Университет ИТМО

Факультет программной инженерии и компьютерных технологий

Лабораторная работа №4 по Методам и Средствам Программной Инженерии

Выполнил: Богатов Александр Сергеевич

Группа: P3233

Вариант: 6699

Преподаватель: Цопа Евгений Алексеевич

Санкт-Петербург

2022

Задание:

1. Для своей программы из лабораторной работы #3 по дисциплине "Веб-программирование" реализовать:

- MBean, считающий общее число установленных пользователем точек, а также число точек, не попадающих в область. В случае, если количество установленных пользователем точек стало кратно 15, разработанный MBean должен отправлять оповещение об этом событии.
- MBean, определяющий процентное отношение "промахов" к общему числу кликов пользователя по координатной плоскости.

2. С помощью утилиты JConsole провести мониторинг программы:

- Снять показания MBean-классов, разработанных в ходе выполнения задания 1.
- Определить количество классов, загруженных в JVM в процессе выполнения программы.

3. С помощью утилиты VisualVM провести мониторинг и профилирование программы:

- Снять график изменения показаний MBean-классов, разработанных в ходе выполнения задания 1, с течением времени.
- Определить имя класса, объекты которого занимают наибольший объём памяти JVM; определить пользовательский класс, в экземплярах которого находятся эти объекты.

4. С помощью утилиты VisualVM и профилировщика IDE NetBeans, Eclipse или Idea локализовать и устранить проблемы с производительностью в программе. По результатам локализации и устранения проблемы необходимо составить отчёт, в котором должна содержаться следующая информация:

- Описание выявленной проблемы.
- Описание путей устранения выявленной проблемы.
- Подробное (со скриншотами) описание алгоритма действий, который позволил выявить и локализовать проблему.

Студент должен обеспечить возможность воспроизведения процесса поиска и локализации проблемы по требованию преподавателя.

MBean методы:

```
@Override
    public long getResultAmount() {
        loadResults();
        return results.size();
    }
```

```
    @Override
    public long getSVGResultAmount() {
        loadResults();
        return results.stream().filter(result -> result.getType().equals("fromSVG")).count();
    }


    @Override
    public long getMissAmount() {
        loadResults();
        return results.stream().filter(result -> result.getHit() == false).count();
    }

    @Override
    public long checkPointAmountDivisor() {
        long amount = getResultAmount();
        long misses = getMissAmount();
        resultAmount = amount;
        missAmount = misses;
        if (amount % 15 == 0) {
            sendNotification(new Notification("Result amount can be divided by 15", this.getClass().getName(),
sequenceNumber++, "Overall number of results: " + results.size() + "\n Missed results: " + misses));
        }
        return amount;
    }

    @Override
    public double getMissPercentage() {
        long amount = getSVGResultAmount();
        long misses = getMissAmount();
        svgAmount = amount;
        missAmount = misses;
        if (amount != 0) {
            missPercentage = misses * 100 / amount;
            return misses * 100 / amount;
        }
        missPercentage = 0;
```

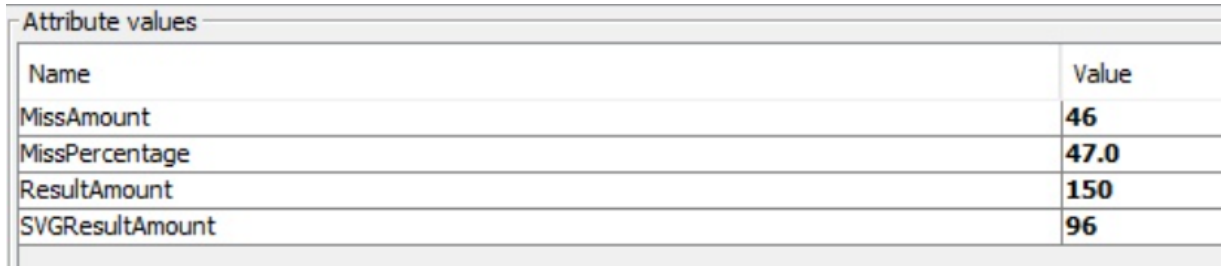
```
        return 0;
    }
```

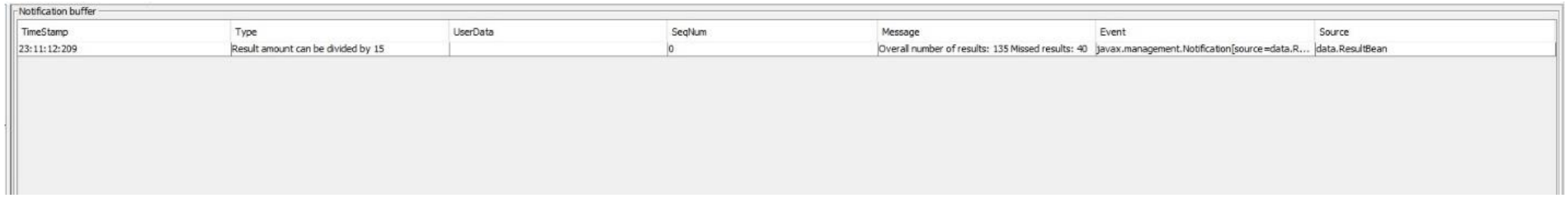
MXBeanContextListener:

```
package management;

import data.ResultBean;

import javax.management.*;
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
import java.lang.management.ManagementFactory;

public class MXBeanContextListener implements ServletContextListener {

    @Override
    public void contextInitialized(ServletContextEvent sce) {
        try {
            MBeanServer = ManagementFactory.getPlatformMBeanServer();
            ResultBean bean = new ResultBean();
            ObjectName beanObjName = new ObjectName("data:type=mbeans,name=result");
            mBeanServer.registerMBean(bean, beanObjName);
            MXBeanListener beanListener = new MXBeanListener();
            mBeanServer.addNotificationListener(beanObjName, beanListener,
beanListener.getNotificationFilter(), null);
        } catch (InstanceAlreadyExistsException | MalformedObjectNameException | MBeanRegistrationException |
NotCompliantMBeanException e) {
            e.printStackTrace();
        } catch (InstanceNotFoundException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void contextDestroyed(ServletContextEvent sce) {
```

```
        try {
            ManagementFactory.getPlatformMBeanServer().unregisterMBean(new
ObjectName("data:type=mbeans,name=result"));
        } catch (MBeanRegistrationException | InstanceNotFoundException | MalformedObjectNameException e) {
            e.printStackTrace();
        }
    }
}
```

Показания MBeans:

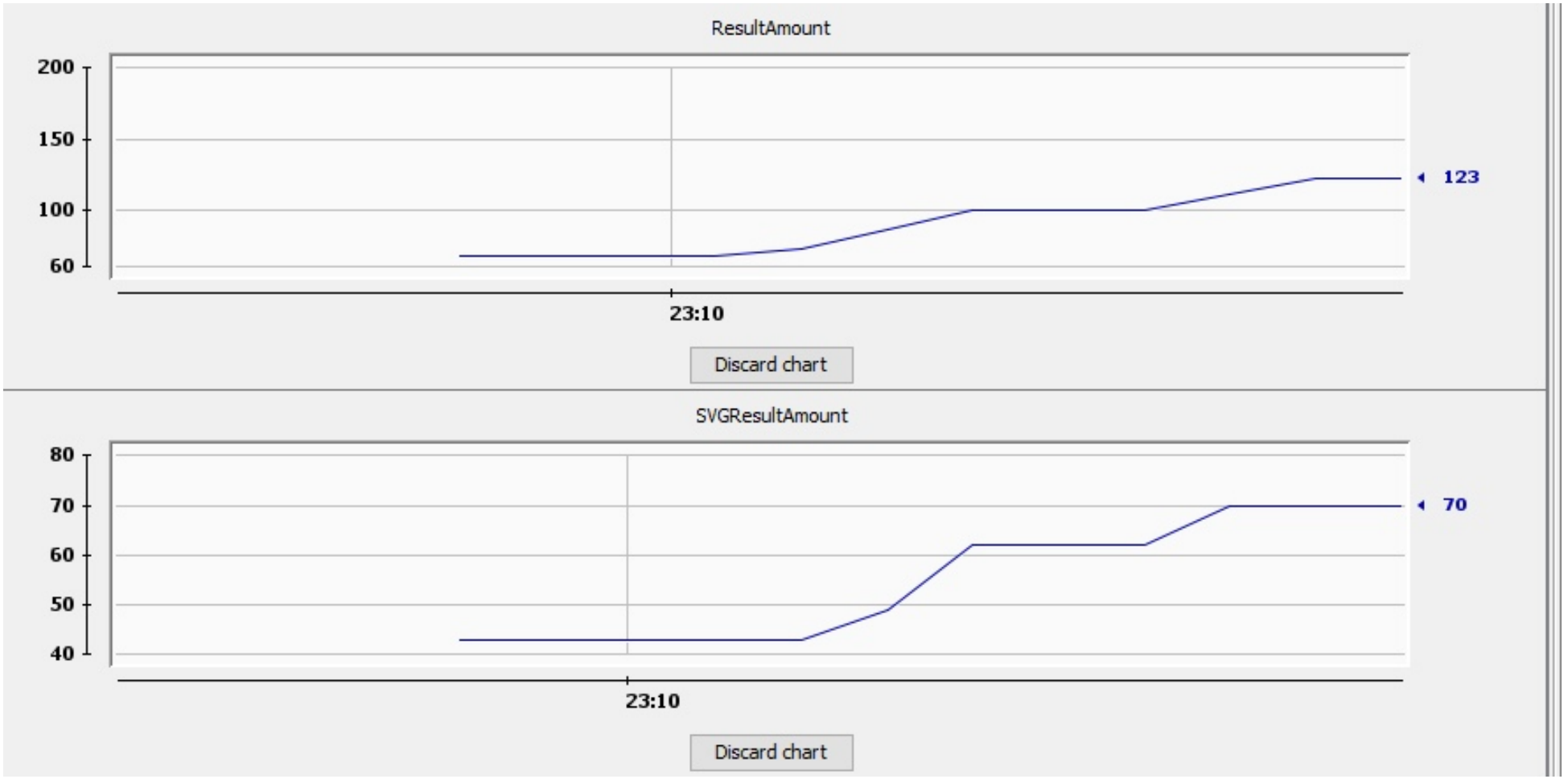




Изменение атрибутов MBean:

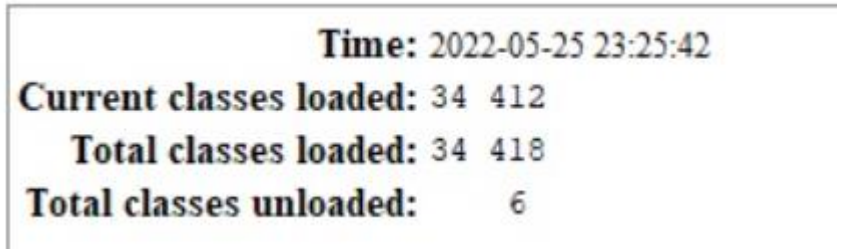
Попользуемся разработанным сайтом какое-то время и посмотрим на то, как изменяются значения атрибутов MBeans.


MissAmount
40
30
20
10
0
34
23:10
Discard chart
MissPercentage
50,
40,
30,
20,
10,
0,
48,0000
23:10
Discard chart



Количество классов в JVM:

Информацию о числе загруженных и выгруженных из JVM классов можно найти во вкладке Classes.



Класс, занимающий больше всего памяти:

Сделав дамп кучи, выясняем, что больше всего места в JVM занимают объекты класса HashMap, который при написании лабораторной не использовался. Вычисляем корень GC для объектов класса и видим, что наибольшее число было создано модулем JBoss, отвечающим за загрузку классов.

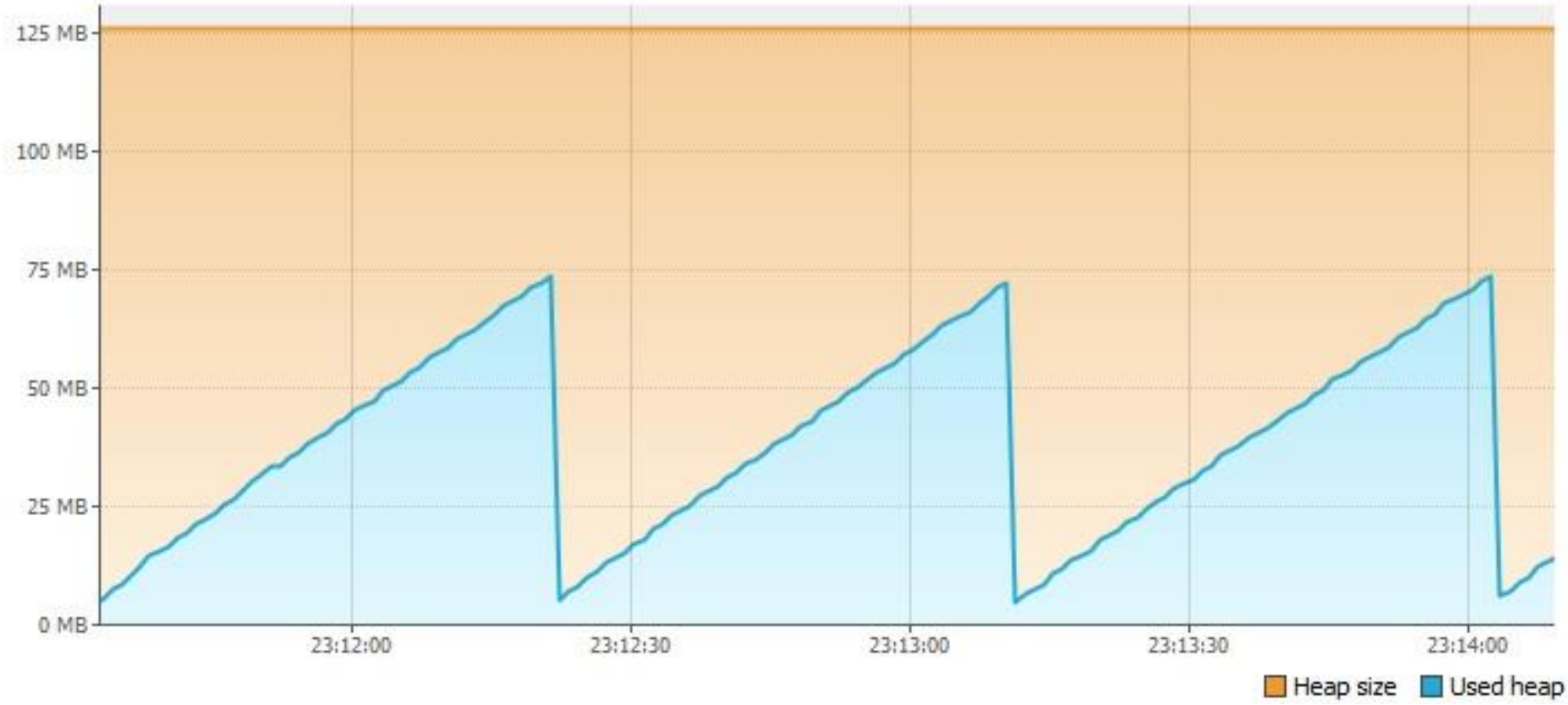
Исправление утечки:

Проводим профилирование на протяжении двух с половиной минут. При наблюдении за кучей JVM видно, что программа постоянно выделяет много места для объектов, которые разом собираются сборщиком мусора. Это показывает наличие в программе создания одинаковых элементов, не участвующих в работе программы.

Size: 132 120 608 B

Used: 14 971 944 B

Max: 2 107 637 784 B



Heap histogram | Per thread allocations

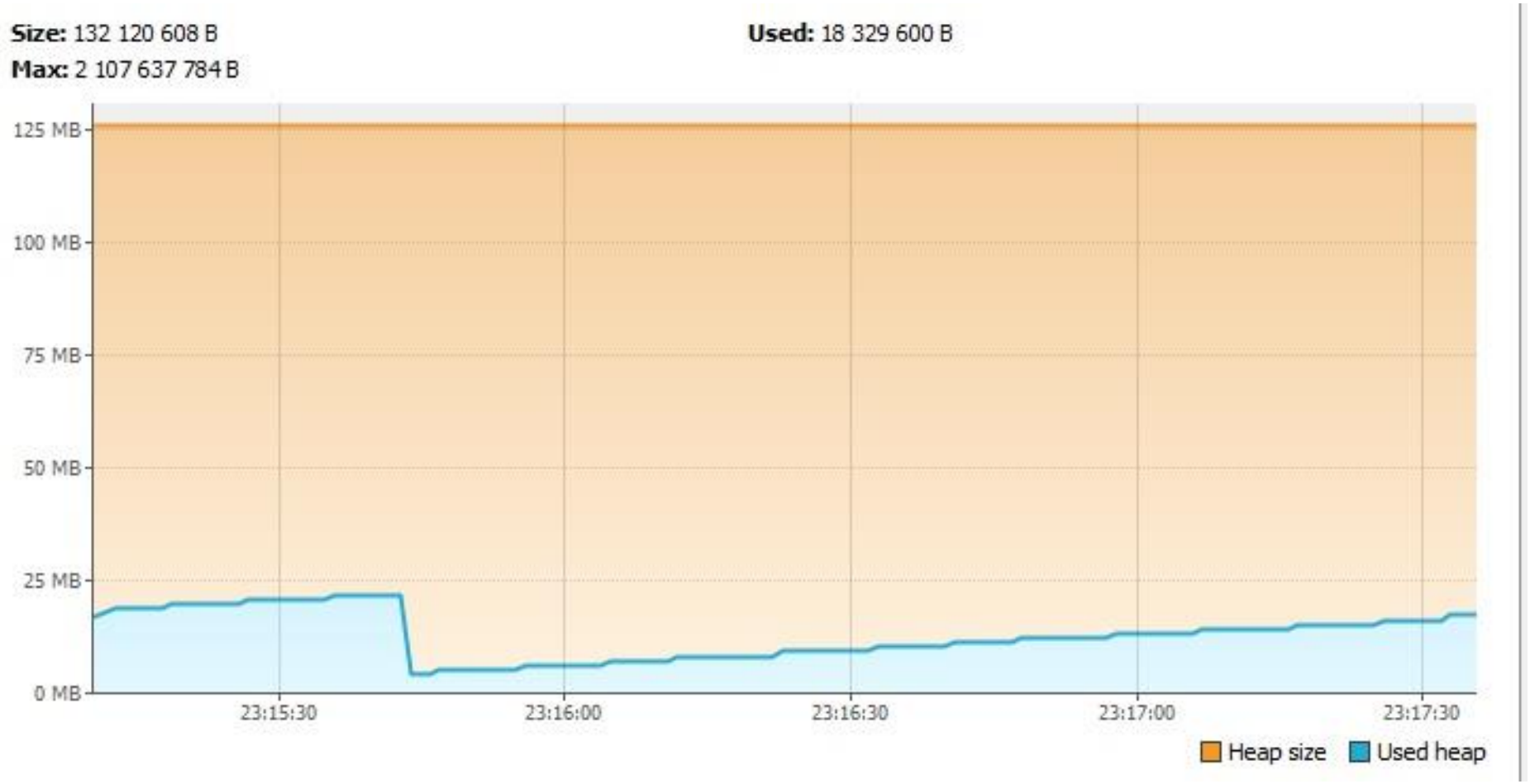
Results: | Collected data: Snapshot | Perform GC | Heap Dump

Name	Live Bytes	Live Objects
byte[]	14 663 216 B (32,6 %)	172 052 (26 %)
char[]	6 460 760 B (14,4 %)	7 411 (1,1 %)
java.lang.reflect.Method	3 313 904 B (7,4 %)	37 658 (5,7 %)
java.lang.Object[]	2 463 432 B (5,5 %)	39 818 (6 %)

В главном методе видим вызов метода getResponse, влекущего за собой создание нового объекта. Метод вызывается внутри бесконечного цикла, хотя для корректной работы программы достаточно иметь один постоянный экземпляр класса WebResponse.

Переместим определение WebResponse из цикла.

При повторном профилировании видно сильное улучшение ситуации, память больше не расходуется на бесполезные копии объекта.



Name	Live Bytes	Live Objects
byte[]	4 579 968 B (20,6 %)	48 711 (11,1 %)
java.lang.Object[]	3 337 944 B (15 %)	86 107 (19,6 %)
int[]	2 611 888 B (11,7 %)	6 207 (1,4 %)
java.util.TreeMap$Entry	1 977 560 B (8,9 %)	49 439 (11,2 %)

Вывод:

При выполнении данной лабораторной работы были изучены методы мониторинга и профилирования Java приложений с помощью утилит JConsole и VisulaVM