(subject).get(metric);
                    int begin = seconds == 0 ? 0 : Math.max(current.size() -
seconds, 0);

tmp.put(((ProcessorComponent)subject).getName()+";"+metric,
current.subList(begin, current.size()));
                }
            }
        }

        return tmp;
    }

    @Override
    public Map<String, List<Integer>> evaluateData(Map<Subject, Map<String,
List<Integer>>> storedData) {
        return evaluateData(storedData, 0);
    }
}
```

StorageProtocol.java

```
package swp4.ue03.protocol.impl;

import swp4.ue03.components.impl.MemoryComponent;
import swp4.ue03.components.impl.StorageComponent;
import swp4.ue03.profiling.Subject;
import swp4.ue03.protocol.Protocol;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class StorageProtocol implements Protocol {

    private static StorageProtocol storageProtocol = null;

    private StorageProtocol() { }

    public static StorageProtocol getInstance() {
        if(storageProtocol == null) {
            storageProtocol = new StorageProtocol();
        }
        return storageProtocol;
    }


    @Override
    public Map<String, List<Integer>> evaluateData(Map<Subject, Map<String,
List<Integer>>> storedData, int seconds) {
        Map<String, List<Integer>> tmp = new HashMap<>();

        // Search for the right components
        for(Subject subject : storedData.keySet()) {
            // check if the component is storageCOmponent
            if (subject instanceof StorageComponent) {
                // iterate through all the metrics for the right component and
filter for time
                for (String metric : storedData.get(subject).keySet()) {
                    ArrayList<Integer> current = (ArrayList<Integer>)
storedData.get(subject).get(metric);
                    int begin = seconds == 0 ? 0 : Math.max(current.size() -
seconds, 0);
                    tmp.put(((StorageComponent)subject).getName()+";"+metric,
current.subList(begin, current.size()));
                }
            }
        }

        return tmp;
    }

    @Override
    public Map<String, List<Integer>> evaluateData(Map<Subject, Map<String,
List<Integer>>> storedData) {
        return evaluateData(storedData, 0);
    }
}
```

ProtocolPrinter.java

```
package swp4.ue03.protocol.impl;

import swp4.ue03.components.Component;
import swp4.ue03.components.impl.MemoryComponent;
import swp4.ue03.profiling.Subject;
import swp4.ue03.protocol.Protocol;

import java.io.FileWriter;
import java.io.IOException;
import java.util.*;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.ConcurrentMap;

public class ProtocolPrinter {
    List<Protocol> protocolList = new ArrayList<>();
    public enum protocolType{NUMBER,CSV}

    // filter the given data depending on the attached protocols
    // return data as a sorted TreeMap
    public synchronized TreeMap<String, List<Integer>> filter(Map<Subject,
Map<String, List<Integer>>> totalData, int mins) {
        Map<String, List<Integer>> concData = new ConcurrentHashMap<>();

        for(Protocol protocol : protocolList) {
            // evaluate each subprotocol and add the metrics to the map
            Map<String, List<Integer>> evaluatedData =
protocol.evaluateData(totalData, mins);
            for(String metric : evaluatedData.keySet()) {
                concData.put(metric, evaluatedData.get(metric));
            }
        }

        return new TreeMap<>(concData);
    }

    public synchronized void print(protocolType type, String filename,
Map<Subject, Map<String, List<Integer>>> totalData, int seconds) {
        TreeMap<String, List<Integer>> evalData = filter(totalData, seconds);

        // If protocol should be numeric, print min/max/avg/median to console
        if(type == protocolType.NUMBER) {
            for(String metric : evalData.keySet()) {
                List<Integer> curValues =
Collections.synchronizedList(evalData.get(metric));
                double min = curValues.stream().mapToDouble(d ->
d).min().orElse(0.0);
                double max = curValues.stream().mapToDouble(d ->
d).max().orElse(0.0);
                double avg = curValues.stream().mapToDouble(d ->
d).average().orElse(0.0);
                double median = curValues.size()%2==0 ?
(curValues.get(curValues.size()/2-1)+curValues.get(curValues.size()/2)) :
curValues.get(curValues.size()/2);
                System.out.println(metric+": "+min+", "+max+", "+avg+",
"+median);
            }
        } else { // if it should be printed as csv, open a new file to write
```

```
            try(FileWriter fout = new FileWriter(filename)) {
                fout.write(String.join(",",evalData.keySet())+"\n");

                // determine linecount, if period exceeds the components time
alive then we print less lines
                // get the maximum amount of entries
                int linecount = 0;
                for(String metric : evalData.keySet()) {
                    linecount = Math.max(linecount,
evalData.get(metric).size());
                }

                // print all metrics that are available in the given period
                for(int i=0; i < linecount; i++) {
                    boolean first = true;
                    // go through all metrics, every line
                    for(String metric:evalData.keySet()) {
                        if(first) {
                            first = false;
                        } else {
                            fout.write(",");
                        }
                        // print the metric in a comma separated fashion
                        if(i < evalData.get(metric).size()) {

fout.write(evalData.get(metric).get(i).toString());
                        }
                    }
                    fout.write("\n");
                }
            } catch(IOException e)
            {
                System.out.println("Could not write to file: " + filename);
                e.printStackTrace();
            }
        }
    }

    public synchronized void print(protocolType type, Map<Subject, Map<String,
List<Integer>>> totalData, int seconds) {
        if(type == protocolType.NUMBER) {
            print(type, "",totalData, seconds);
        } else {
            System.err.println("To use the CSV output the filename field is
mandatory!");
        }
    }

    // add a protocol to the ProtocolPrinter, so it knows what to print
    public void addProtocol(Protocol protocol) {
        protocolList.add(protocol);
    }
}
```

Testfälle

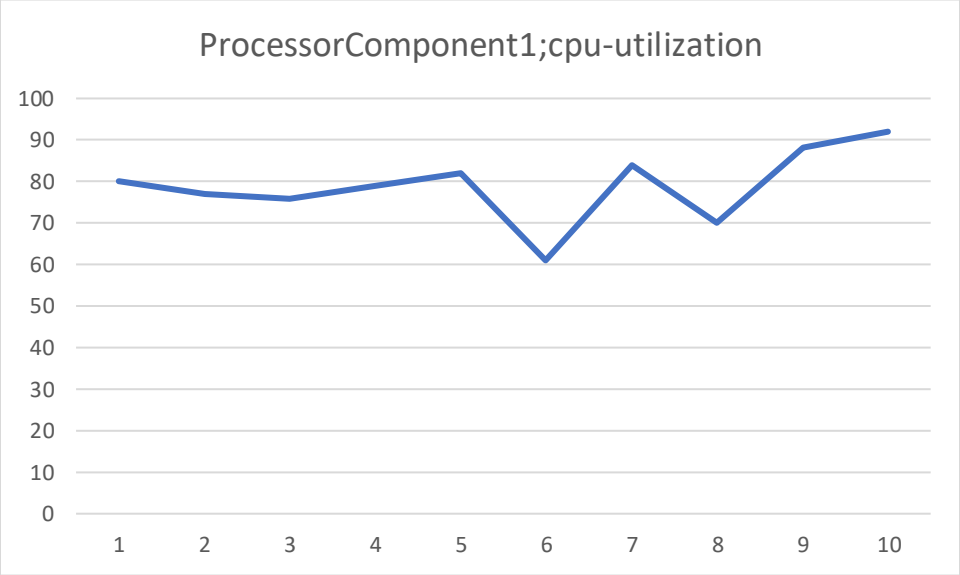
- Numerische Ausgabe
- Ausgabe als Diagramm
- Mehrere gleichartige Komponenten je Profiler
- Threshold Ausgabe
- Ausgabe eines Protokolls mit Komponenten die nie gestartet wurden
- Ausgabe eines Protokolls ohne Subprotokolle

Numerische Ausgabe:

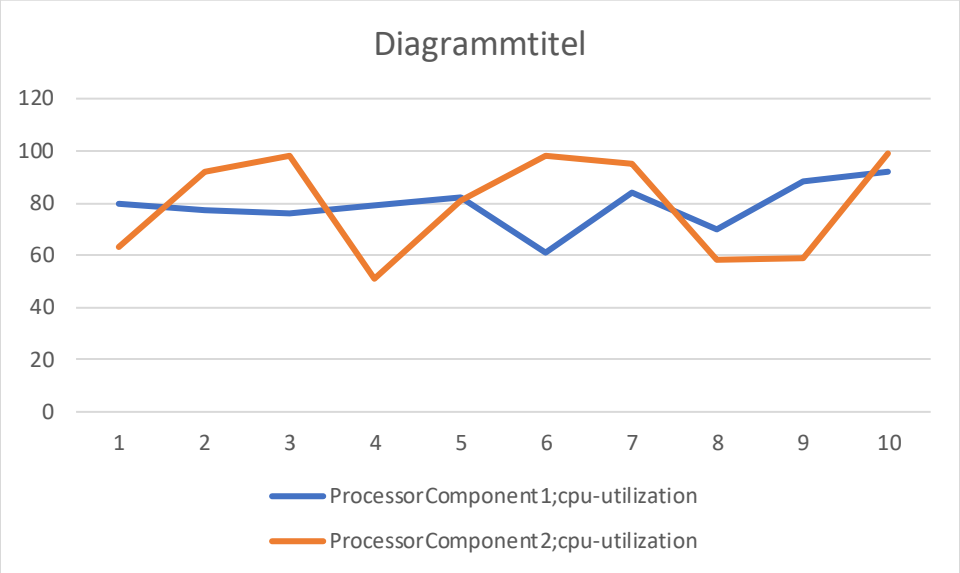
```
MemoryComponent1;free: 8699.0, 21447.0, 13349.6, 12651.0
MemoryComponent1;used: 11321.0, 24069.0, 19418.4, 20117.0
NetworkComponent1;received: 561041.0, 1.707654E7, 1.1892581E7, 1.150967E7
NetworkComponent1;sent: 1527065.0, 5007722.0, 3791425.8, 5007722.0
PowerComponent1;powerconsumption: 193.0, 721.0, 414.0, 193.0
```

Ausgabe als Diagramm:

Wird resultierende CSV in Excel importiert (Reiter: Daten, CSV importieren) so kann für jede Metrik ein Diagramm gebildet werden:



Mehrere gleichartige Komponenten:



Threshold Ausgabe:

```
WARNING: Sensor cpu-utilization exceeded threshold! (92)
WARNING: Sensor cpu-utilization exceeded threshold! (98)
WARNING: Sensor cpu-utilization exceeded threshold! (98)
WARNING: Sensor cpu-utilization exceeded threshold! (95)
```

Für dieses Beispiel wurde der Threshold auf 90 gesetzt. Die Überschreitung der Grenzwerte ist auch im Diagramm oben zu sehen.

Ausgabe eines Protokolls mit Komponenten die nie gestartet wurden:

```

Process finished with exit code 0
```

Es führt zu keiner Ausgabe, da die Map einfach leer ist.

Ausgabe eines Protokolls ohne Subprotokolle:

```

Process finished with exit code 0
```

Es kommt wiederum keine Ausgabe, da die Daten zwar befüllt sind, aber der Filter alle Daten filtert, da keine Protokolle hinterlegt sind. Ich habe mein Beispiel so definiert, dass die Protokolle explizit angegeben werden müssen.