LIITE 3. Mittausohjelmiston lähdekoodi

Timo Tuominen

Tiedostot

NativeBenchmark

Java-komponentit

Hakemistossa src/fi/helsinki/cs/tituomin/nativebenchmark/.

ApplicationState.java

```java
package fi.helsinki.cs.tituomin.nativebenchmark;

import android.content.res.Resources;

public interface ApplicationState {
    public void updateState(State state);

    public void updateState(State state, String message);

    public boolean userWantsToRetry(Exception exception);

    public DetailedState getState();

    public Resources getResources();

    public static enum State {
        INITIALISED(R.string.app_name),
        MEASURING_STARTED(R.string.measuring_started),
        INTERRUPTING(R.string.interrupting),
        INTERRUPTED(R.string.interrupted),
        MILESTONE(R.string.measuring_milestone),
        ERROR(R.string.error),
        INIT_FAIL(R.string.error),
        MEASURING_FINISHED(R.string.measuring_finished);

        public final int stringId;

        State(int stringId) {
            this.stringId = stringId;
        }
    }

    public class DetailedState {
        public State state;
        public String message;
        private ApplicationState parent;

        public DetailedState(ApplicationState parent) {
            this.parent = parent;
            this.state = null;
            this.message = null;
        }

        public DetailedState(ApplicationState parent, DetailedState d) {
            this.parent = parent;
            this.state = d.state;
            this.message = d.message;
        }

        public String toString() {
            Resources resources = parent.getResources();
            if (this.state == null) {
                return "<unknown state>";
            }
            String type = resources.getString(this.state.stringId);
            return String.format(
                    "%s%s", type,
                    (this.message != null) ? " " + this.message : "");
        }
    }
}
```

ApplicationStateListener.java

```
package fi.helsinki.cs.tituomin.nativebenchmark;

public interface ApplicationStateListener {

    public void stateUpdated(ApplicationState.DetailedState state);
}
```

BenchmarkController.java

```
package fi.helsinki.cs.tituomin.nativebenchmark;

import android.app.ActivityManager;
import android.content.Context;
import android.content.res.Resources;
import android.os.PowerManager;
import android.util.Log;

import java.io.File;
import java.io.IOException;

import fi.helsinki.cs.tituomin.nativebenchmark.measuringtool.MeasuringTool;
```

```java

public class BenchmarkController implements ApplicationState {

    // todo: state changing things synchronized

    public BenchmarkController(Context aContext, File dataDir) {
        this.detailedState = new ApplicationState.DetailedState(this);
        this.dataDir = dataDir;
        this.listeners = new Listeners();

        PowerManager pm = (PowerManager) aContext.getSystemService(Context
                .POWER_SERVICE);
        wakeLock = pm.newWakeLock(PowerManager.PARTIAL_WAKE_LOCK,
                "Benchmarking");
        ActivityManager am = (ActivityManager) aContext.getSystemService
                (Context.ACTIVITY_SERVICE);
        this.resources = aContext.getResources();
        int memoryClass = am.getLargeMemoryClass();
        Log.v("Selector", "Memory size " + Runtime.getRuntime().maxMemory());
        Log.v("onCreate", "memoryClass:" + Integer.toString(memoryClass));

    }

    public Resources getResources() {
        return this.resources;
    }

    public void updateState(ApplicationState.State state) {
        updateState(state, null);
    }

    public void updateState(ApplicationState.State state, String message) {
        synchronized (this) {
            this.detailedState.state = state;
            this.detailedState.message = message;

            switch (state) {
                case MEASURING_STARTED:
                    try {
                        LogAccess.start(dataDir);
                    } catch (IOException e) {
                        this.detailedState.state = ApplicationState.State.ERROR;
                        this.detailedState.message = "Could not initialize " +
                                "log file.";
                        Log.e(TAG, this.detailedState.message);
                    }
                    wakeLock.acquire();
                    break;
                case ERROR:
                case INTERRUPTED:
                case MEASURING_FINISHED:
                    if (wakeLock.isHeld()) {
                        wakeLock.release();
                    }
                    LogAccess.end();
                case INITIALISED:
                case INIT_FAIL:
                case MILESTONE:
                    if (this.listeners.milestoneListener != null) {
                        this.listeners.milestoneListener.stateUpdated(this
                                .detailedState);
                    }
            }
        }
    }

    public ApplicationState.DetailedState getState() {
        synchronized (this) {
            return new ApplicationState.DetailedState(this, this.detailedState);
        }
    }

    public boolean userWantsToRetry(Exception e) {
        return false; // TODO: make interaction configurable
    }

    private class BenchRunnable implements Runnable {
        BenchRunnable(BenchmarkRunner runner, ToolConfig configuration) {
            this.runner = runner;
            this.configuration = configuration;
        }

        public void run() {
            this.runner.runBenchmarks(
                    BenchmarkController.this,
                    this.configuration,
                    dataDir);
        }

        private BenchmarkRunner runner;
        private ToolConfig configuration;
    }

    public void startMeasuring(BenchmarkRunner runner, ToolConfig
            configuration) {
        String message = null;
        if (this.resources.getString(R.string.app_dirty).equals("1")) {
            message = this.resources.getString(R.string.warning_changed);
        }

        BenchmarkRunner.BenchmarkSet set = configuration.getBenchmarkSet();
        runner.setBenchmarkSet(set);
        measuringThread = new Thread(new BenchRunnable(runner, configuration));
        if (message != null) {
            this.updateState(ApplicationState.State.MEASURING_STARTED, message);
        } else {
```

```java
            this.updateState(ApplicationState.State.MEASURING_STARTED);
        }
        measuringThread.start();
    }

    public void interruptMeasuring() {
        MeasuringTool.userInterrupt();
        measuringThread.interrupt();
    }

    public void addListener(ApplicationStateListener listener,
                            ApplicationState.State state) {
        if (state == ApplicationState.State.MILESTONE) {
            this.listeners.milestoneListener = listener;
        }
    }

    public void removeListeners() {
        this.listeners = null;
    }

    private class Listeners {
        public ApplicationStateListener milestoneListener;

        public Listeners() {
            milestoneListener = null;
        }
    }

    private Listeners listeners;
    private ApplicationState.State state;
    private String message;
    private ApplicationState.DetailedState detailedState;
    private PowerManager.WakeLock wakeLock;
    private File dataDir;
    private Thread measuringThread;
    private static final String TAG = "BenchmarkController";
    private Resources resources;

}
```

Benchmark.java

```java
package fi.helsinki.cs.tituomin.nativebenchmark;

import android.os.Process;

public abstract class Benchmark implements Runnable {
    protected abstract void runInternal();

    public abstract String from();

    public abstract String to();

    public abstract String description();

    public abstract String id();

    public abstract int sequenceNo();

    public abstract boolean isAllocating();

    public abstract boolean isNonvirtual();

    public abstract boolean dynamicParameters();

    public abstract boolean representative();

    public long repetitionsLeft;

    public void run() {
        Process.setThreadPriority(-5);
        runInternal();
    }

    protected BenchmarkParameter benchmarkParameter;
    protected long repetitions;

    public void init(BenchmarkParameter bp) {
        repetitionsLeft = 0;
        benchmarkParameter = bp;
        repetitions = -1;
    }

    public void setRepetitions(long reps) {
        if (reps < 1) {
            return;
        }
        repetitions = reps;
        BenchmarkRegistry.setRepetitions(reps);
    }
}
```

BenchmarkParameter.java

```java
package fi.helsinki.cs.tituomin.nativebenchmark;

import android.content.pm.PermissionInfo;
import android.util.Log;

import java.nio.ByteBuffer;
import java.util.Arrays;
import java.util.Iterator;
import java.util.NoSuchElementException;

import fi.helsinki.cs.tituomin.nativebenchmark.benchmark.JavaCounterparts;

// Important! Imported as an example class.

public class BenchmarkParameter implements Iterable<Integer> {

    private native int initReturnvalues(int size, MockObject o);

    private native void freeReturnvalues();

    public void setUp() {
        initReturnvalues(index * DEFAULTSIZE, mockObjectInstance);
        JavaCounterparts.initParams(this);
    }

    public void tearDown() {
        freeReturnvalues();
    }

    public static final int DEFAULTSIZE = 64;
    public static final int RANGE = 8;
    public static final int MAXSIZE = DEFAULTSIZE * RANGE;

    // The byte buffer has to hold longs and doubles too.
    public static final int NIO_MAXSIZE = DEFAULTSIZE * RANGE * 8;

    public static MockObject mockObjectInstance = new MockObject();

    public MockObject retrieveMockObject() {
        return mockObjectInstance;
    }

    public BenchmarkParameter() {
        index = 1;
        if (!generated) {
            generateAll();
            generated = true;
        }

        for (int i = 0; i < RANGE + 1; i++) {
            if (STRINGS[i] == null) {
                Log.v("Parameter", "STRINGS is null at " + i);
            }
            if (OBJECTS[i] == null) {
                Log.v("Parameter", "OBJECTS is null at " + i);
            }
            if (THROWABLES[i] == null) {
                Log.v("Parameter", "THROWABLES is null at " + i);
            }
            if (BOOLEAN_ARRAYS[i] == null) {
                Log.v("Parameter", "BOOLEAN_ARRAYS is null at " + i);
            }
            if (BYTE_ARRAYS[i] == null) {
                Log.v("Parameter", "BYTE_ARRAYS is null at " + i);
            }
            if (CHAR_ARRAYS[i] == null) {
                Log.v("Parameter", "CHAR_ARRAYS is null at " + i);
            }
            if (DOUBLE_ARRAYS[i] == null) {
                Log.v("Parameter", "DOUBLE_ARRAYS is null at " + i);
            }
            if (FLOAT_ARRAYS[i] == null) {
                Log.v("Parameter", "FLOAT_ARRAYS is null at " + i);
            }
            if (INT_ARRAYS[i] == null) {
                Log.v("Parameter", "INT_ARRAYS is null at " + i);
            }
            if (LONG_ARRAYS[i] == null) {
                Log.v("Parameter", "LONG_ARRAYS is null at " + i);
            }
            if (SHORT_ARRAYS[i] == null) {
                Log.v("Parameter", "SHORT_ARRAYS is null at " + i);
            }
            if (OBJECT_ARRAYS[i] == null) {
                Log.v("Parameter", "OBJECT_ARRAYS is null at " + i);
            }
        }
    }


    // must call initreturnvalues and freereturnvalues after...
    public void setIndex(int index) {
        if (index < (RANGE + 1)) {
            this.index = index;
        } else {
            throw new IllegalArgumentException("Requested size too large. " +
                    index);
        }
    }

    public int getIndex() {
        return this.index;
    }

    public int getSize() {
        return DEFAULTSIZE * index;
    }


    public Iterator<Integer> iterator() {
        return new RangeIterator();
    }

    private class RangeIterator implements Iterator<Integer> {
        private int index;

        public RangeIterator() {
            index = -1;
        }

        public boolean hasNext() {
            return index < RANGE;
        }

        public Integer next() {
            if (!hasNext()) {
                throw new NoSuchElementException();
            }
            index++;
            setIndex(index);
            return DEFAULTSIZE * index;

        }

        public void remove() {
            throw new UnsupportedOperationException();
        }
    }

    public boolean[] retrieveBooleanArray() {
        return BOOLEAN_ARRAYS[index];
    }

    public byte[] retrieveByteArray() {
        return BYTE_ARRAYS[index];
    }

    public char[] retrieveCharArray() {
        return CHAR_ARRAYS[index];
    }

    public double[] retrieveDoubleArray() {
        return DOUBLE_ARRAYS[index];
    }

    public float[] retrieveFloatArray() {
        return FLOAT_ARRAYS[index];
    }

    public int[] retrieveIntArray() {
        return INT_ARRAYS[index];
    }

    public long[] retrieveLongArray() {
        return LONG_ARRAYS[index];
    }

    public short[] retrieveShortArray() {
        return SHORT_ARRAYS[index];
    }

    public Object[] retrieveObjectArray() {
        return OBJECT_ARRAYS[index];
    }

    public Object retrieveObject() {
        return OBJECTS[index];
    }

    public Class retrieveClass() {
        // todo: causes bump in measurements
        return OBJECTS[index].getClass();
    }

    public String retrieveString() {
        String ret = STRINGS[index];
        return ret;
    }

    public Throwable retrieveThrowable() {
        return THROWABLES[index];
    }

    public ByteBuffer retrieveDirectByteBuffer() {
        return BYTEBUFFER;
    }

    // -------------------------------------------------

    private static void generateAll() {
        int index = RANGE - 1;
        int size = MAXSIZE - DEFAULTSIZE;
        generateMax();
        while (index > -1) {
            BOOLEAN_ARRAYS[index] = Arrays.copyOf(BOOLEAN_ARRAYS[RANGE], size);
            CHAR_ARRAYS[index] = Arrays.copyOf(CHAR_ARRAYS[RANGE], size);
            BYTE_ARRAYS[index] = Arrays.copyOf(BYTE_ARRAYS[RANGE], size);
            INT_ARRAYS[index] = Arrays.copyOf(INT_ARRAYS[RANGE], size);
            LONG_ARRAYS[index] = Arrays.copyOf(LONG_ARRAYS[RANGE], size);
            SHORT_ARRAYS[index] = Arrays.copyOf(SHORT_ARRAYS[RANGE], size);
            DOUBLE_ARRAYS[index] = Arrays.copyOf(DOUBLE_ARRAYS[RANGE], size);
            FLOAT_ARRAYS[index] = Arrays.copyOf(FLOAT_ARRAYS[RANGE], size);

            OBJECT_ARRAYS[index] = Arrays.copyOf(OBJECT_ARRAYS[RANGE], size);

            OBJECTS[index] = OBJECT;
            THROWABLES[index] = THROWABLE;
            STRINGS[index] = STRING_BUILDER.substring(0, size);

            index--;
            size -= DEFAULTSIZE;
        }
    }

    private static void generateMax() {
        char c = 0;
        boolean b = true;
        byte by = 0;
        int v = 0;
        long l = 0;
        short s = 0;
        double d = 0;
        float f = 0;

        BOOLEAN_ARRAYS[RANGE] = new boolean[MAXSIZE];
        CHAR_ARRAYS[RANGE] = new char[MAXSIZE];
        BYTE_ARRAYS[RANGE] = new byte[MAXSIZE];
        INT_ARRAYS[RANGE] = new int[MAXSIZE];
        LONG_ARRAYS[RANGE] = new long[MAXSIZE];
        SHORT_ARRAYS[RANGE] = new short[MAXSIZE];
        DOUBLE_ARRAYS[RANGE] = new double[MAXSIZE];
        FLOAT_ARRAYS[RANGE] = new float[MAXSIZE];
        OBJECT_ARRAYS[RANGE] = new Object[MAXSIZE];

        for (int i = 0; i < MAXSIZE; i++) {
            STRING_BUILDER.append(c);

            BOOLEAN_ARRAYS[RANGE][i] = b;
            CHAR_ARRAYS[RANGE][i] = c;
            BYTE_ARRAYS[RANGE][i] = by;
            INT_ARRAYS[RANGE][i] = v;
            LONG_ARRAYS[RANGE][i] = l;
            SHORT_ARRAYS[RANGE][i] = s;
            DOUBLE_ARRAYS[RANGE][i] = d;
            FLOAT_ARRAYS[RANGE][i] = f;
            OBJECT_ARRAYS[RANGE][i] = OBJECT;
            OBJECTS[RANGE] = OBJECT;
            THROWABLES[RANGE] = THROWABLE;

            b = !b;
            by = (byte) ((by + 1) % Byte.MAX_VALUE);
            v = (v + 1) % Integer.MAX_VALUE;
            l = (l + 1) % Long.MAX_VALUE;
            s = (short) ((s + 1) % Short.MAX_VALUE);
            d = (d + 0.1);
            f = (f + 0.1f);
            c = (char) ((char) (c + '\u0001') % (char) Character.MAX_VALUE);
        }
```

```java
        STRINGS[RANGE] = STRING_BUILDER.substring(0, MAXSIZE);
    }

    private int index;
    private static boolean generated = false;

    private static final StringBuilder STRING_BUILDER = new StringBuilder();
    private static final Object OBJECT = new PermissionInfo();
    private static final Throwable THROWABLE = new Exception();

    private static final String[] STRINGS = new String[RANGE + 1];
    private static final Object[] OBJECTS = new PermissionInfo[RANGE + 1];
    private static final Throwable[] THROWABLES = new Exception[RANGE + 1];

    private static final boolean[][] BOOLEAN_ARRAYS = new boolean[RANGE + 1][];
    private static final byte[][] BYTE_ARRAYS = new byte[RANGE + 1][];
    private static final char[][] CHAR_ARRAYS = new char[RANGE + 1][];
    private static final double[][] DOUBLE_ARRAYS = new double[RANGE + 1][];
    private static final float[][] FLOAT_ARRAYS = new float[RANGE + 1][];
    private static final int[][] INT_ARRAYS = new int[RANGE + 1][];
    private static final long[][] LONG_ARRAYS = new long[RANGE + 1][];
    private static final short[][] SHORT_ARRAYS = new short[RANGE + 1][];
    private static final Object[][] OBJECT_ARRAYS = new Object[RANGE + 1][];

    private static final ByteBuffer BYTEBUFFER = ByteBuffer.allocateDirect
            (NIO_MAXSIZE);

}
```

BenchmarkRegistry.java

```java
package fi.helsinki.cs.tituomin.nativebenchmark;

import java.util.LinkedList;
import java.util.List;

import fi.helsinki.cs.tituomin.nativebenchmark.benchmark.JavaCounterparts;

public class BenchmarkRegistry {

    private static List<Benchmark> benchmarks;

    public static long repetitions;
    public static final long CHECK_INTERRUPTED_INTERVAL = 1000;

    public static List<Benchmark> getBenchmarks() {
        return benchmarks;
    }

    public static void init(long reps) throws ClassNotFoundException {
        repetitions = reps;
        benchmarks = new LinkedList<Benchmark>();
        Class jCounterparts = Class.forName("fi.helsinki.cs.tituomin" +
                ".nativebenchmark.benchmark.JavaCounterparts");
        Class threadClass = Class.forName("java.lang.Thread");
        initNative(reps, CHECK_INTERRUPTED_INTERVAL, jCounterparts,
                JavaCounterparts.INSTANCE, threadClass);
    }

    public static native void initNative(long repetitions, long interval,
                                         Class javaCounterparts,
                                         JavaCounterparts counterInstance,
                                         Class threadClass);

    public static native void setRepetitions(long repetitions);

    public static native void interruptNative();

    public static native void resetInterruptFlag();
}
```

BenchmarkResult.java

```
package fi.helsinki.cs.tituomin.nativebenchmark;

import android.util.Log;

import java.util.HashMap;
import java.util.Map;
import java.util.Map.Entry;


public class BenchmarkResult {

    public BenchmarkResult() {
        values = new String[ESTIMATED_CAPACITY];
        valueCount = 0;
    }

    public void put(String label, String value) {
        Integer labelIndex = labelIndexes.get(label);
        if (labelIndex == null) {
            lastIndex++;
            if (lastIndex > labels.length) {
                Log.e("BenchmarkResults", "Error, too many kinds of values, " +
                        "increase capacity!");
            }
            labels[lastIndex] = label;
```

```java
            labelIndexes.put(label, lastIndex);
            labelIndex = lastIndex;
        }
        values[labelIndex] = value;
        valueCount++;
    }

    public String get(String label) {
        Integer index = labelIndexes.get(label);
        if (index != null) {
            return values[index];
        } else {
            return null;
        }
    }

    public String get(int i) {
        return values[i];
    }

    public void putAll(BenchmarkResult other) {
        String[] otherValues = other.getValues();
        for (int i = 0; i < size(); i++) {
            if (otherValues[i] != null) {
                values[i] = otherValues[i];
                valueCount++;
            }
        }
    }

    public void putAll(Map<String, String> map) {
        for (Entry<String, String> entry : map.entrySet()) {
            put(entry.getKey(), entry.getValue());
        }
    }

    public String[] getValues() {
        return values;
    }

    public static String getLabel(int i) {
        return labels[i];
    }

    public static String[] labels() {
        return labels;
    }

    public boolean isEmpty() {
        return (valueCount == 0);
    }

    public static int size() {
        return (lastIndex + 1);
    }

    private static final int ESTIMATED_CAPACITY = 200;
    private String[] values;
    private int valueCount;

    private static String[] labels = new String[ESTIMATED_CAPACITY];
    private static Map<String, Integer> labelIndexes = new HashMap<String,
            Integer>(ESTIMATED_CAPACITY);
    private static int lastIndex = -1;
}
```

BenchmarkRunner.java

```
package fi.helsinki.cs.tituomin.nativebenchmark;

import android.os.SystemClock;
import android.util.Log;
import android.util.Pair;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.io.PrintWriter;
import java.lang.reflect.Method;
import java.lang.reflect.Modifier;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.Date;
import java.util.HashMap;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
import java.util.zip.ZipEntry;
import java.util.zip.ZipOutputStream;

import fi.helsinki.cs.tituomin.nativebenchmark.measuringtool.MeasuringTool;
import fi.helsinki.cs.tituomin.nativebenchmark.measuringtool.MeasuringTool
        .RunnerException;

import static fi.helsinki.cs.tituomin.nativebenchmark.Utils.colPr;

public enum BenchmarkRunner {
    INSTANCE; // singleton enum pattern

    private static final String SEPARATOR = ",";
    private static final String MISSING_VALUE = "-";
    private static final long WARMUP_REPS = 50000;
    private static BenchmarkParameter benchmarkParameter;
    private static List<MeasuringTool> measuringTools;
    private static int benchmarkCounter = 0;

    private static boolean interrupted = false;

    public static BenchmarkParameter getBenchmarkParameter() {
        if (benchmarkParameter == null) {
            benchmarkParameter = new BenchmarkParameter();
        }
        return benchmarkParameter;
    }

    private BenchmarkRunner() {
        this.allocatingRepetitions = -1;
    }

    public void initTools(ToolConfig conf, long repetitions, long
            allocRepetitions) throws IOException, InterruptedException {

        conf.setDefaultRepetitions(this.repetitions);
        conf.setDefaultRunAllBenchmarks(this.runAllBenchmarks);
        if (this.allocatingRepetitions > 0) {
            conf.setDefaultAllocRepetitions(this.allocatingRepetitions);
        }

        measuringTools = new ArrayList<MeasuringTool>();
        for (MeasuringTool tool : conf) {
            measuringTools.add(tool);
        }
    }

    private long repetitions;
    private long allocatingRepetitions;
    private CharSequence appRevision;
    private CharSequence appChecksum;
    private File cacheDir;
    private boolean runAllBenchmarks;
    private boolean runAtMaxSpeed;
    private String benchmarkSubstring;

    public enum BenchmarkSet {
        ALLOC,
        NON_ALLOC
    }

    ;
    private BenchmarkSet benchmarkSet;

```

```java
    public BenchmarkRunner setRepetitions(long x) {
        repetitions = x;
        return this;
    }

    public BenchmarkRunner setAllocatingRepetitions(long x) {
        allocatingRepetitions = x;
        return this;
    }

    public BenchmarkRunner setAppRevision(CharSequence x) {
        appRevision = x;
        return this;
    }

    public BenchmarkRunner setAppChecksum(CharSequence x) {
        appChecksum = x;
        return this;
    }

    public BenchmarkRunner setCacheDir(File x) {
        cacheDir = x;
        return this;
    }

    public BenchmarkRunner setRunAllBenchmarks(boolean x) {
        runAllBenchmarks = x;
        return this;
    }

    public BenchmarkRunner setRunAtMaxSpeed(boolean x) {
        runAtMaxSpeed = x;
        return this;
    }

    public BenchmarkRunner setBenchmarkSubstring(String x) {
        benchmarkSubstring = x;
        return this;
    }

    public BenchmarkRunner setBenchmarkSet(BenchmarkSet x) {
        benchmarkSet = x;
        return this;
    }

    public void runBenchmarks(ApplicationState mainUI, ToolConfig config,
                              File dataDir) {
        interrupted = false;

        try {
            BenchmarkRegistry.init(this.repetitions);
            // todo replace with config
            MeasuringTool.setDataDir(dataDir);
            Log.v(TAG, config.toString());
            initTools(config, this.repetitions, this.allocatingRepetitions);
        } catch (Exception e) {
            mainUI.updateState(ApplicationState.State.ERROR);
            Log.e("BenchmarkRunner", "Error initialising", e);
            return;
        }

        benchmarkParameter = getBenchmarkParameter();
        BenchmarkInitialiser.init(benchmarkParameter);

        List<Benchmark> allBenchmarks = BenchmarkRegistry.getBenchmarks();

        // todo enable
        long seed = System.currentTimeMillis();
        Log.i(TAG, String.format("Random seed %d", seed));
        java.util.Random random = new java.util.Random(seed);
        Collections.shuffle(allBenchmarks, random);
        try {
            Init.initEnvironment(true); // run warmup at max speed
        } catch (IOException e) {
            handleException(e, mainUI);
            return;
        }
        for (MeasuringTool tool : measuringTools) {
            if (tool == null) {
                return;
            }
            if (interrupted) {
                return;
            }

            List<Benchmark> benchmarks = new ArrayList<Benchmark>();

            if (this.benchmarkSubstring == null) {
                this.benchmarkSubstring = "";
            }
            String substringToApply = "";

            String filter = tool.getFilter();
            if (filter != null && !filter.equals("")) {
                substringToApply = tool.getFilter().toLowerCase();
            }
            this.benchmarkSubstring = substringToApply;

            for (Benchmark b : allBenchmarks) {
                boolean selected;
                if (tool.runAllBenchmarks()) {
                    selected = true;
                } else if (!substringToApply.equals("")) {
                    selected = (
                            b.getClass().getSimpleName().toLowerCase().indexOf(
                                    substringToApply) != -1);
                } else {
                    selected = (
                            b.representative() &&
                                    ((!b.isAllocating()) && this.benchmarkSet
                                            == BenchmarkSet.NON_ALLOC) ||
                                    (b.isAllocating() && this.benchmarkSet ==
                                            BenchmarkSet.ALLOC));
                }
                if (b.isNonvirtual() &&
                        b.from() != "C" &&
                        b.to() != "J") {
                    if (L.og) {
                        Log.i(TAG, String.format("skipping nonvirtual %s", b
                                .id()));
                    }
                    selected = false;
                }
                if (selected) {
                    benchmarks.add(b);
                }
            }
            int numOfBenchmarks = benchmarks.size();

            if (L.og) {
                Log.i(TAG, tool.getClass().getSimpleName());
            }

            if (!tool.ignore()) {
                // set the slower CPU frequency etc. after the warmup
                // round(s), taking less time
                if (!this.runAtMaxSpeed) {
                    try {
                        Init.initEnvironment(false);
                    } catch (IOException e) {
                        handleException(e, mainUI);
                        return;
                    }
                }
            }

            int max_rounds = tool.getRounds();
            String measurementID = Utils.getUUID();
            File resultFile = new File(dataDir, "benchmarks-" + measurementID
                    + ".csv");
            long startTime = SystemClock.uptimeMillis();
            Date start = new Date();
            long endTime = 0;

            int round = -1;
            boolean labelsWritten = false;

            ROUNDLOOP:
            while (++round < max_rounds) {
```

```java
                benchmarkCounter = 0;
                PrintWriter tempWriter = null;
                File tempFile = new File(this.cacheDir, "benchmarks-temp.csv");
                try {
                    tempWriter = Utils.makeWriter(tempFile, false);
                } catch (FileNotFoundException e) {
                    handleException(e, mainUI);
                    return;
                }


                List<BenchmarkResult> collectedData;

                try {
                    if (tool.explicitGC()) {
                        System.gc();
                        Thread.sleep(500);
                    }
                } catch (InterruptedException e) {
                    logE("Measuring thread was interrupted");
                    mainUI.updateState(
                            ApplicationState.State.INTERRUPTED);
                    interrupted = true;
                    break ROUNDLOOP;
                }
                int count = benchmarks.size();
                int j = 0;
                for (Benchmark benchmark : benchmarks) {
                    if (L.og) {
                        Log.i(TAG, (count - j) + " left");
                        Log.i(TAG, benchmark.getClass().getSimpleName());
                    }
                    ;
                    j++;
                    try {
                        collectedData = runSeries(benchmark, mainUI, tool,
                                round, max_rounds, count, j);
                    } catch (RunnerException e) {
                        logE("Exception was thrown", e.getCause());
                        mainUI.updateState(
                                ApplicationState.State.ERROR);
                        interrupted = true;
                        break ROUNDLOOP;
                    } catch (InterruptedException e) {
                        logE("Measuring thread was interrupted");
                        mainUI.updateState(
                                ApplicationState.State.INTERRUPTED);

                        interrupted = true;
                        break ROUNDLOOP;
                    }

                    if (collectedData.isEmpty() || tool.ignore()) {
                        continue;
                    }

                    endTime = SystemClock.uptimeMillis();

                    // print data
                    for (BenchmarkResult result : collectedData) {
                        for (int i = 0; i < BenchmarkResult.size(); i++) {
                            String value = result.get(i);
                            if (value == null) {
                                value = MISSING_VALUE;
                            }
                            tempWriter.print(value);
                            tempWriter.print(SEPARATOR);
                        }
                        tempWriter.println("");
                        tempWriter.flush();
                    }
                }
                endTime = SystemClock.uptimeMillis();
                tempWriter.close();

                if (tool.ignore()) {
                    continue; // todo test
                }

                InputStream in = null;
                OutputStream out = null;
                PrintWriter resultWriter = null;
                try {
                    resultWriter = Utils.makeWriter(resultFile, true);
                    // note: labels should be written last, after
                    // all possible keys have been created
                    if (!labelsWritten) {
                        String[] labels = BenchmarkResult.labels();
                        for (int i = 0; i < BenchmarkResult.size(); i++) {
                            resultWriter.print(labels[i] + SEPARATOR);
                        }
                        resultWriter.println("");
                        resultWriter.close();
                        labelsWritten = true;
                    }

                    in = new FileInputStream(tempFile);
                    out = Utils.makeOutputStream(resultFile, true);
                    Utils.copyStream(in, out);
                    //tempFile.delete();
                } catch (Exception e) {
                    mainUI.updateState(ApplicationState.State.ERROR);
                    Log.e("BenchmarkRunner", "Error writing results", e);
                    interrupted = true;
                    break ROUNDLOOP;
                } finally {
                    try {
                        if (in != null) {
                            in.close();
                        }
                        if (out != null) {
                            out.flush();
                            out.close();
                        }
                        if (resultWriter != null) {
                            resultWriter.close();
                        }
                    } catch (IOException e) {
                        mainUI.updateState(ApplicationState.State.ERROR);
                        Log.e("BenchmarkRunner", "Error closing files", e);
                        interrupted = true;
                        break ROUNDLOOP;
                    }
                }
            }

            if (!tool.ignore()) {
                // there has been at least one succesful round
                writeMeasurementMetadata(
                        new File(dataDir, "measurements.txt"),
                        measurementID,
                        (this.runAtMaxSpeed ? Init.CPUFREQ_MAX : Init.CPUFREQ),
                        startTime, endTime, start, tool, numOfBenchmarks,
                        round);

                List<String> filenames = tool.getFilenames();
                if (!filenames.isEmpty()) {
                    OutputStream os = null;

                    try {
                        File zip = new File(dataDir, "perfdata-" +
                                measurementID + ".zip");
                        os = Utils.makeOutputStream(zip, false);
                        Log.v("BenchmarkRunner", "filenames " + filenames
                                .size());
                        File measurementMetadataFile = new File(
                                dataDir, "benchmarks-" + measurementID + "" +
                                ".csv");
                        if (measurementMetadataFile.exists()) {
                            // TODO: actually make sure csv is written for
                            // incomplete runs
                            filenames.add(measurementMetadataFile
                                    .getAbsolutePath());
                        }
                        writeToZipFile(os, filenames, measurementID);
                        deleteFiles(filenames);
                    } catch (FileNotFoundException e) {
                        logE(e);
                    } catch (IOException e) {
                        logE("Error writing zip file.", e);
                    }
                    try {
                        if (os != null) {
                            os.close();
                        }
                    } catch (IOException e) {
                        logE("Error closing file.", e);
                    }
                }
            }

        }
        mainUI.updateState(
                ApplicationState.State.MEASURING_FINISHED);
    }

    private void writeMeasurementMetadata(
            File catalogFile, String measurementID, int cpuFreq,
            long startTime, long endTime, Date start, MeasuringTool tool,
            int numOfBenchmarks, int rounds) {

        Date end = new Date();
        PrintWriter c = null;
        try {
            c = Utils.makeWriter(catalogFile, true);

            c.println("");
            for (Pair<String, String> pair : tool.configuration()) {
                colPr(c, pair.first, pair.second);
            }
            colPr(c, "id", measurementID);
            colPr(c, "cpu-freq", cpuFreq);
            colPr(c, "logfile", LogAccess.filename());
            colPr(c, "rounds", rounds);
            colPr(c, "start", start);
            colPr(c, "end", end);
            colPr(c, "duration", Utils.humanTime(endTime - startTime));
            colPr(c, "benchmarks", numOfBenchmarks);
            colPr(c, "code-revision", this.appRevision);
            colPr(c, "code-checksum", this.appChecksum);
            colPr(c, "benchmark-set", this.benchmarkSet);
            colPr(c, "substring-filter", this.benchmarkSubstring);
            c.println("");
        } catch (IOException e) {
            logE(e);
        } finally {
            c.close();
        }
    }

    private final static String TAG = "BenchmarkRunner";

    private static void logE(String message, Throwable e) {
        Log.e(TAG, message, e);
    }

    private static void logE(Throwable e) {
        Log.e(TAG, "exception", e);
    }

    private static void logE(String msg) {
        Log.e(TAG, msg);
    }

    private static void deleteFiles(List<String> filenames) {
        for (String filename : filenames) {
            boolean success = new File(filename).delete();
            if (!success) {
                logE("Error deleting file " + filename);
            }
        }
    }

    private static void handleException(Exception e, ApplicationState UI) {
        logE(e);
        UI.updateState(ApplicationState.State.ERROR);
        return;
    }

    private static void
    writeToZipFile(OutputStream os, List<String> filenames, String mID)
            throws IOException {
        ZipOutputStream zos = new ZipOutputStream(os);
        final int byteCount = 512 * 1024;
        byte[] bytes = new byte[byteCount];
        for (String filename : filenames) {
            try {
                InputStream is = Utils.makeInputStream(filename);
                ZipEntry entry = new ZipEntry(mID + "/" + new File(filename)
                        .getName());
                int bytesRead = -1;
                zos.putNextEntry(entry);
                while ((bytesRead = is.read(bytes, 0, byteCount)) != -1) {
                    zos.write(bytes, 0, bytesRead);
                }
                zos.closeEntry();
            } finally {
                zos.flush();
            }
        }
        try {
            zos.close();
        } catch (IOException e) {
            logE(e);
        }
    }

    private static List<BenchmarkResult> runSeries(
            Benchmark benchmark, ApplicationState mainUI, MeasuringTool tool,
            int roundNo, int roundCount, int benchmarkCount, int benchmarkIndex)
            throws InterruptedException, RunnerException {

        List<BenchmarkResult> compiledMetadata = new
                ArrayList<BenchmarkResult>();

        if (Thread.interrupted()) {
            throw new InterruptedException();
        }
        BenchmarkParameter bPar = getBenchmarkParameter();
        BenchmarkResult introspected;
        try {
            introspected = inspectBenchmark(benchmark);
        } catch (ClassNotFoundException e) {
            Log.e("BenchmarkRunner", "Could not find class", e);
            return compiledMetadata;
        }

        String refTypesString = introspected.get("has_reference_types");
        boolean hasRefTypes = (refTypesString != null) && (refTypesString
                .equals("1"));

        String parameterCountString = introspected.get("parameter_count");
        int parameterCount = (parameterCountString == null) ? -1 : Integer
                .parseInt(parameterCountString);

        boolean dynamicParameters =
                benchmark.dynamicParameters() ||
                        (hasRefTypes && (-1 < parameterCount &&
                                parameterCount < 2));

        Iterator<Integer> iterator = bPar.iterator();
        Integer i;
        int j = -1;
        if (iterator.hasNext()) {
            i = iterator.next();
            j++;
        } else {
            i = null;
        }
        while (i != null) {
            if (Thread.interrupted()) {
                throw new InterruptedException();
            }
            bPar.setUp(); // (I) needs tearDown (see II)

            try {
                tool.startMeasuring(benchmark);
                benchmarkCounter++;
            } catch (IOException e) {
                logE("Measuring caused IO exception", e);
                if (mainUI.userWantsToRetry(e)) {
                    continue; // without incrementing i
                } else {
                    throw new InterruptedException("User wants to abort");
                }
            } finally {
                bPar.tearDown(); // (II) needs setUp (see I)
            }
            if (tool.explicitGC() && benchmarkCounter % 25 == 0) {
                System.gc();
                Thread.sleep(350);
            }

            String message = String.format(
                    "%s%s round %d/%d benchmark %d/%d range %d/%d",
                    tool.getClass().getSimpleName(),
                    tool.isWarmupRound() ? " (warmup)" : "",
                    roundNo + 1,
                    roundCount,
                    benchmarkIndex,
                    benchmarkCount,
                    j,
                    dynamicParameters ? BenchmarkParameter.RANGE : 1
            );
            mainUI.updateState(ApplicationState.State.MILESTONE, message);

            List<BenchmarkResult> measurements = tool.getMeasurements();
            for (BenchmarkResult measurement : measurements) {
                if (!measurement.isEmpty()) {
                    // todo: actual vs. requested size (objects etc.)
                    if (dynamicParameters) {
                        measurement.put("dynamic_size", "" + i);
                    } else {
                        measurement.put("dynamic_size", MISSING_VALUE);
                    }
                    measurement.put("id", benchmark.id());
                    measurement.put("class", benchmark.getClass()
                            .getSimpleName());
                    measurement.putAll(introspected);
                    compiledMetadata.add(measurement);
                }
            }
            // if parameter size can be varied, vary it - else break with
            // first size
            if (!dynamicParameters) {
                break;
            }
            if (iterator.hasNext()) {
                i = iterator.next();
                j++;
            } else {
                break;
            }
        }
        return compiledMetadata;
    }


    private static BenchmarkResult inspectBenchmark(Benchmark benchmark)
            throws ClassNotFoundException {
        BenchmarkResult bdata = new BenchmarkResult();
        int seqNo = benchmark.sequenceNo();
        String from = benchmark.from();
        String to = benchmark.to();
        bdata.put("no", "" + benchmark.sequenceNo());
        bdata.put("description", benchmark.description());
        bdata.put("from", from);
        bdata.put("to", to);
        bdata.put("representative", benchmark.representative() ? "1" : "0");
        bdata.put("nonvirtual", benchmark.isNonvirtual() ? "1" : "0");

        if (seqNo == -1) {
            bdata.put("custom", "1");
            return bdata;
        }

        Class c = Class.forName(
                String.format(
                        "fi.helsinki.cs.tituomin.nativebenchmark.benchmark" +
                                ".J2CBenchmark%05d",
                        seqNo));

        Method[] methods = c.getDeclaredMethods();
        for (int i = 0; i < methods.length; i++) {

            Method m = methods[i];
            int modifiers = m.getModifiers();

            if (Modifier.isNative(modifiers)) {
                return inspectMethod(m, bdata);
            }
        }
        return bdata;
    }

    private static BenchmarkResult inspectMethod(Method m, BenchmarkResult
            bdata) {

        Class[] parameters = m.getParameterTypes();
        List<Class> parameterList = new ArrayList<Class>(Arrays.asList
                (parameters));
        int modifiers = m.getModifiers();

        Map<String, Integer> parameterTypes = new HashMap<String, Integer>();
        for (Class param : parameterList) {
            Integer previousValue = null;
            String param_typename = param.getSimpleName();
            previousValue = parameterTypes.get(param_typename);
            parameterTypes.put(
                    param_typename,
                    (previousValue == null ? 1 : ((int) previousValue) + 1));
        }
        for (String typename : parameterTypes.keySet()) {
            bdata.put("parameter_type_" + typename + "_count", parameterTypes
                    .get(typename) + "");
        }

        Class returnType = m.getReturnType();

        boolean hasRefTypes = false;
        parameterList.add(returnType);
        for (Class cl : parameterList) {
            if (Object.class.isAssignableFrom(cl)) {
                hasRefTypes = true;
                break;
            }
        }

        bdata.put("has_reference_types", hasRefTypes ? "1" : "0");
        bdata.put("parameter_type_count", parameterTypes.keySet().size() + "");
        bdata.put("parameter_count", parameters.length + "");
        bdata.put("return_type", returnType.getCanonicalName());
        bdata.put("native_static", Modifier.isStatic(modifiers) ? "1" : "0");
        bdata.put("native_private", Modifier.isPrivate(modifiers) ? "1" : "0");
        bdata.put("native_protected", Modifier.isProtected(modifiers) ? "1" :
                "0");
        bdata.put("native_public", Modifier.isPublic(modifiers) ? "1" : "0");

        return bdata;
    }
}
```

BenchmarkSelector.java

```
package fi.helsinki.cs.tituomin.nativebenchmark;

//import fi.helsinki.cs.tituomin.nativebenchmark.BenchmarkInitialiser;
//import fi.helsinki.cs.tituomin.nativebenchmark.BenchmarkRegistry;
//import fi.helsinki.cs.tituomin.nativebenchmark.BenchmarkRunner;

import android.app.Activity;
import android.app.AlertDialog;
import android.app.Dialog;
import android.app.DialogFragment;
import android.app.Notification;
import android.app.NotificationManager;
import android.app.PendingIntent;
import android.app.TaskStackBuilder;
import android.content.Context;
import android.content.DialogInterface;
import android.content.Intent;
import android.content.res.Resources;
import android.media.RingtoneManager;
import android.net.Uri;
import android.os.Bundle;
import android.os.Environment;
import android.util.Log;
import android.view.View;
import android.view.WindowManager;
import android.widget.AdapterView;
import android.widget.AdapterView.OnItemSelectedListener;
import android.widget.ArrayAdapter;
import android.widget.Button;
import android.widget.Checkable;
import android.widget.NumberPicker;
import android.widget.Spinner;
import android.widget.TextView;

import java.io.File;
import java.io.InputStream;
import java.io.OutputStream;
import java.util.Map;

public class BenchmarkSelector extends Activity {
    /**
     * Called when the activity is first created.
     */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        this.requestWindowFeature(getWindow().FEATURE_NO_TITLE);
        setContentView(R.layout.main);

        this.retry = false;

        this.resultView = (TextView) findViewById(R.id.resultview);
        this.button = (Button) findViewById(R.id.mybutton);
        this.repView = (TextView) findViewById(R.id.repetitions);
        this.numPick = (NumberPicker) findViewById(R.id.picker_num);
        this.expPick = (NumberPicker) findViewById(R.id.picker_exp);

        NumberPicker.OnValueChangeListener listener = new RepsListener();

        numPick.setMinValue(1);
        numPick.setMaxValue(9);
        numPick.setValue(1);
```

```java
        expPick.setMinValue(0);
        expPick.setMaxValue(9);
        expPick.setValue(6);
        numPick.setOnValueChangedListener(listener);
        expPick.setOnValueChangedListener(listener);

        listener.onValueChange(numPick, 0, 0);

        if (getResources().getString(R.string.app_dirty).equals("1")) {
            this.resultView.setText(R.string.warning_changed);
        }

        final Resources resources = getResources();
        this.runner = BenchmarkRunner.INSTANCE
                .setAppChecksum(resources.getText(R.string.app_checksum))
                .setAppRevision(resources.getText(R.string.app_revision))
                .setCacheDir(getCacheDir());

        File sd = Environment.getExternalStorageDirectory();
        dataDir = new File(sd, "results");
        dataDir.mkdir();

        this.controller = new BenchmarkController(this, dataDir);

        this.socketCommunicator = new SocketCommunicator();
        //this.controller.addListener(socketCommunicator);

        // TODO: configuration is not UI specific.
        File configFile = new File(
                Environment.getExternalStorageDirectory(),
                "nativebenchmark_setup.json"
        );
        try {
            if (!configFile.exists()) {
                InputStream templateStream = getResources()
                        .openRawResource(R.raw.setup);
                OutputStream configFileStream = Utils.makeOutputStream
                        (configFile, false);
                Utils.copyStream(templateStream, configFileStream);
            }
            this.configuration = ToolConfig.readConfigFile();
        } catch (Exception e) {
            String msg = getResources().getString(R.string.config_error);
            Log.e(TAG, msg, e);
            displayMessage(ApplicationState.State.INIT_FAIL, msg);
        }

        if (this.configuration != null) {
            initSpinner(this.configuration);
            if (allocationArray == null) {
                // pre-enlarges the heap
                // commented out because
                // space is not reclaimed
                // even on gc
                allocationArray = new byte[1024 * 1024 * 100];
            }
        }
        this.socketCommunicator.startServer(this.controller, this.runner);
    }

    public void onDestroy() {
        socketCommunicator.stopServer();
        super.onDestroy();
    }

    private void initSpinner(Map<String, ToolConfig> conf) {
        Spinner spinner = (Spinner) findViewById(R.id.config_spinner);
        String keys[] = conf.keySet().toArray(new String[1]);
        ArrayAdapter<CharSequence> adapter = new ArrayAdapter(this, android.R
                .layout.simple_spinner_item, keys);

        int indexOfDefault = -1;
        while (++indexOfDefault < keys.length &&
                !keys[indexOfDefault].equals("default")) ;

        adapter.setDropDownViewResource(android.R.layout
                .simple_spinner_dropdown_item);
        spinner.setAdapter(adapter);
        spinner.setOnItemSelectedListener(new OnItemSelectedListener() {
            public void onItemSelected(
                    AdapterView<?> parent, View view,
                    int pos, long id) {
                selectedConfiguration = (String) parent.getItemAtPosition(pos);
            }

            public void onNothingSelected(AdapterView<?> parent) {
            }
        });

        spinner.setSelection(indexOfDefault);

    }

    public void displayMessage(ApplicationState.State state, String message) {
        displayMessage(state.stringId, message);
    }

    public void displayMessage(int id) {
        this.resultView.setText(id);
    }

    public void displayMessage(String message) {
        this.resultView.setText(message);
    }

    public void displayMessage(int id, String message) {
```

```java
        this.resultView.setText(getResources().getString(id) + " " + message);
    }

    private class StateChanger implements Runnable {
        public void run() {
            ApplicationState.DetailedState detailedState = controller
                    .getState();
            ApplicationState.State state = detailedState.state;
            String message = detailedState.message;

            if (message == null) {
                displayMessage(state.stringId);
            } else {
                displayMessage(state.stringId, message);
            }
            switch (state) {
                case MEASURING_STARTED:
                    getWindow().addFlags(WindowManager.LayoutParams
                            .FLAG_KEEP_SCREEN_ON);
                    expPick.setEnabled(false);
                    numPick.setEnabled(false);
                    switchButton(button);
                    state = ApplicationState.State.MILESTONE;
                    break;
                case MILESTONE:
                    break;
                case ERROR:
                    displayMessage(ApplicationState.State.ERROR, message);
                    // intended fallthrough
                case INTERRUPTED:
                    // intended fallthrough
                case MEASURING_FINISHED:
                    getWindow().clearFlags(WindowManager.LayoutParams
                            .FLAG_KEEP_SCREEN_ON);
                    stateThread.interrupt();
                    notifyFinished();
                    // intended fallthrough
                case INITIALISED:
                    resetButton(button);
                    numPick.setEnabled(true);
                    expPick.setEnabled(true);
                    break;
                case INIT_FAIL:
            }
        }
    }

    public boolean userWantsToRetry(final Exception e) {
        runOnUiThread(new Runnable() {
            public void run() {
                DialogFragment dialog = new RetryDialog(e.getMessage());
                dialog.show(getFragmentManager(), "foo");
            }
        });
        boolean waiting = true;
        while (waiting) {
            synchronized (this) {
                try {
                    this.wait();
                    waiting = false;
                } catch (InterruptedException ie) {
                    waiting = true;
                }
            }
        }
        return this.retry;
    }

    private void setRetry(boolean answer) {
        this.retry = answer;
        synchronized (this) {
            this.notify();
        }
    }

    private void resetButton(Button button) {
        button.setText(R.string.start_task);
        button.setOnClickListener(new View.OnClickListener() {
            public void onClick(View v) {
                startMeasuring(v);
            }
        });
    }

    private void switchButton(Button bt) {
        bt.setText(R.string.end_task);
        bt.setOnClickListener(
                new View.OnClickListener() {
                    public void onClick(View v) {
                        button.setText(R.string.end_task_confirmation);
                        button.setOnClickListener(
                                new View.OnClickListener() {
                                    public void onClick(View v) {
                                        displayMessage(ApplicationState.State
                                                .INTERRUPTING.stringId);
                                        controller.interruptMeasuring();
                                    }
                                });
                    }
                });
    }


    private boolean isChecked(int id) {
        return ((Checkable) findViewById(id)).isChecked();
    }

    private String textValue(int id) {
        return ((TextView) findViewById(id)).getText().toString();
    }

    public void startMeasuring(View view) {
        this.runner
                .setRepetitions(repetitions)
                .setAllocatingRepetitions(Long.parseLong(textValue(R.id
                        .alloc_reps)))
                .setBenchmarkSubstring(textValue(R.id.benchmark_substring)
                        .toLowerCase())
                .setRunAllBenchmarks(isChecked(R.id.checkbox_long))
                .setRunAtMaxSpeed(isChecked(R.id.checkbox_max))
                .setBenchmarkSet(isChecked(R.id.run_alloc) ?
                        BenchmarkRunner.BenchmarkSet.ALLOC :
                        BenchmarkRunner.BenchmarkSet.NON_ALLOC);

        stateThread = new Thread(
                new Runnable() {
                    public void run() {
                        while (!Thread.currentThread().isInterrupted()) {
                            runOnUiThread(new StateChanger());
                            try {
                                Thread.sleep(5000);
                            } catch (InterruptedException e) {
                                break;
                            }
                        }
                    }
                });

        stateThread.start();
        allocationArray = null;
        controller.startMeasuring(this.runner, this.configuration.get
                (selectedConfiguration));
    }

    private void notifyFinished() {
        Uri alarmSound = RingtoneManager.getDefaultUri(
                RingtoneManager.TYPE_NOTIFICATION);

        Notification.Builder mBuilder =
                new Notification.Builder(this)
                        .setSmallIcon(android.R.drawable.ic_media_play)
                        .setContentTitle(getResources().getText(R.string
                                .measuring_finished))
                        .setContentText(getResources().getText(R.string
                                .measuring_finished))
                        .setSound(alarmSound);

        // Creates an explicit intent for an Activity in your app
        Intent resultIntent = new Intent(this, BenchmarkSelector.class);

        // The stack builder object will contain an artificial back stack for
        // the
        // started Activity.
        // This ensures that navigating backward from the Activity leads out of
        // your application to the Home screen.
        TaskStackBuilder stackBuilder = TaskStackBuilder.create(this);
        // Adds the back stack for the Intent (but not the Intent itself)
        stackBuilder.addParentStack(BenchmarkSelector.class);
        // Adds the Intent that starts the Activity to the top of the stack
        stackBuilder.addNextIntent(resultIntent);
        PendingIntent resultPendingIntent =
                stackBuilder.getPendingIntent(
                        0,
                        PendingIntent.FLAG_UPDATE_CURRENT
                );
        //        mBuilder.setContentIntent(resultPendingIntent);
        NotificationManager mNotificationManager =
                (NotificationManager) getSystemService(Context
                        .NOTIFICATION_SERVICE);
        // mId allows you to update the notification later on.
        mNotificationManager.notify(1, mBuilder.build());
    }

    private class RepsListener implements NumberPicker.OnValueChangeListener {
        public void onValueChange(NumberPicker picker, int oldVal, int newVal) {
            long exp = BenchmarkSelector.this.expPick.getValue();
            long value = BenchmarkSelector.this.numPick.getValue();
            //            rounds = BenchmarkSelector.this.roundPick.getValue();
            while (exp-- > 0) {
                value *= 10;
            }
            repetitions = value;
            repView.setText("" + repetitions);
        }
    }

    private class RetryDialog extends DialogFragment {
        private String message;

        public RetryDialog(String message) {
            this.message = message;
        }

        @Override
        public Dialog onCreateDialog(Bundle savedInstanceState) {
            // Use the Builder class for convenient dialog construction
            AlertDialog.Builder builder = new AlertDialog.Builder(getActivity
                    ());
            builder.setMessage(getResources().getText(R.string
                    .retry_question) + ":\n" + this.message)
                    .setPositiveButton(R.string.retry_answer_positive, new
                            DialogInterface.OnClickListener() {
                                public void onClick(DialogInterface dialog,
                                                    int id) {
                                    setRetry(true);
                                }
                            })
                    .setNegativeButton(R.string.retry_answer_negative, new
                            DialogInterface.OnClickListener() {
                                public void onClick(DialogInterface dialog,
                                                    int id) {
                                    setRetry(false);
                                }
                            });
            // Create the AlertDialog object and return it
            return builder.create();
        }
    }

    private long repetitions;
    private boolean retry;
    private TextView textView, resultView, repView;
    private NumberPicker numPick, expPick;
    private Button button;
    private Thread stateThread;
    private static byte[] allocationArray;
    private File dataDir;
    private BenchmarkController controller;
    private SocketCommunicator socketCommunicator;
    private BenchmarkRunner runner;

    private String selectedConfiguration;

    private static final String TAG = "BenchmarkSelector";

    private Map<String, ToolConfig> configuration;

    static {
        System.loadLibrary("nativebenchmark");
    }

}
```

Init.java

```
package fi.helsinki.cs.tituomin.nativebenchmark;

import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

```

```java
public class Init {

    private static final String TAG = "NativeBenchmark";
    public static final int CPUFREQ = 400000;
    public static final int CPUFREQ_MAX = 1000000;

    public static List<String> initScript(int cpufreq) {
        List<String> script = new ArrayList<String>();
        script.add(
                "echo \"userspace\" > " +
                        "/sys/devices/system/cpu/cpu0/cpufreq" +
                        "/scaling_governor");
        script.add(
                "echo \"" + cpufreq + "\" > " +
                        "/sys/devices/system/cpu/cpu0/cpufreq" +
                        "/scaling_setspeed");
        return script;

    }

    public static void initEnvironment(boolean maxSpeed) throws IOException {
        ShellEnvironment.runAsRoot(initScript(maxSpeed ? CPUFREQ_MAX :
                CPUFREQ));
    }
}
```

L.java

```java
package fi.helsinki.cs.tituomin.nativebenchmark;

public class L {
    // Set L.og to false to statically remove
    // debugging logging statements from
    // measuring code.
    public static final boolean og = false;
}
```

LogAccess.java

```java
package fi.helsinki.cs.tituomin.nativebenchmark;

import android.util.Log;

import java.io.File;
import java.io.IOException;
import java.util.regex.Pattern;

public class LogAccess {

    private static final String LOGCAT_COMMAND =
            "logcat -f %s " +
                    "-b main -b system -b radio -b events " +
                    "-v time";

    public static void start(File dir) throws IOException {
        currentRunId = Utils.getUUID();
        mark(START, currentRunId);
        String filename = new File(dir, filename()).getAbsolutePath();
        String command = String.format(LOGCAT_COMMAND, filename);
        logcatProcess = Runtime.getRuntime().exec(command);
    }

    public static void end() {
        if (logcatProcess != null) {
            mark(END, currentRunId);
            logcatProcess.destroy();
            logcatProcess = null;
        }
    }

    private static void mark(String type, String id) {
        Log.println(LOGLEVEL, TAG, marker(type) + id);
    }

    private static String marker(String type) {
        return "[" + type + "] ";
    }

    public static String filename() {
        return "log-" + currentRunId + ".txt";
    }
    //06-27 23:52:37.348 I/LogAccess( 2378): [End]
    // 43ba52d0-61b0-47e4-991b-c98a3dd21f9f

    private static Pattern makeMarkerPattern(String type) {
        return Pattern.compile("[-:. [0-9]]+ I/" + TAG + "\\([ 0-9]+\\): \\["
                + type + "\\] " + currentRunId);
    }

    private static final int LOGLEVEL = Log.INFO;
    private static final String TAG = "LogAccess";
    private static final String START = "Start";
    private static final String END = "End";
    private static String currentRunId;
    private static Process logcatProcess;
}
```

measuringtool/AllocatingBenchmarkLongRunningWrapper.java

```java
package fi.helsinki.cs.tituomin.nativebenchmark.measuringtool;

import java.io.IOException;

import fi.helsinki.cs.tituomin.nativebenchmark.Benchmark;
import fi.helsinki.cs.tituomin.nativebenchmark.BenchmarkRegistry;

public class AllocatingBenchmarkLongRunningWrapper extends
        AllocatingBenchmarkWrapper {

    public AllocatingBenchmarkLongRunningWrapper(Benchmark b, long r) {
        super(b, r);
    }

    private static final long MAX_REPS = Long.MAX_VALUE;

    public void begin(MeasuringTool tool) throws InterruptedException,
            IOException {
        Benchmark benchmark = getBenchmark();
        init(benchmark);
        tool.putMeasurement("repetitions", this.repetitions + "");
        tool.start(this);
        tool.finishMeasurement();
    }

    public void run() {
        // This method ensures the garbage collector is run
        // every benchmark.maxrepetitions iteration
        // but otherwise the measurement is
        // run for a period long enough for profiling.
        Benchmark benchmark = getBenchmark();
        long interval = BenchmarkRegistry.CHECK_INTERRUPTED_INTERVAL;
        long division, remainder;
        long repetitions = MAX_REPS;

        division = repetitions / interval + 1;
        remainder = repetitions % interval + 1;

        while (--division != 0) {
            interval = BenchmarkRegistry.CHECK_INTERRUPTED_INTERVAL + 1;
            while (--interval != 0) {
                try {
                    benchmark.run();
                    System.gc();
                    Thread.sleep(GC_PAUSE_MS);
                } catch (InterruptedException e) {
                    setInterrupted();
                    return;
                } catch (Exception e) {
                    setException(e);
                    return;
                }
            }
            if (Thread.currentThread().isInterrupted()) {
                setInterrupted();
                return;
            }
        }

        while (--remainder != 0) {
            try {
                benchmark.run();
                System.gc();
                Thread.sleep(GC_PAUSE_MS);
            } catch (InterruptedException e) {
                setInterrupted();
                return;
            } catch (Exception e) {
                setException(e);
                return;
            }
        }

    }

}
```

measuringtool/AllocatingBenchmarkShortRunningWrapper.java

```java
package fi.helsinki.cs.tituomin.nativebenchmark.measuringtool;

import java.io.IOException;

import fi.helsinki.cs.tituomin.nativebenchmark.Benchmark;

public class AllocatingBenchmarkShortRunningWrapper extends
        AllocatingBenchmarkWrapper {

    public AllocatingBenchmarkShortRunningWrapper(Benchmark b, long r) {
        super(b, r);
    }

    private static final long MULTIPLIER = 25;

    public void begin(MeasuringTool tool) throws InterruptedException,
            IOException {
        Benchmark benchmark = getBenchmark();
        init(benchmark);
        long reps = MULTIPLIER;
        reps += 1;
        while (--reps != 0) {
            tool.putMeasurement("repetitions", this.repetitions + "");
            tool.putMeasurement("multiplier", MULTIPLIER + "");
            try {
                tool.start(getBenchmark());
                tool.finishMeasurement();
                System.gc();
                Thread.sleep(GC_PAUSE_MS);
            } catch (InterruptedException e) {
                setInterrupted();
                return;
            } catch (Exception e) {
                setException(e);
                return;
            }
        }
    }
}
```

measuringtool/AllocatingBenchmarkWrapper.java

```
package fi.helsinki.cs.tituomin.nativebenchmark.measuringtool;

import fi.helsinki.cs.tituomin.nativebenchmark.Benchmark;

public class AllocatingBenchmarkWrapper extends RunningWrapper {

    public AllocatingBenchmarkWrapper(Benchmark b, long repetitions) {
        super(b);
        this.repetitions = repetitions;
    }

    public static final int GC_PAUSE_MS = 400;

    public void init(Benchmark benchmark) {
        super.init(benchmark);
        benchmark.setRepetitions(repetitions);
    }

    protected long repetitions;

}
```

measuringtool/BasicOption.java

```java
package fi.helsinki.cs.tituomin.nativebenchmark.measuringtool;

import android.util.Pair;

public class BasicOption implements MeasuringOption {

    public BasicOption(OptionSpec type) {
        this(type, null);
    }

    public BasicOption(OptionSpec type, String value) {
        this.type = type;
        this.value = value;
    }

    // -----

    public OptionSpec id() {
        return this.type;
    }

    public void set(String value) {
        this.value = value;
    }

    public OptionSpec type() {
        return this.type;
    }

    public String value() {
        return value;
    }

    public Pair<String, String> toStringPair() {
        return new Pair<String, String>(this.type.id(), this.value);
    }

    public String toString() {
        return this.type() + " " + this.value();
    }

    // ----

    private OptionSpec type;
    private String value;

    // ----

}
```

measuringtool/CommandlineTool.java

```java
package fi.helsinki.cs.tituomin.nativebenchmark.measuringtool;

import android.util.Log;

import java.io.IOException;
import java.text.DateFormat;
import java.util.ArrayList;
import java.util.Date;
import java.util.LinkedList;
import java.util.List;
import java.util.Random;

import fi.helsinki.cs.tituomin.nativebenchmark.BenchmarkRegistry;
import fi.helsinki.cs.tituomin.nativebenchmark.ShellEnvironment;


public abstract class CommandlineTool extends MeasuringTool {

    // public CommandlineTool(int i, long reps, long allocreps) throws
    // IOException, InterruptedException {
    //     super(i, reps, allocreps);
    // }

    private static final long REPETITIONS = Long.MAX_VALUE;

    public CommandlineTool(int rounds, long repetitions, long allocreps,
                           boolean warmup, boolean x) throws IOException,
            InterruptedException {
        super(rounds, REPETITIONS, allocreps, false, x);
        this.filenames = new ArrayList<String>();
    }

    protected abstract String command();

    protected String formatParameter(MeasuringOption option) {
        throw new UnsupportedOptionException();
    }

    protected String formatDefaultParameter(MeasuringOption option) {
        if (option.type() == OptionSpec.COMMAND_STRING) {
            return option.value();
        } else {
            return formatParameter(option);
        }
    }

    public void initCommand() {
        List<String> commandline = new LinkedList<String>();
        commandline.addAll(generateCommandlineArguments());
        this.command = "";
        for (String s : commandline) {
            this.command += s + " ";
        }
    }

    protected List<OptionSpec> specifyAllowedOptions(List<OptionSpec> options) {
        options = super.specifyAllowedOptions(options);
        options.add(OptionSpec.COMMAND_STRING);
        return options;
    }

    protected List<MeasuringOption> defaultOptions(List<MeasuringOption>
                                                             options) {
        options.add(new BasicOption(OptionSpec.COMMAND_STRING, command()));
        return options;
    }

    protected void init() throws IOException, InterruptedException {
        super.init();
        if (!ShellEnvironment.runAsRoot(initScript())) {
            throw new IOException("Error executing as root.");
        }
    }

    protected abstract List<String> initScript();

    public boolean isLongRunning() {
        return true;
    }

    public void start(Runnable benchmark)
            throws InterruptedException, IOException {
        if (Thread.interrupted()) {
            throw new InterruptedException();
        }

        initCommand();
        BenchmarkRegistry.resetInterruptFlag();
        Thread benchmarkThread = new Thread(benchmark);
        Random r = new Random();
        int delay = r.nextInt(20);

        benchmarkThread.start();
        Thread.sleep(delay);

        try {
            ShellEnvironment.run(this.command);
        } catch (InterruptedException e) {
            throw e;
        } finally {
            benchmarkThread.interrupt();
            BenchmarkRegistry.interruptNative();
            benchmarkThread.join();
```

```java
        }
    }

    public void setFilename(String name, String path) {
        filenames.add(path + "/" + name);
        putMeasurement("Filename", name);
    }

    public List<String> getFilenames() {
        return filenames;
    }


    public void setUUID(String uuid) {
        putMeasurement("UUID", uuid);
    }

    // -----

    public List<String> generateCommandlineArguments() {
        List<String> result = new LinkedList<String>();
        for (OptionSpec type : allowedOptions) {
            MeasuringOption option = options.get(type);

            if (option == null) {
                Log.v("mt", type + " is null");
            } else {
                result.add(formatDefaultParameter(option));
            }
        }
        return result;
    }

    // -----

    private Date startDate;
    private Date endDate;
    private long startTime;
    private String command;

    private List<String> filenames;
    private static final DateFormat dateFormat = DateFormat
            .getDateTimeInstance();

}
```

measuringtool/JavaSystemNanoResponseTimeRecorder.java

```java
package fi.helsinki.cs.tituomin.nativebenchmark.measuringtool;

import java.io.IOException;

public class JavaSystemNanoResponseTimeRecorder extends ResponseTimeRecorder {

    public JavaSystemNanoResponseTimeRecorder(
            int i,
            long reps,
            long allocreps,
            boolean warmup,
            boolean runAllBenchmarks
    ) throws IOException, InterruptedException {
        super(i, reps, allocreps, warmup, runAllBenchmarks);
    }

    public void start(Runnable benchmark)
            throws InterruptedException, IOException {
        long endTime, startTime;
        startTime = System.nanoTime();
        benchmark.run();
        endTime = System.nanoTime();
        String delta = "" + (endTime - startTime);
        putMeasurement("response_time", delta);
        putMeasurement("time_unit", "nanoseconds");
    }
}
```

measuringtool/LinuxPerfRecordTool.java

```java
package fi.helsinki.cs.tituomin.nativebenchmark.measuringtool;

import java.io.File;
import java.io.IOException;
import java.util.LinkedList;
import java.util.List;

import fi.helsinki.cs.tituomin.nativebenchmark.Utils;

public class LinuxPerfRecordTool extends CommandlineTool {

    // public LinuxPerfRecordTool(int i, long reps, long allocreps) throws
    // IOException, InterruptedException {
    //     super(i, reps, allocreps);
    // }

    public LinuxPerfRecordTool
            (
                    int rounds,
                    long repetitions,
                    long allocreps,
                    boolean warmup,
                    boolean runAllBenchmarks) throws
            IOException, InterruptedException {
        super(rounds, repetitions, allocreps, warmup, runAllBenchmarks);
    }

    protected List<OptionSpec> specifyAllowedOptions(List<OptionSpec> options) {
        options = super.specifyAllowedOptions(options);
        options.add(OptionSpec.OUTPUT_FILEPATH);
        options.add(OptionSpec.MEASURE_LENGTH); // must be last
        return options;
    }

    protected List<String> initScript() {
        List<String> commands = new LinkedList<String>();
        commands.add("echo \"0\" > /proc/sys/kernel/kptr_restrict");
        commands.add("echo \"-1\" > /proc/sys/kernel/perf_event_paranoid");
        return commands;
    }

    protected List<MeasuringOption> defaultOptions(List<MeasuringOption>
                                                           options) {
        options = super.defaultOptions(options);
        String uuid = Utils.getUUID();
        String filename = generateFilename(uuid);
        String basePath = getPerfDir().getPath() + "/";
        setFilename(filename, basePath);
        setUUID(uuid);
        options.add(new BasicOption(OptionSpec.OUTPUT_FILEPATH, basePath +
                filename));
        return options;
    }

    protected void init() throws IOException, InterruptedException {
        super.init();
        getPerfDir().mkdir();
    }

    private File getPerfDir() {
        return new File(getDataDir(), "perf");
    }

    protected String command() {
        return "perf record -a -g";
    }

    private String generateFilename(String uuid) {
        return "perf-" + uuid + ".data";
    }

    public String formatParameter(MeasuringOption option) {
```

```java
        if (option.type() == OptionSpec.OUTPUT_FILEPATH) {
            return "--output=" + option.value();
        } else if (option.type() == OptionSpec.MEASURE_LENGTH) {
            return "sleep " + option.value();
        }
        return super.formatParameter(option);
    }
}
```

measuringtool/MeasuringOption.java

```java
package fi.helsinki.cs.tituomin.nativebenchmark.measuringtool;

import android.util.Pair;

public interface MeasuringOption {

    public void set(String value);

    public OptionSpec id();

    public String value();

    public OptionSpec type();

    public Pair<String, String> toStringPair();
}
```

measuringtool/MeasuringTool.java

```java
package fi.helsinki.cs.tituomin.nativebenchmark.measuringtool;


import android.util.Log;
import android.util.Pair;

import java.io.File;
import java.io.IOException;
import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Date;
import java.util.HashMap;
import java.util.HashSet;
import java.util.LinkedList;
import java.util.List;
import java.util.Locale;
import java.util.Map;
import java.util.Set;

import fi.helsinki.cs.tituomin.nativebenchmark.ApplicationState;
import fi.helsinki.cs.tituomin.nativebenchmark.Benchmark;
import fi.helsinki.cs.tituomin.nativebenchmark.BenchmarkRegistry;
import fi.helsinki.cs.tituomin.nativebenchmark.BenchmarkResult;

public abstract class MeasuringTool implements Runnable {

    public MeasuringTool
            (
                    int rounds,
                    long repetitions,
                    long allocRepetitions,
                    boolean warmup,
                    boolean runAllBenchmarks
            ) throws
            IOException, InterruptedException {
        clearMeasurements();
        specifyOptions();
        this.rounds = rounds;
        this.repetitions = repetitions;
        this.allocRepetitions = allocRepetitions;
        this.warmup = warmup;
        this.explicitGC = !warmup;
        this.runAllBenchmarks = runAllBenchmarks;
        init();
    }

    public static synchronized void userInterrupt() {
        userInterrupted = true;
        BenchmarkRegistry.interruptNative();
    }

    public static synchronized boolean userInterrupted() {
        if (userInterrupted == true) {
            userInterrupted = false;
            return true;
        }
        return false;
    }

    protected void init() throws IOException, InterruptedException {
    }

    protected abstract void start(Runnable benchmark)
            throws InterruptedException, IOException;

    public boolean isLongRunning() {
        return false;
    }

    public RunningWrapper wrap(Benchmark benchmark) {
        if (!benchmark.isAllocating()) {
            // Default running algorithm
            return new RunningWrapper(benchmark);
        } else {
            // the benchmark does allocations, we have
            // to limit the amount of loops -> compensate
            // by repeating the loop many times
            if (isLongRunning()) {
                return new AllocatingBenchmarkLongRunningWrapper(benchmark,
                        allocRepetitions);
            } else {
                return new AllocatingBenchmarkShortRunningWrapper(benchmark,
                        allocRepetitions);
            }
        }
    }

    public void startMeasuring(Benchmark benchmark) throws
            InterruptedException, IOException, RunnerException {
        String benchmarkName = benchmark.getClass().getSimpleName();
        clearMeasurements();
        setDefaultOptions();
        benchmark.setRepetitions(this.repetitions);
        RunningWrapper wrapper = wrap(benchmark);
        Date start = null, end = null;

        start = new Date();
        wrapper.begin(this);
        end = new Date();

        putMeasurement("start", DATEFORMAT.format(start));
        putMeasurement("end", DATEFORMAT.format(end));

        if (wrapper.wasInterrupted() && userInterrupted()) {
            throw new InterruptedException("Interrupted by user");
        }
        if (wrapper.exceptionWasThrown()) {
            throw new RunnerException(wrapper.getException());
        }
    }

    public class RunnerException extends Exception {
        public RunnerException(Throwable t) {
            super(t);
        }
    }


    public void run() {
        try {
            start(benchmark);
        } catch (InterruptedException e) {
            Log.e("BenchmarkRunner", "Measuring was interrupted ", e);
        } catch (IOException e) {
            Log.e("BenchmarkRunner", "IOException", e);
        }
    }

    public boolean explicitGC() {
        return this.explicitGC;
    }

    public boolean runAllBenchmarks() {
        return this.runAllBenchmarks;
    }

    public void setExplicitGC(boolean e) {
        this.explicitGC = e;
    }

    protected abstract List<MeasuringOption>
    defaultOptions(List<MeasuringOption> container);

    protected List<OptionSpec> specifyAllowedOptions(List<OptionSpec>
                                                         container) {
        return container;
    }

    protected void specifyOptions() {
        this.allowedOptions = specifyAllowedOptions(
                new LinkedList<OptionSpec>());

        if (this.requiredOptions == null) {
            this.requiredOptions = new HashSet<OptionSpec>();
        }
        for (OptionSpec o : this.allowedOptions) {
            if (o.required()) {
                this.requiredOptions.add(o);
            }
        }
    }

    protected void setDefaultOptions() {
        for (MeasuringOption op : defaultOptions(
                new LinkedList<MeasuringOption>())) {
            setOption(op);
        }
        ;
    }

    // -----

    public MeasuringTool set(String id, String value) {
        setOption(new BasicOption(OptionSpec.byId(id), value));
        return this;
    }

    public MeasuringTool set(OptionSpec spec, String value) {
        // todo: not typesafe (assumes basicoption)
        setOption(new BasicOption(spec, value));
        return this;
    }

    public String getOption(OptionSpec id) {
        return this.options.get(id).value();
    }

    public void setDescription(String d) {
        this.description = d;
    }

    public void setOption(MeasuringOption option) {
        if (this.allowedOptions == null) {
            specifyOptions();
        }
        if (!allowedOptions.contains(option.type())) {
            throw new UnsupportedOptionException();
        } else {
            this.options =
                    options != null ?
                            options :
                            new HashMap<OptionSpec, MeasuringOption>();

            this.options.put(option.type(), option);
        }
    }

    public void setBenchmark(Benchmark b) {
        benchmark = b;
    }

    public boolean ignore() {
        return false;
    }

    private boolean hasRequiredOptions() {
        return options.keySet().containsAll(requiredOptions);
    }

    protected List<OptionSpec> allowedOptions;
    protected Set<OptionSpec> requiredOptions;
    // currently support one value per option, change?
    protected Map<OptionSpec, MeasuringOption> options;

    private List<ApplicationState> observers;

    private BenchmarkResult currentMeasurement;
    private List<BenchmarkResult> measurements;

    protected long repetitions;

    protected void putMeasurement(String key, String value) {
        currentMeasurement.put(key, value);
    }

    protected void finishMeasurement() {
        currentMeasurement = new BenchmarkResult();
        measurements.add(currentMeasurement);
    }

    public List<BenchmarkResult> getMeasurements() {
        for (BenchmarkResult measurement : measurements) {
            if (this.options != null) {
                for (MeasuringOption op : options.values()) {
                    measurement.put(
                            op.toStringPair().first,
                            op.toStringPair().second);
                }
            }
        }
        return this.measurements;
    }

    public void clearMeasurements() {
        this.currentMeasurement = new BenchmarkResult();
        this.measurements = new LinkedList<BenchmarkResult>();
        measurements.add(currentMeasurement);
    }

    private static final List<String> emptyList = new ArrayList<String>();

    public List<String> getFilenames() {
        return emptyList;
    }

    public int getRounds() {
        return rounds;
    }

    public static void setDataDir(File dir) {
        dataDir = dir;
    }

    public static File getDataDir() {
        return dataDir;
    }

    public void setFilter(String substring) {
        filterSubstring = substring;
```

```java
    }

    public String getFilter() {
        return filterSubstring;
    }

    public List<Pair<String, String>> configuration() {
        List<Pair<String, String>> pairs = new ArrayList<Pair<String,
                String>>();
        pairs.add(new Pair<String, String>("tool", this.getClass()
                .getSimpleName()));
        pairs.add(new Pair<String, String>("repetitions", "" + this
                .repetitions));
        pairs.add(new Pair<String, String>("description", this.description));
        pairs.add(new Pair<String, String>("warmup", "" + this.warmup));
        if (options != null) {
            for (MeasuringOption opt : options.values()) {
                pairs.add(new Pair<String, String>(opt.type().id(), opt.value
                        ()));
            }
        }
        return pairs;
    }

    public boolean isWarmupRound() {
        return warmup;
    }

    private static File dataDir;

    private Benchmark benchmark;
    private int rounds;
    private long allocRepetitions;
    private String description;
    private String filterSubstring;
    protected boolean warmup;
    private boolean explicitGC;
    private boolean runAllBenchmarks;
    private static boolean userInterrupted = false;
    private final static String TAG = "MeasuringTool";

    public static class UnsupportedOptionException extends RuntimeException {
    }

    private static SimpleDateFormat DATEFORMAT = new SimpleDateFormat("MM-dd " +
            "HH:mm:ss.SSS", Locale.US);

}
```

measuringtool/MockCommandlineTool.java

```java
package fi.helsinki.cs.tituomin.nativebenchmark.measuringtool;

import java.io.IOException;
import java.util.List;

public class MockCommandlineTool extends CommandlineTool {

    public MockCommandlineTool(int i, long reps) throws IOException,
            InterruptedException {
        super(i, reps, reps, false, false);
    }

    protected List<String> initScript() {
        return null;
    }

    public boolean ignore() {
        return true;
    }

    protected String command() {
        return "cat /dev/null";
    }

}
```

measuringtool/OptionSpec.java

```java
package fi.helsinki.cs.tituomin.nativebenchmark.measuringtool;

import java.util.HashMap;
import java.util.Map;

public enum OptionSpec {

    COMMAND_STRING("Command run", "command_string", true),
    OUTPUT_FILEPATH("Output path", "output_filepath", true),
    MEASURE_LENGTH("Measuring time (sec)", "measure_length", true),
    VARIABLE("Variable parameter in benchmark", "variable", false),
    CPUFREQ("Fixed CPU frequency", "cpu_freq", true);

    OptionSpec(String name, String id, boolean required) {
        this.name = name;
        this.id = id;
        this.required = required;
        put(id, this);
    }

    public String id() {
        return id;
    }

    public boolean required() {
        return required;
    }

    public static OptionSpec byId(String id) {
        return specs.get(id);
    }

    private static void put(String id, OptionSpec value) {
        getSpecs().put(id, value);
    }

    public static Map<String, OptionSpec> getSpecs() {
        if (specs == null) {
            specs = new HashMap<String, OptionSpec>();
        }
        return specs;
    }

    private String name;
    private String id;
    private boolean required;

    private static Map<String, OptionSpec> specs;

}
```

measuringtool/PlainRunner.java

```java
package fi.helsinki.cs.tituomin.nativebenchmark.measuringtool;

import java.io.IOException;
import java.util.List;

public class PlainRunner extends MeasuringTool {

    public PlainRunner(int i, long reps, long allocreps) throws IOException,
            InterruptedException {
        super(i, reps, allocreps, true, false);
    }

    protected List<MeasuringOption> defaultOptions(List<MeasuringOption>
                                                           options) {
        return options;
    }

    protected List<OptionSpec> specifyAllowedOptions(List<OptionSpec> options) {
        options = super.specifyAllowedOptions(options);
        options.add(OptionSpec.CPUFREQ);
        return options;
    }

    public boolean explicitGC() {
        return false;
    }

    public boolean ignore() {
        return true;
    }

    public long repetitions() {
        return 10000;
    }

    public void start(Runnable benchmark)
            throws InterruptedException, IOException {
        benchmark.run();
    }
}
```

measuringtool/ResponseTimeRecorder.java

```java
package fi.helsinki.cs.tituomin.nativebenchmark.measuringtool;

import android.os.SystemClock;

import java.io.IOException;
import java.util.List;

public class ResponseTimeRecorder extends MeasuringTool {

    public ResponseTimeRecorder(
            int rounds,
            long reps,
            long allocreps,
            boolean warmup,
            boolean x)
            throws IOException, InterruptedException {
        super(rounds, reps, allocreps, warmup, x);
    }

    protected List<MeasuringOption> defaultOptions(List<MeasuringOption>
                                                              options) {
        // no options needed, time is returned as is (not in extra file)
        return options;
    }

    public boolean ignore() {
        return warmup;
    }

    public void start(Runnable benchmark)
            throws InterruptedException, IOException {
        long endTime, startTime;
        startTime = SystemClock.uptimeMillis();
        benchmark.run();
        endTime = SystemClock.uptimeMillis();
        String delta = "" + (endTime - startTime);
        putMeasurement("response_time", delta);
        putMeasurement("time_unit", "milliseconds");
    }

}
```

measuringtool/RunningWrapper.java

```java
package fi.helsinki.cs.tituomin.nativebenchmark.measuringtool;

import java.io.IOException;

import fi.helsinki.cs.tituomin.nativebenchmark.Benchmark;

public class RunningWrapper implements Runnable {
    private Benchmark benchmark;
    private Exception exceptionThrown;
    private boolean interrupted;
    private long repetitions;

    public RunningWrapper(Benchmark b) {
        benchmark = b;
    }

    protected void setException(Exception e) {
        exceptionThrown = e;
    }

    protected void setInterrupted() {
        interrupted = true;
    }

    public boolean exceptionWasThrown() {
        return exceptionThrown != null;
    }

    public void setRepetitions(long r) {
        repetitions = r;
    }

    public Benchmark getBenchmark() {
        return benchmark;
    }

    public long getRepetitions() {
        return repetitions;
    }

    public boolean wasInterrupted() {
        return interrupted;
    }

    public Exception getException() {
        return exceptionThrown;
    }

    public void init(Benchmark benchmark) {
        interrupted = false;
        exceptionThrown = null;
    }

    public void run() {
        benchmark.run();
    }

    public void begin(MeasuringTool tool) throws InterruptedException,
            IOException {
        init(benchmark);
        tool.start(this);
        if (Thread.currentThread().isInterrupted()) {
            setInterrupted();
        }
    }

}
```

MockObject.java

```
package fi.helsinki.cs.tituomin.nativebenchmark;

public class MockObject {

    public boolean jbooleanField;
    public byte jbyteField;
    public char jcharField;
    public double jdoubleField;
    public float jfloatField;
    public int jintField;
    public long jlongField;
    public short jshortField;
    public Object jobjectField;

    public static boolean jbooleanStaticField;
    public static byte jbyteStaticField;
    public static char jcharStaticField;
    public static double jdoubleStaticField;
    public static float jfloatStaticField;
    public static int jintStaticField;
    public static long jlongStaticField;
    public static short jshortStaticField;
    public static Object jobjectStaticField;

    public MockObject() {
        jbooleanField = true;
        jbyteField = 1;
        jcharField = 2;
        jdoubleField = 1.2;
        jfloatField = 3.1f;
        jintField = 3;
        jlongField = 4;
        jshortField = 5;
        jobjectField = this;

        jbooleanStaticField = true;
        jbyteStaticField = 1;
        jcharStaticField = 2;
        jdoubleStaticField = 1.2;
        jfloatStaticField = 3.1f;
        jintStaticField = 3;
        jlongStaticField = 4;
        jshortStaticField = 5;
        jobjectStaticField = this;
    }
}
```

ShellEnvironment.java

```
package fi.helsinki.cs.tituomin.nativebenchmark;

import android.util.Log;

import java.io.BufferedReader;
import java.io.DataOutputStream;
import java.io.IOException;
```

```java
import java.io.InputStream;
import java.io.InputStreamReader;
import java.util.List;

public class ShellEnvironment {

    // thanks http://muzikant-android.blogspot
    // .fi/2011/02/how-to-get-root-access-and-execute.html
    public static boolean runAsRoot(List<String> commands) throws IOException {
        if (commands == null) {
            return true;
        }
        boolean retval = false;
        DataOutputStream os = null;
        Process suProcess = null;
        try {
            if (null != commands && commands.size() > 0) {
                suProcess = Runtime.getRuntime().exec("su");

                os = new DataOutputStream(suProcess.getOutputStream());

                // Execute commands that require root access
                for (String currCommand : commands) {
                    os.writeBytes(currCommand + "\n");
                    os.flush();
                }

                os.writeBytes("exit\n");
                os.flush();

                try {
                    int suProcessRetval = suProcess.waitFor();
                    if (255 != suProcessRetval) {
                        // Root access granted
                        retval = true;
                    } else {
                        // Root access denied
                        retval = false;
                    }
                } catch (Exception ex) {
                    Log.e("ROOT", "Error executing root action", ex);
                }
            }
        } catch (IOException ex) {
            Log.w("ROOT", "Can't get root access", ex);
        } catch (SecurityException ex) {
            Log.w("ROOT", "Can't get root access", ex);
        } catch (Exception ex) {
            Log.w("ROOT", "Error executing internal operation", ex);
        } finally {
            if (suProcess != null) {
                suProcess.destroy();
            }
        }
        if (os != null) {
            os.close();
        }

        return retval;
    }

    public static void run(String command)
            throws IOException, InterruptedException {

        Process process = null;
        InputStream err = null;
        try {
            process = Runtime.getRuntime().exec(command);
            err = process.getErrorStream();
            process.waitFor();

            if (process.exitValue() != 0) {
                String line;
                BufferedReader br = new BufferedReader(new InputStreamReader
                        (err));
                StringBuilder sb = new StringBuilder("Command failed.\n");
                while ((line = br.readLine()) != null) {
                    Log.e("tm", line);
                    sb.append(line);
                    sb.append("\n");
                }
                process.destroy();
                br.close();
                throw new IOException(sb.toString());
            }
        } catch (IOException e) {
            throw e;
        } catch (InterruptedException e) {
            throw e;
        } finally {
            if (err != null) err.close();
            if (process != null) process.destroy();
        }
    }

}
```

SocketCommunicator.java

```java
package fi.helsinki.cs.tituomin.nativebenchmark;


```

```java
import android.util.Log;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.io.PrintWriter;
import java.net.InetSocketAddress;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketTimeoutException;
import java.util.Map;

public class SocketCommunicator implements ApplicationStateListener {
    /**
     * Thread to initialize Socket connection
     */
    private final Runnable InitializeConnection = new Thread() {
        @Override
        public void run() {
            // initialize server socket
            while (!Thread.currentThread().isInterrupted()) {
                try {
                    server = new ServerSocket();
                    //server.setSoTimeout(TIMEOUT * 1000);
                    server.setReuseAddress(true);
                    server.bind(new InetSocketAddress(38300));

                    //attempt to accept a connection
                    client = server.accept();

                    SocketCommunicator.this.out = client.getOutputStream();
                    SocketCommunicator.this.printWriter = new PrintWriter
                            (out, true);
                    SocketCommunicator.nis = client.getInputStream();
                    try {
                        SocketCommunicator.this.output(helpMessage);

                        byte[] bytes = new byte[1024];
                        int numRead = 0;
                        while (numRead >= 0) {
                            SocketCommunicator.this.output("[Awaiting input]");
                            numRead = SocketCommunicator.nis.read(bytes, 0,
                                    1024);
                            if (numRead < 1) {
                                executeCommand("quit");
                                return;
                            }
                            receivedCommand = new String(bytes, 0, numRead)
                                    .trim();
                            executeCommand(receivedCommand);
                        }
                    } catch (IOException ioException) {
                        Log.e(SocketCommunicator.TAG, "" + ioException);
                    }
                } catch (SocketTimeoutException e) {
                    receivedCommand = "Connection has timed out! Please try " +
                            "again";
                    Log.v(TAG, receivedCommand);
                } catch (IOException e) {
                    Log.e(SocketCommunicator.TAG, "" + e);
                }

                if (client != null) {
                    receivedCommand = "Connection was successful!";
                    try {
                        server.close();
                    } catch (IOException e) {
                        Log.e(TAG, "Error closing server connection.");
                    } finally {
                        server = null;
                    }
                    Log.v(TAG, receivedCommand);
                }
            }
        }
    };

    private class StateChanger implements Runnable {
        public void run() {
        }
    }

    private void executeStart(String command) {
        String[] split = command.split(":");
        if (split.length < 2) {
            Log.e(TAG, "No configuration key provided.");
            return;
        }
        String configKey = split[1];
        Map<String, ToolConfig> configurations = null;
        ToolConfig config = null;
        try {
            configurations = ToolConfig.readConfigFile();
            config = configurations.get(configKey);
        } catch (Exception e) {
            Log.e(TAG, "Error reading configuration file.", e);
        }
        if (config == null) {
            ApplicationState.DetailedState ds = new ApplicationState
                    .DetailedState(controller);
            ds.state = ApplicationState.State.ERROR;
            ds.message = "Could not find configuration for " + configKey;
            stateUpdated(ds);
            return;
        }
        stateThread = new Thread(
                new Runnable() {
                    public void run() {
                        while (!Thread.currentThread().isInterrupted()) {
                            ApplicationState.DetailedState detailedState =
                                    controller.getState();
                            ApplicationState.State state = detailedState.state;
                            stateUpdated(detailedState);
                            if (state == ApplicationState.State
                                    .MEASURING_FINISHED ||
                                    state == ApplicationState.State.ERROR) {
                                return;
                            }
                            try {
                                Thread.sleep(10000);
                            } catch (InterruptedException e) {
                                break;
                            }
                        }
                    }
                }
        );

        stateThread.start();
        this.controller.startMeasuring(this.runner, config);
    }

    private void output(String message) {
        synchronized (this) {
            this.printWriter.println(message);
        }
    }

    public void executeCommand(String command) {
        boolean executed = false;
        command = command.trim();
        if (command.length() == 0) {
            return;
        } else if (command.startsWith("start")) {
            this.executeStart(command);
            executed = true;
        } else if (command.startsWith("end")) {
            this.controller.interruptMeasuring();
            executed = true;
        } else if (command.startsWith("quit")) {
            this.restartServer();
            executed = true;
        } else {
            this.output(String.format("Unkown command %s", command));
            Log.v(TAG, "" + command + " == unknown command.");
        }
        if (!executed) {
            return;
        }
        this.output(String.format(
                "[Executed %s]", command));
    }

    public void startServer(
            BenchmarkController controller,
            BenchmarkRunner runner)

    {
        this.controller = controller;
        //this.controller.addListener(this, ApplicationState.State.MILESTONE);
        this.runner = runner;
        //initialize server socket in a new separate thread
        this.serverThread = new Thread(InitializeConnection);
        this.serverThread.start();
        String msg = "Attempting to connect";
        Log.v(TAG, msg);
    }

    public boolean stopServer() {
        try {
            // TODO
            // Close the opened resources on activity destroy
            this.output("Stopping socket server.");
            serverThread.interrupt();
            if (SocketCommunicator.nis != null) {
                SocketCommunicator.nis.close();
            }
            if (this.out != null) {
                this.out.close();
            }
            if (server != null) {
                server.close();
            }
            return true;
        } catch (IOException ec) {
            Log.e(SocketCommunicator.TAG, "Cannot close server socket" + ec);
            return false;
        }
    }

    public void restartServer() {
        if (this.stopServer()) {
            this.startServer(this.controller, this.runner);
        }
    }

    public void stateUpdated(ApplicationState.DetailedState state) {
        this.output(state.toString());
    }

    public static final String TAG = "SocketCommunicator";
    public static final int TIMEOUT = 10;
    private ServerSocket server = null;
    private Socket client = null;
    private OutputStream out;
    private PrintWriter printWriter;
    private String receivedCommand = null;
    public static InputStream nis = null;
    private BenchmarkController controller;
    private Map<String, ToolConfig> configurations;
    private BenchmarkRunner runner;
    private Thread serverThread;
    private Thread stateThread;

    private final String helpMessage = "\n" +
            "Measuring application ready.\n" +
            "Available commands:\n" +
            "  start :CONFIG_KEY\n" +
            "    Starts measuring with the configuration\n" +
            "    loaded from nativebenchmark_setup.json file\n" +
            "    under the top level key CONFIG_KEY.\n" +
            "  end\n" +
            "    Interrupts measuring.\n";

}
```

ToolConfig.java

```
package fi.helsinki.cs.tituomin.nativebenchmark;

import android.os.Environment;
import android.util.Log;

import org.json.JSONArray;
import org.json.JSONException;
import org.json.JSONObject;

import java.io.File;
import java.io.IOException;
import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.NoSuchElementException;

import fi.helsinki.cs.tituomin.nativebenchmark.measuringtool.MeasuringTool;

public class ToolConfig implements Iterable<MeasuringTool> {

    public static Map<String, ToolConfig> readConfigurations(String jsonConfig)
```

```java
            throws JSONException {
        JSONObject cfgObject = new JSONObject(jsonConfig);
        Map<String, ToolConfig> result = new HashMap<String, ToolConfig>();
        Iterator i = cfgObject.keys();
        while (i.hasNext()) {
            String key = (String) i.next();
            result.put(key, new ToolConfig(cfgObject.getJSONObject(key)));
        }
        return result;
    }

    public static Map<String, ToolConfig> readConfigFile() throws
            IOException, JSONException {
        String jsonContents = null;
        File configFile = new File(
                Environment.getExternalStorageDirectory(),
                "nativebenchmark_setup.json"
        );
        jsonContents = Utils.readFileToString(configFile);
        return ToolConfig.readConfigurations(jsonContents);
    }

    public ToolConfig(JSONObject job) {
        this.globalOptions = job;
        this.defaultAllocRepetitions = 400;
    }

    public String toString() {
        return this.globalOptions.toString();
    }

    public Iterator<MeasuringTool> iterator() {
        try {
            return new ToolIterator();
        } catch (JSONException e) {
            Log.e("ToolConfig", "Error reading json config", e);
        }
        return null;
    }

    private class ToolIterator implements Iterator<MeasuringTool> {
        private JSONArray toolArray;
        private int currentToolIndex;

        public ToolIterator() throws JSONException {
            currentToolIndex = -1;
            toolArray = globalOptions.getJSONArray("tools");
        }

        public boolean hasNext() {
            return currentToolIndex + 1 < toolArray.length();
        }

        public MeasuringTool next() {
            MeasuringTool tool = null;
            try {
                tool = createTool(toolArray.getJSONObject(++currentToolIndex));
                if (tool == null) {
                    throw new NoSuchElementException();
                }
            } catch (JSONException e) {
                Log.e("ToolConfig", "Error reading json config", e);
            }
            return tool;
        }

        public void remove() {
            throw new UnsupportedOperationException();
        }
    }

    private MeasuringTool createTool(JSONObject toolOptions) {

        MeasuringTool tool = null;
        try {
            long repetitions = toolOptions.optLong(
                    "repetitions", globalOptions.optLong(
                            toolOptions.optString("repetitions"), this
                                    .defaultRepetitions));

            // todo
            int defaultRounds = 1;
            // todo

            int rounds = toolOptions.optInt(
                    "rounds", globalOptions.optInt(
                            toolOptions.optString("rounds"), defaultRounds));

            long allocRepetitions = toolOptions.optLong(
                    "alloc_repetitions", globalOptions.optLong(
                            toolOptions.optString("alloc_repetitions"),
                            this.defaultAllocRepetitions));

            boolean warmup = toolOptions.optBoolean("warmup", false);

            boolean runAllBenchmarks = toolOptions.optBoolean(
                    "run_all", this.defaultRunAllBenchmarks);

            Class<?> _class = Class.forName(TOOL_PACKAGE + "." + toolOptions
                    .getString("class"));

            Constructor<?> ctor = _class.getConstructor(
                    Integer.TYPE, Long.TYPE, Long.TYPE, Boolean.TYPE, Boolean
                            .TYPE);

            Log.v("ToolConfig", "Tool instantiation " + rounds + " " +
```

```java
                    repetitions + " " + warmup);

            try {
                tool = (MeasuringTool) ctor.newInstance(
                        rounds, repetitions, allocRepetitions,
                        warmup, runAllBenchmarks);
            } catch (InvocationTargetException e) {
                Log.e("ToolConfig", "Constructor exception", e.getCause());
            }
            tool.setDescription(toolOptions.optString("description", ""));
            tool.setFilter(toolOptions.optString("filter", ""));
            tool.setExplicitGC(toolOptions.optBoolean("gc", !warmup));

            JSONObject options = toolOptions.optJSONObject("options");
            if (options != null) {
                Iterator keys = options.keys();
                while (keys.hasNext()) {
                    String key = (String) keys.next();
                    tool.set(key, options.getString(key));
                }
            }

        } catch (Exception e) {
            Log.e("ToolConfig", "Error instantiating tool", e);
            return null;
        }
        return tool;
    }

    public ToolConfig setDefaultRepetitions(long r) {
        defaultRepetitions = r;
        return this;
    }

    public ToolConfig setDefaultAllocRepetitions(long r) {
        defaultAllocRepetitions = r;
        return this;
    }

    public ToolConfig setDefaultRunAllBenchmarks(boolean r) {
        defaultRunAllBenchmarks = r;
        return this;
    }

    public BenchmarkRunner.BenchmarkSet getBenchmarkSet() {
        String key = globalOptions.optString("benchmark_set", "non_alloc");
        if (key.equals("alloc")) {
            return BenchmarkRunner.BenchmarkSet.ALLOC;
        }
        return BenchmarkRunner.BenchmarkSet.NON_ALLOC;
    }

    private long defaultRepetitions;
    private long defaultAllocRepetitions;
    private boolean defaultRunAllBenchmarks;

    private JSONObject globalOptions;
    private static final String TOOL_PACKAGE = "fi.helsinki.cs.tituomin" +
            ".nativebenchmark.measuringtool";
    private static final String TAG = "nativebenchmark.ToolConfig";

}
```

Utils.java

```java
package fi.helsinki.cs.tituomin.nativebenchmark;

import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.io.PrintWriter;
import java.nio.MappedByteBuffer;
import java.nio.channels.FileChannel;
import java.nio.charset.Charset;
import java.util.UUID;


public class Utils {

    public static String getUUID() {
        return UUID.randomUUID().toString();
    }

    public static String humanTime(long millis) {
        String time;
        long seconds = millis / 1000;
        long minutes = seconds / 60;
        long hours = minutes / 60;
        long seconds_total = seconds;
        seconds %= 60;
        minutes %= 60;
        return (hours + "h " +
                minutes + "m " +
                seconds + "s " +
                "(" + seconds_total + " s tot.)");
    }

    public static String colFmt(String label, String value) {
        return String.format("%20s: %s", label, value);
    }

    public static void colPr(PrintWriter p, String label, Object value) {
        p.println(colFmt(label, value.toString()));
    }

    public static void copyStream(InputStream in, OutputStream out) throws
            IOException {
        int count;
        while ((count = in.read(buffer, 0, BUFFERSIZE)) != -1) {
            out.write(buffer, 0, count);
        }
        out.flush();
    }

    public static PrintWriter
    makeWriter(File dir, String filename, boolean append)
            throws FileNotFoundException {
        return new PrintWriter(makeOutputStream(dir, filename, append));
    }

    public static PrintWriter
    makeWriter(File file, boolean append)
            throws FileNotFoundException {
        return new PrintWriter(makeOutputStream(file, append));
    }

    public static OutputStream
    makeOutputStream(File dir, String filename, boolean append)
            throws FileNotFoundException {
        return makeOutputStream(new File(dir, filename), append);
    }

    public static OutputStream
    makeOutputStream(File file, boolean append)
            throws FileNotFoundException {
        return new BufferedOutputStream(
                new FileOutputStream(
                        file, append));
    }

    public static InputStream
    makeInputStream(File file)
            throws FileNotFoundException {
        return new BufferedInputStream(new FileInputStream(file));
    }


    public static InputStream
    makeInputStream(String filename)
```

```
            throws FileNotFoundException {
        return makeInputStream(new File(filename));
    }

    public static String readFileToString(File file) throws IOException {
        FileInputStream stream = new FileInputStream(file);
        try {
            FileChannel fc = stream.getChannel();
            MappedByteBuffer bb = fc.map(FileChannel.MapMode.READ_ONLY, 0, fc
                    .size());
            return Charset.defaultCharset().decode(bb).toString();
        } finally {
            stream.close();
        }
    }

    private static final int BUFFERSIZE = 128 * 1024;
    private static byte[] buffer = new byte[BUFFERSIZE];
}
```

Python-komponentit (koodingenerointi)

Hakemistossa script/.

benchmark_generator.py

```python
from templates import java_benchmark
from templates import java_counterparts
from templates import c_module
from templates import c_jni_function
from templates import c_nativemethod

from templating import put

import itertools
import logging

import jni_types
from jni_types import types, primitive_types, return_types

packagename = [
    'fi',
     'helsinki',
     'cs',
     'tituomin',
     'nativebenchmark',
     'benchmark']
library_name = 'nativebenchmark'
java_counterpart_classname = 'JavaCounterparts'
native_method_name = 'nativemethod'
class_counter = 0

# todo: initialize complex return values for c side


def next_sequence_no():
    global class_counter
    class_counter += 1
    return str(class_counter).zfill(5)


def benchmark_classname(prefix, number):
    return prefix + 'Benchmark' + number


def parameter_initialisation(language, typespec, name):
    if typespec.get('is-array', False):
        if language == 'java':
            expression = 'benchmarkParameter.retrieve{_type}Array()'.format(
                _type=typespec['java-element-type'].capitalize())
        else:
            expression = '{_type}ArrayValue'.format(
                _type=typespec['c-element-type'])

    elif typespec.get('is-object', False):
        if language == 'java':
            expression = 'benchmarkParameter.retrieve{_type}()'.format(
                _type=typespec['java'])
        else:
            expression = '{_type}Value'.format(
                _type=typespec['c'])

    elif language == 'java':
        if typespec.get('java-literal'):
            expression = typespec['java-literal']
        else:
            expression = ''

    elif language == 'c':
        if typespec.get('c-literal'):
            expression = typespec['c-literal']
        else:
            expression = ''

    if name:
        return name + " = " + expression
    else:
        return expression


def modifier_combinations():
    privacy = ['public', 'private', 'protected']
    static = ['static', '']
    return list(itertools.product(privacy, static))


def method_combinations():
    combinations = []
    combinations.append({
        'description': 'no arguments or return types',
            'representative': True,
            'return_types': [return_types['v']],
            'target_modifiers': modifier_combinations(),
            'types': []})

    for symbol, _type in types.iteritems():
        combinations.append({
            'description': 'varying count {0}'.format(symbol),
            'representative': _type.get('representative', False),
            'return_types': [return_types['v']],
            'target_modifiers': [('public', '')],
            'types': jni_types.type_combinations(
                size=20,
                typeset=[types[symbol]])})

    # Start from 1 to avoid duplicates.
    combinations.append({
        'description': 'vary number of types',
            'representative': True,
            'skip': 1,
            'return_types': [return_types['v']],
            'target_modifiers': [('public', '')],
            'types': jni_types.type_combinations(
                typeset=types.values())
    })

    combinations.append({
        'description': 'modifier combinations',
            'representative': True,
            'return_types': [return_types['l']],
            'target_modifiers': modifier_combinations(),
            'include_nonvirtual': True,
            'types': [types['i']]
    })

    filtered_return_types = filter(
        lambda x: x['symbol'] != 'l',
        jni_types.type_combinations(typeset=types.values()))

    combinations.append({
        'description': 'return types',
            'return_types': filtered_return_types,
            'target_modifiers': [('public', '')],
            'types': [types['i']]
    })

    return combinations


def generate_benchmarks():
    global class_counter
    java = []
    java_callees = {}
    c_implementations = []
    c_runners = []
    c_methodid_inits = []

    for spec in method_combinations():
        virtualities = [True]
        if spec.get('include_nonvirtual'):
            virtualities.append(False)
        for virtualcall in virtualities:
            for target_modifier in spec['target_modifiers']:
                for return_type in spec['return_types']:

                    if 'representative' not in spec:
                        representative = return_type.get(
                            'representative', False)
                    else:
                        representative = spec['representative']

                    representative = "true" if representative else "false"

                    type_combination = spec['types']

                    native_method_modifiers = " ".join(target_modifier)
                    return_expression = parameter_initialisation(
                        'c', return_type, None)

                    # Declare/initialize the parameters that are passed
                    # to the called function/method.

                    parameter_names = []

                    parameter_declarations = []
                    parameter_initialisations = []
                    c_parameter_declarations = []
                    c_parameter_initialisations = []

                    if target_modifier[1] == 'static':
                        c_parameter_declarations.append('jclass cls')
                    else:
                        c_parameter_declarations.append('jobject instance')

                    for i, type_data in enumerate(type_combination):
                        parameter_names.append(
                            type_data['symbol'] + str(i + 1))

                        parameter_declarations.append(
                            type_data['java'] + ' ' + parameter_names[-1])
                        c_parameter_declarations.append(
                            type_data['c'] + ' ' + parameter_names[-1])
                        parameter_initialisations.append(
                            parameter_initialisation(
                                'java', type_data, parameter_names[-1])){
                        c_parameter_initialisations.append(
                            parameter_initialisation(
                                'c', type_data, parameter_names[-1]))

                    skip = spec.get('skip', 0)
                    upper_bound = len(type_combination){
                    if upper_bound == 0:
                        upper_bound = 1
                    for i in range(skip, upper_bound):

                        sequence_no = next_sequence_no()

                        for from_lang, to_lang in [
                                ('J', 'C'), ('J', 'J'),
    (’C’, ’C’), (’C’, ’J’)]:

pairing = (from_lang, to_lang)

if virtualcall == False:
    # Nonvirtual calls only apply to C -> J
    # instance methods J2C is generated as a proxy
    # to get benchmark metadata
    if (target_modifier[1] == ’static’ or
            pairing in [(’J’, ’J’), (’C’, ’C’)]):
            continue

# 1. Set up call targets.

if to_lang == ’C’:
    counterpart_method_name = ’nativemethod’
if to_lang == ’J’:
    counterpart_method_unqualified = (
        ’benchmark’ + sequence_no)

if pairing == (’J’, ’J’):
    if target_modifier[0] == ’private’:
        # A private method cannot
        # be called from Java.
        # Don’t generate a benchmark.
        continue

    if target_modifier[1] == ’static’:
        clsname = java_counterpart_classname
    else:
        clsname = ’counterpartInstance’
    counterpart_method_name = (
        clsname + ’.’ +
        counterpart_method_unqualified)
    native_method = ’’

if pairing == (’C’, ’J’):
    counterpart_method_name = ’benchmark’ + \
        sequence_no

# 2. Set up calling sources.

if from_lang == ’C’:
    run_method = java_benchmark.native_run_method_t
    native_method = ’’

if from_lang == ’J’:
    run_method = put(
        java_benchmark.java_run_method_t,
        parameter_declarations="; ".join(
            parameter_declarations[0:i + 1]),
        parameter_initialisations="; ".join(
            parameter_initialisations[0:i + 1]),
        counterpart_method_name=(
            counterpart_method_name),
        counterpart_method_arguments=(
            ", ".join(parameter_names[0:i + 1])))

if pairing == (’J’, ’C’):
    native_method = put(
        java_benchmark.native_method_t,
        modifiers=native_method_modifiers,

        return_type=return_type[
            ’java’],
        name=native_method_name,
        parameters=", ".join(
            parameter_declarations[0:i + 1]))

# 3. Common benchmark class wrapper for all
# benchmarks.

classname = benchmark_classname(
    ’2’.join(pairing), sequence_no)

nm = native_method if to_lang == ’C’ else ’’
java.append({
    ’filename’: classname + ".java",
        ’class’:
            ’.’.join(
                packagename) + "." + classname,
        ’path’: ’/’.join(packagename),
        ’code’: put(
            java_benchmark.t,
            representative=representative,
            imports=’’,
            has_dynamic_parameters=’false’,
            # todo
            is_allocating=’false’,
            is_nonvirtual=(
                ’false’ if virtualcall else ’true’),
            _id=benchmark_classname(
                "", sequence_no),
            description=spec[
                ’description’],
            seq_no=class_counter,
            from_language=from_lang,
            to_language=to_lang,
            class_relations=’’,
            packagename=’.’.join(
            packagename),
            classname=classname,
            library_name=library_name,
            run_method=run_method,
            native_method=nm})
if (return_type.get('is-object') or
    return_type.get('is-array')):
    ret_expression = "{_type}Value".format(
        _type=return_type['c'])
else:
    ret_expression = parameter_initialisation(
        'java', return_type, None)

# 4. Call target implementations.

if to_lang == 'J':
    cmu = counterpart_method_unqualified
    java_callees[cmu] = put(
        java_counterparts.counterpart_t,
            return_type=return_type[
                'java'],
            privacy=target_modifier[0],
            static=target_modifier[1],
            methodname=cmu,
            parameters=", ".join(
                parameter_declarations[0:i + 1]),
            return_expression=ret_expression
    )

if pairing == ('J', 'C'):
    c_implementations.append(
        put(c_nativemethod.t,
            return_type=return_type['c'],
            packagename=(
            '_').join(
            packagename),
            classname=classname,
            function=native_method_name,
            parameters=", ".join(
            c_parameter_declarations[0:i + 2]),
            return_expression=return_expression))

if pairing == ('C', 'C'):
    jni_param_names = ['env', 'instance']
    jni_param_names.extend(parameter_names)

    c_runners.append(
        put(c_nativemethod.t_caller_native,
            packagename='_'.join(packagename),
            classname=classname,
            parameter_declarations="; ".join(
                c_parameter_declarations[1:i + 2]),
            parameter_initialisations="; ".join(
                c_parameter_initialisations[
                    0:i + 1]),
            counterpart_method_name=(
                "Java_{}_{}_{}".format(
                    ('_').join(packagename),
                                    (’J2CBenchmark’ +
                                     str(sequence_no)),
                                    native_method_name),
                            counterpart_method_arguments=", ".join(
                                jni_param_names[0:i + 3])))

                if pairing == (’C’, ’J’):
                    if return_type.get(’is-object’,
                                       return_type.get(
                                           ’is-array’, False)):
                        java_method_type = ’Object’
                    else:
                        java_method_type = return_type[
                            ’java’].capitalize()

                    arguments = ’, ’.join(parameter_names[0:i + 1])
                    if arguments != ’’:
                        arguments = ’, ’ + arguments

                    c_runners.append(
                        c_nativemethod.call_java_from_c(
                            static=(
                                target_modifier[1] == ’static’),
                            nonvirtual=not virtualcall,
                            seq_no=class_counter,
                            packagename=’_’.join(packagename),
                            classname=classname,
                            parameter_declarations="; ".join(
                                c_parameter_declarations[1:i + 2]),
                            parameter_initialisations="; ".join(
                                c_parameter_initialisations[
                                    0:i + 1]),
                            java_method_type=java_method_type,
                            call_variant=’’,  # todo test variants?
                            arguments=arguments))

                    c_methodid_inits.append(
                        put(c_module.mid_init_t,
                            seq_no=class_counter,
                            static=target_modifier[
                                1].capitalize(),
                            method_name=counterpart_method_name,
                            method_descriptor=(
                                jni_types.method_descriptor(
                                    return_type,
                                    type_combination[0:i + 1]))))

ref_types = jni_types.object_types.values(
) + jni_types.array_types.values()
jcp_decl = ’’
jcp_init = ’’
for _type in ref_types:
jcp_decl += "private static {_type1} {_type2}Value;\n".format(
_type1=_type['java'], _type2=_type['c'])
jcp_init += parameter_initialisation(
'java', _type, _type['c'] + 'Value') + ";\n"

java_counterparts_class = {
'filename': java_counterpart_classname + ".java",
'class': '.'.join(
    packagename) +
"." +
java_counterpart_classname,
'path': '/'.join(packagename),
'code': put(java_counterparts.t,
            packagename='.'.join(packagename),
            imports='',
            return_value_declarations=jcp_decl,
            return_value_inits=jcp_init,
            counterpart_methods=''.join(java_callees.values()))}

c_file = put(
c_module.t,
jni_function_templates=''.join(c_implementations),
initialisers='')

c_runners_file = put(
c_module.t,
jni_function_templates=''.join(c_runners),
initialisers=put(
c_module.initialisers_t,
mid_inits=''.join(c_methodid_inits),
amount_of_methods=class_counter))

return {'java': java,
'java_counterparts': java_counterparts_class,
'c': c_file,
'c_runners': c_runners_file}
```

jni_types.py

```
import itertools

primitive_types = None
object_types = None
other_types = None
types = None
return_types = None

representative_types = None

array_types = None
```

```

primitive_type_definitions = [
    {
        'symbol': 'b',
        'java': 'boolean',
        'c': 'jboolean',
        'c-literal': '1',
        'java-literal': 'true',
        'jvm-desc': 'Z',
        'byte_count': '1'
    },

    {
        'symbol': 'y',
        'java': 'byte',
        'c': 'jbyte',
        'c-literal': "'a'",
        'java-literal': '(byte)100',
        'jvm-desc': 'B',
        'byte_count': '1',
        'representative': True,
        # todo: same value?
    },

    {
        'symbol': 'c',
        'java': 'char',
        'c': 'jchar',
        'c-literal': '12',
        'java-literal': "'\u0012'",
        'jvm-desc': 'C',
        'byte_count': '2'
    },

    {
        'symbol': 's',
        'java': 'short',
        'c': 'jshort',
        'c-literal': '101',
        'java-literal': '(short)101',
        'jvm-desc': 'S',
        'byte_count': '2'
    },

    {
        'symbol': 'i',
        'java': 'int',
        'c': 'jint',
        'c-literal': '102',
        'java-literal': '102',
        'jvm-desc': 'I',
        'representative': True,
        'byte_count': '4'
    },

    {
        'symbol': 'l',
        'java': 'long',
        'c': 'jlong',
        'c-literal': '103',
        'java-literal': '103',
        'jvm-desc': 'J',
        'representative': True,
        'byte_count': '8'
    },

    {
        'symbol': 'f',
        'java': 'float',
        'c': 'jfloat',
        'c-literal': '104.1',
        'java-literal': '104.1f',
        'jvm-desc': 'F',
        'representative': True,
        'byte_count': '4'
    },

    {
        'symbol': 'd',
        'java': 'double',
        'c': 'jdouble',
        'c-literal': '105.1',
        'java-literal': '105.1',
        'jvm-desc': 'D',
        'representative': True,
        'byte_count': '8'
    },
]

object_type_definitions = [

    {
        'symbol': 'O',
        'java': 'Object',
        'package': 'java.lang',
        'c': 'jobject',
        'c-literal': None,
        'java-literal': None,
        'is-object': True,
        'representative': True,
        'jvm-desc': 'Ljava/lang/Object;'
    },

    {
        'symbol': 'C',
        'java': 'Class',
        'package': 'java.lang',
        'c': 'jclass',
        'c-literal': None,
        'java-literal': None,
        'is-object': True,
        'jvm-desc': 'Ljava/lang/Class;'
    },

    {
        'symbol': 'S',
        'java': 'String',
        'package': 'java.lang',
        'c': 'jstring',
        'c-literal': None,
        'java-literal': '"a string"',
        'is-object': True,
        'jvm-desc': 'Ljava/lang/String;'
    },

    {
        'symbol': 'T',
        'java': 'Throwable',
        'package': 'java.lang',
        'c': 'jthrowable',
        'c-literal': None,
        'java-literal': None,
        'is-object': True,
        'jvm-desc': 'Ljava/lang/Throwable;'
    }

]

other_type_definitions = [

    {
        'symbol': 'v',
        'java': 'void',
        'c': 'void',
        'c-literal': None,
        'java-literal': None,
        'representative': True,
        'jvm-desc': 'V'
    }
]


def java_native_methodname(is_static, returntype, parametertypes):
    ret = "_"
    if is_static:
        ret += "st_"
    ret += types.get(returntype)['java'] + "_"
    for t in parametertypes:
        ret += types.get(t)
```

```python


def java_native_methodsignature(is_static, returntype, parametertypes):
    ret = "private"
    if is_static:
        ret += "static "

    # todo here


def type_combinations(size=0, typeset=None):
    if size == 0:
        size = len(typeset)

    return list(itertools.islice(itertools.cycle(typeset), 0, size))


def method_descriptor(return_type, parameter_types):
    return "({parameters}){returndesc}".format(
        parameters=''.join([td['jvm-desc'] for td in parameter_types]),
        returndesc=return_type['jvm-desc'])


def init_types():
    global primitive_types, object_types, other_types, types
    global return_types, array_types, representative_types

    primitive_types = dict([(typedef['symbol'], typedef)
                            for typedef in primitive_type_definitions])
    object_types = dict([(typedef['symbol'], typedef)
                         for typedef in object_type_definitions])
    other_types = dict([(typedef['symbol'], typedef)
                        for typedef in other_type_definitions])

    array_element_types = {}
    array_element_types.update(primitive_types)
    array_element_types['O'] = object_types['O']

    # todo here
    array_types = dict(
        [
            ('A' + key,
             {'symbol': 'A' + key,
              'java': tipe['java'] + '[]',
              'package': tipe.get('package', None),
              'c': tipe['c'] + 'Array',
              'c-literal': None,
              'java-literal': None,
              'is-array': True,
              'representative': tipe.get('representative', False),
              'java-element-type': tipe['java'],
              'c-element-type': tipe['c'],
              'jvm-desc': '[' + tipe['jvm-desc']
```

```
                })
            for key, tipe in array_element_types.iteritems()])

    types = dict()
    types.update(primitive_types)
    types.update(object_types)
    types.update(array_types)

    return_types = dict()
    return_types.update(types)
    return_types.update(other_types)

#    types.update(other_types)

init_types()
```

make_benchmarks.py

```
from benchmark_generator import generate_benchmarks, packagename
from make_custom_benchmarks import write_custom_benchmarks
from templates import java_registry_init
from templating import put

import sys
from sys import argv
import os.path
import os
import logging

# Log everything, and send it to stderr.
logging.basicConfig(level=logging.DEBUG)


def write_benchmark(benchmark, java_output_dir):
        java_output_path = os.path.join(
            java_output_dir,
            benchmark["path"])

        try:
            os.makedirs(java_output_path)
        except OSError:
            pass

        java_output = open(
            os.path.join(
                java_output_path,
                benchmark["filename"]), 'w')

        java_output.write(benchmark["code"])
```

```


def write_benchmarks(c_output, c_runners_output, java_output_dir):
    benchmarks = generate_benchmarks()

    c_output.write(benchmarks['c'])
    c_runners_output.write(benchmarks['c_runners'])

    write_benchmark(benchmarks['java_counterparts'], java_output_dir)
    for benchmark in benchmarks['java']:
        write_benchmark(benchmark, java_output_dir)

    return [benchmark['class'] for benchmark in benchmarks['java']]


def write_benchmark_initialiser(classes):
    benchmark_inits = []

    for _class in classes:
        benchmark_inits.append(java_registry_init.inits(_class))

    path = os.path.join(
        java_output_dir,
        'fi/helsinki/cs/tituomin/nativebenchmark',
        'BenchmarkInitialiser.java')

    init_output = open(path, 'w')
    init_output.write(
        put(java_registry_init.t,
            register_benchmarks="\n".join(benchmark_inits)))


if __name__ == "__main__":
    try:
        argv.pop(0)
        c_output_name = argv.pop(0)
        c_run_output_name = argv.pop(0)
        c_custom_output_name = argv.pop(0)
        java_output_dir = argv.pop(0)
        c_definition_filename = argv.pop(0)
        java_definition_filename = argv.pop(0)

        definition_files = {
            'C': open(c_definition_filename),
                'J': open(java_definition_filename)}

        c_run_output = open(c_run_output_name, 'w')
        c_output = open(c_output_name, 'w')

        classes = (write_benchmarks(c_output, c_run_output, java_output_dir) +
                   write_custom_benchmarks(
                           definition_files,
                           c_custom_output_name,
```

```python
                             java_output_dir))

        write_benchmark_initialiser(classes)
        print(",".join(classes))
    except Exception as e:
        logging.exception("Exception was thrown.")
        sys.exit(1)
    else:
        sys.exit(0)
```

make_custom_benchmarks.py

```python
import re
import logging
from os import path
from sys import argv
import sys

from templating import put

from templates import arrays
from templates import loop_code
from templates import c_nativemethod
from templates import java_benchmark
from templates import java_registry_init

import jni_types

# Log everything, and send it to stderr.
logging.basicConfig(level=logging.DEBUG)

MAX_ALLOC_REPETITIONS = 500

i = re.IGNORECASE
begin_re = re.compile('\s*//\s*@begin\s*', flags=i)
end_re = re.compile('\s*//\s*@end\s*', flags=i)
inits_re = re.compile('\s*//\s*@inits-end\s*', flags=i)
benchmark_re = re.compile('\s*//\s*@(\S+)\s*')


def inits_block_end(line):
    return inits_re.match(line)


def begins_block(line):
    return begin_re.match(line)


def ends_block(line):
    return end_re.match(line)


def is_benchmark_header(line):
    return benchmark_re.match(line)


def parse_benchmark_header(line):
    tokens = line.split()[1:]
    b_properties = parse_properties(tokens[1:])
    b_properties['id'] = tokens[0][1:]
    return b_properties


def parse_properties(seq):
    kvs = []
    for s in seq:
        splitted = s.split('=')
        kvs.append((splitted[0], splitted[1]))
    try:
        return dict(kvs)
    except ValueError as e:
        print seq
        print seq[0].split('=')
        exit(1)


def abort_if_last(line):
    if line == '':
        logging.error("Invalid benchmark input file.")
        exit(1)


def read_until(f, predicate, collect=None):
    line = ''
    while not predicate(line):
        if collect is not None:
            collect.append(line)
        line = f.readline()
        abort_if_last(line)
    abort_if_last(line)
    return line


def read_benchmarks(definition_files):
    benchmarks = {}
    for lang, f in definition_files.iteritems():
        benchmarks[lang] = {'module': None, 'benchmarks': []}

        module_start = []
        inits = []
        read_until(f, begins_block, collect=module_start)
        read_until(f, inits_block_end, collect=inits)
        benchmarks[lang]['inits'] = ''.join(inits)
        benchmarks[lang]['module'] = ''.join(module_start)
        line = read_until(f, is_benchmark_header)

        while line != '':
            bm_props = parse_benchmark_header(line)

            bm_code = []
            line = read_until(f,
                lambda x: ends_block(x) or is_benchmark_header(x),
                collect=bm_code)

            bm_props['code'] = ''.join(bm_code)
            benchmarks[lang]['benchmarks'].append(bm_props)

            if ends_block(line):
                break

    add_field_and_array_benchmarks(benchmarks)
    add_overhead_benchmarks(benchmarks)
    return benchmarks

OVERHEAD_STEP = 2
OVERHEAD_STEPS = 11  # incl. zero
OVERHEAD_CODE_STATEMENT = "__a = (((__a * __a * __a) / __b) + __b) / __a;\n"


def add_overhead_benchmark(benchmarks, i, prefix, alloc):
    overhead_code = []
    for j in range(0, i):
        overhead_code.append(OVERHEAD_CODE_STATEMENT)

    benchmark = {
        'code': ''.join(overhead_code),
        'id': prefix + 'Overhead' + str(i).zfill(5),
        'description': i
    }

    if alloc:
        benchmark['alloc'] = True

    c_b = benchmark.copy()
    double_benchmark = benchmark.copy()
    # double the amount of work for java (uses optimizations unlike c)
    double_benchmark['code'] = ''.join(overhead_code + overhead_code)
    j_b = double_benchmark
    c_b['direction'] = 'cj'
    j_b['direction'] = 'jj'
    benchmarks['C']['benchmarks'].append(c_b)
    benchmarks['J']['benchmarks'].append(j_b)


def add_overhead_benchmarks(benchmarks):
```

```python
    for i in range(1, OVERHEAD_STEPS * OVERHEAD_STEP, OVERHEAD_STEP):
        for prefix, alloc in [('Alloc', True), ('Normal', False)]:
            add_overhead_benchmark(benchmarks, i, prefix, alloc)
    add_overhead_benchmark(benchmarks, 200, 'Warmup', False)


def macro_call(template, _type):
    return template.format(
        _type=_type['c'],
        java_type_name=_type['java'].capitalize())


def make_id(template, _type):
    return template.format(
        _type=_type['java'].capitalize())


def add_field_and_array_benchmarks(benchmarks):
    c = benchmarks['C']['benchmarks']
    java = benchmarks['J']['benchmarks']

    for _type in (
            jni_types.primitive_types.values() +
            [jni_types.object_types['O']]):
        representative = _type.get('representative', False)

        c.append({
                'id': make_id('GetStatic{_type}Field', _type),
                'representative': representative,
                'direction': 'cj',
                'code': macro_call(
                    'GET_STATIC_TYPE_FIELD({_type}, {java_type_name});',
                    _type)})

        java.append({
                'id': make_id('GetStatic{_type}Field', _type),
                'representative': representative,
                'direction': 'jj',
                'class_init':
                    'public {} persistentValue;'.format(_type['java']),
                'method_init': '{} localPersistentValue = {};'.format(
                    _type['java'], _type.get('java-literal') or 'objectValue'),
                'code':
                    ("localPersistentValue = "
                     "mockObject.{_ctype}StaticField;"
                    ).format(
                        _javatype=_type['java'],
                        _ctype=_type['c']
                    ),
                'finished': 'persistentValue = localPersistentValue;'
                })
        # todo: separate inits from global inits
        # and make side-effect real

        c.append({
                'direction': 'cj',
                'representative': representative,
                'id': make_id('SetStatic{_type}Field', _type),
                'code': macro_call(
                    'SET_STATIC_TYPE_FIELD({_type}, {java_type_name});',
                    _type)})

        java.append({
                'id': make_id('SetStatic{_type}Field', _type),
                'representative': representative,
                'direction': 'jj',
                'code':
                    "mockObject.{_ctype}StaticField = {_literal} ;".format(
                        _ctype=_type['c'],
                        _literal=_type.get('java-literal') or 'objectValue'

                    ),
                })

        c.append({
                'id': make_id('Get{_type}Field', _type),
                'direction': 'cj',
                'representative': representative,
                'code': macro_call(
                    'GET_TYPE_FIELD({_type}, {java_type_name});',
                    _type)})

        java.append({
                'id': make_id('Get{_type}Field', _type),
                'representative': representative,
                'class_init':
                    'public {} persistentValue;'.format(_type['java']),
                'method_init': '{} localPersistentValue = {};'.format(
                    _type['java'], _type.get('java-literal') or 'objectValue'),
                'direction': 'jj',
                'code':
                    "localPersistentValue = mockObject.{_ctype}Field;".format(
                        _javatype=_type['java'],
                        _ctype=_type['c']

                    ),
                'finished': 'persistentValue = localPersistentValue;'
                })

        c.append({
                'id': make_id('Set{_type}Field', _type),
                'direction': 'cj',
                'representative': representative,
                'code': macro_call(
                    'SET_TYPE_FIELD({_type}, {java_type_name});',
                    _type)})

        java.append({
                'id': make_id('Set{_type}Field', _type),
                'representative': representative,
                'direction': 'jj',
                'code':
                    "mockObject.{_ctype}Field = {_literal} ;".format(
                        _ctype=_type['c'],
                        _literal=_type.get('java-literal') or 'objectValue'

                    ),
                })

    for _type in jni_types.primitive_types.values():
        representative = _type.get('representative', False)
        c.append({
                'id': make_id('New{_type}Array', _type),
                'representative': representative,
                'direction': 'cj',
                'vary': 'size',
                'alloc': 'true',
                'code': macro_call(
                    'NEW_PRIMITIVE_ARRAY({_type}, {java_type_name});',
                    _type)
                })

        # java

        c.append({
                'id': make_id('Get{_type}ArrayElements', _type),
                'representative': representative,
                'direction': 'cj',
                'vary': 'size',
                'code': macro_call(
                    ('GET_PRIMITIVE_ARRAY_ELEMENTS({_type}, {java_type_name});'
                    'RELEASE_PRIMITIVE_ARRAY_ELEMENTS'
                     '({_type}, {java_type_name});'),
                    _type)})

        c.append({
                'vary': 'size',
                'direction': 'cj',
                'representative': representative,
                'id': make_id('Get{_type}ArrayRegion', _type),
                'code': macro_call(
                    'GET_PRIMITIVE_ARRAY_REGION({_type}, {java_type_name});',
                    _type)})

        c.append({
                'vary': 'size',
                'representative': representative,
                'direction': 'cj',
                'id': make_id('Set{_type}ArrayRegion', _type),
                'code': macro_call(
                    'SET_PRIMITIVE_ARRAY_REGION({_type}, {java_type_name});',
                    _type)})

        c.append({
                'vary': 'size',
                'representative': representative,
                'direction': 'cj',
                'id': make_id('Get{_type}ArrayLength', _type),
                'code': macro_call(
                    'GET_PRIMITIVE_ARRAY_LENGTH({_type});',
                    _type)})

        c.append({
                'vary': 'size',
                'representative': representative,
                'direction': 'cc',
                'id': make_id('ReadComplete{_type}Array', _type),
                'code': put(
                    arrays.t_read,
                    declare_idx='jint idx;',
                    variable_in='%s__IN' % _type['c'],
                    array_variable='%s_buf__IN' % _type['c'],
                    element_literal=_type['c-literal']
                    )})

        java_declarations = ('{0} {0}In;\n{0} [] {0}Arr = '
                             'benchmarkParameter.retrieve{1}Array();').format(
            _type['java'], _type['java'].capitalize())

        java.append({
                'vary': 'size',
                'direction': 'jj',
                'representative': representative,
                'class_init': 'public int persistentValue;',
                'method_init': 'int localPersistentValue = 0;',
                'id': make_id('ReadComplete{_type}Array', _type),
                'code': put(
                    arrays.t_read,
                    declare_idx='int idx;',
                    declare_variables=java_declarations,
                    variable_in='%sIn' % _type['java'],
                    array_variable='%sArr' % _type['java'],
                    element_literal=_type['java-literal']
                    ),
                'finished': 'persistentValue = localPersistentValue;'
                })

        c.append({
                'vary': 'size',
                'direction': 'cc',
                'representative': representative,
                'id': make_id('WriteComplete{_type}Array', _type),
                'code': put(
                    arrays.t_write,
                    declare_idx='jint idx;',
                    # todo: writing affects other tests?
                    array_variable='%s_buf__IN' % _type['c'],
                    element_literal=_type['c-literal']
                    )})

        java.append({
                'vary': 'size',
                'direction': 'jj',
                'representative': representative,
                'class_init': 'public int persistentValue;',
                'method_init': 'int localPersistentValue = 0;',
                'id': make_id('WriteComplete{_type}Array', _type),
                'code': put(
                    arrays.t_write,
                    declare_variables=java_declarations,
                    declare_idx='int idx;',
                    # todo: writing affects other tests?
                    array_variable='%sArr' % _type['java'],
                    element_literal=_type['java-literal']),
                'finished': 'persistentValue = localPersistentValue;'
                })

        # NIO variations of array/buffer reading/writing
        if _type['java'] == 'boolean':
            # Not available for booleans
            continue

        if _type['java'] == 'byte':
            uppercase_typename = ''
        else:
            uppercase_typename = _type['java'].title()

        # Read with hardcoded type method
        java.append({
            'vary': 'size',
            'direction': 'jj',
            'representative': True,
            'class_init': 'public int persistentValue;',
            'method_init': put(
                arrays.t_init_nio,
                type_declarations=java_declarations),
            'id': make_id('ReadComplete{_type}NioByteBuffer', _type),
            'code': put(
                arrays.t_read_nio,
                declare_idx='int idx;',
                variable_in='%sIn' % _type['java'],
                array_variable='directByteBufferValue',
                type_name=uppercase_typename,
                element_literal=_type['java-literal']),
            'finished': 'persistentValue = localPersistentValue;'
        })
        # Write with hardcoded type method
        java.append({
            'vary': 'size',
            'direction': 'jj',
            'representative': True,
            'class_init': 'public int persistentValue;',
            'method_init': put(
                arrays.t_init_nio,
                type_declarations=java_declarations),
            'id': make_id('WriteComplete{_type}NioByteBuffer', _type),
            'code': put(
                arrays.t_write_nio,
                declare_variables='',
                declare_idx='int idx;',
                array_variable='directByteBufferValue',
                type_name=uppercase_typename,
                element_literal=_type['java-literal']),
            'finished': 'persistentValue = localPersistentValue;'
        })

        declaration = java_declarations + "\n"
        if _type['java'] == 'byte':
            array_variable = 'directByteBufferValue'
        else:
            # Views are only relevent for non-byte types.
            declaration += ('java.nio.{0}Buffer bufferView = '
                            'directByteBufferValue.as{0}Buffer();').format(
                uppercase_typename)
            array_variable = 'bufferView'

        # Bulk read through a typecast view buffer
        java.append({
            'vary': 'size',
            'direction': 'jj',
            'representative': True,
            'class_init': 'public int persistentValue;',
            'method_init': put(
                arrays.t_init_nio,
                type_declarations=declaration),
            'id': make_id('ReadBulk{_type}NioByteBufferView', _type),
            'code': put(
                arrays.t_bulk_read,
                array_variable=array_variable,
                array_in='%sArr' % _type['java']),
            'finished': 'persistentValue = localPersistentValue;'
        })

        # Bulk write through a typecast view buffer
        java.append({
            'vary': 'size',
            'direction': 'jj',
            'representative': True,
            'class_init': 'public int persistentValue;',
            'method_init': put(
                arrays.t_init_nio,
                type_declarations=declaration),
            'id': make_id('WriteBulk{_type}NioByteBufferView', _type),
            'code': put(
                arrays.t_bulk_write,
                array_variable=array_variable,
                array_in='%sArr' % _type['java']),
            'finished': 'persistentValue = localPersistentValue;'
        })

        if _type['java'] == 'byte':
            continue

        # Read through a typecast view buffer
        java.append({
            'vary': 'size',
            'direction': 'jj',
            'representative': True,
            'class_init': 'public int persistentValue;',
            'method_init': put(
                arrays.t_init_nio,
                type_declarations=declaration),
            'id': make_id('ReadComplete{_type}NioByteBufferView', _type),
            'code': put(
                arrays.t_read_nio_as_view,
                declare_idx='int idx;',
                variable_in='%sIn' % _type['java'],
                array_variable='bufferView',
                type_name=uppercase_typename,
                element_literal=_type['java-literal']),
            'finished': 'persistentValue = localPersistentValue;'
        })
        # Write through a typecast view buffer
        java.append({
            'vary': 'size',
            'direction': 'jj',
            'representative': True,
            'class_init': 'public int persistentValue;',
            'method_init': put(
                arrays.t_init_nio,
                type_declarations=declaration),
            'id': make_id('WriteComplete{_type}NioByteBufferView', _type),
            'code': put(
                arrays.t_write_nio_as_view,
                declare_idx='int idx;',
                array_variable='bufferView',
                element_literal=_type['java-literal']),
            'finished': 'persistentValue = localPersistentValue;'
        })


```

```
def write_custom_benchmarks(
    definition_files,
     c_custom_output_name,
     java_output_dir):
    packagename = (
        'fi',
        'helsinki',
     'cs',
     'tituomin',
     'nativebenchmark',
     'benchmark')

    all_benchmarks = read_benchmarks(definition_files)

    out_c = open(c_custom_output_name, 'w')
    out_c.write(all_benchmarks['C']['module'])

    java_classes = {}  # classname, contents

    for lang, data in all_benchmarks.iteritems():
        for benchmark in data['benchmarks']:

            direction = benchmark['direction']
            from_lang, to_lang = direction[0].upper(), direction[1].upper()
            if from_lang != lang:
                logging.error("Invalid language spec.")
                exit(1)

            classname = '{0}2{1}'.format(from_lang, to_lang) + benchmark['id']
            if 'vary' in benchmark:
                dyn_par = 'true'
            else:
                dyn_par = 'false'
            if 'alloc' in benchmark:
                # large heap 128/2 = 64 Mb, 128 el 8 byte array...
                is_allocating = 'true'
            else:
                is_allocating = 'false'

            representative = benchmark.get('representative', True)

            if representative:
                representative = "true"
            else:
                representative = "false"

            if from_lang == 'C':
                out_c.write(put(
                        c_nativemethod.t_run_method,
                        return_type='void',
                        parameters='jobject instance',
                        function='runInternal',
                        packagename='_'.join(packagename),
                    classname=classname,
                    body=put(
                        loop_code.t_c_jni_call,
                        debug=classname,
                        benchmark_body=benchmark['code']))

                java_classes[classname] = {
                    'filename': classname + '.java',
                    'code': (put(
                        java_benchmark.t,
                        representative=representative,
                        _id=benchmark['id'],
                        packagename='.'.join(packagename),
                        classname=classname,
                        description=benchmark.get('description', ''),
                        is_allocating=is_allocating,  # todo: measure
                        from_language=from_lang,
                        to_language=to_lang,
                        seq_no='-1',
                        has_dynamic_parameters=dyn_par,
                        is_nonvirtual='false',
                        run_method='public native void runInternal();',
                        ))}

            elif from_lang == 'J' and to_lang == 'J':
                java_classes[classname] = {
                    'filename': classname + '.java',
                    'code': (
                        put(
                            java_benchmark.t,
                            representative=representative,
                            _id=benchmark['id'],
                            packagename='.'.join(packagename),
                            imports="\n".join(
                                ['import android.content.pm.PermissionInfo;',
                                 'import java.nio.ByteBuffer;'
                                 'import java.lang.ref.WeakReference;'
                                 ]),
                            classname=classname,
                            class_fields=benchmark.get('class_init', ''),
                            description=benchmark.get('description' ''),
                            is_allocating=is_allocating,
                            from_language=from_lang,
                            to_language=to_lang,
                            is_nonvirtual='false',
                            seq_no='-1',
                            has_dynamic_parameters=dyn_par,
                            run_method=put(
                                java_benchmark.java_run_method_inline_t,
                                init=data['inits'],
                                type_init=benchmark.get('method_init', ''),
                                loop=put(
                                    loop_code.t_java,
                                          finished=benchmark.get('finished', ''),
                                          benchmark_body=benchmark['code'])))}

    out_c.flush()
    out_c.close()

    for classname, contents in java_classes.iteritems():
        f = open(
            path.join(java_output_dir,
     '/'.join(packagename),
                contents['filename']),
            'w')
        f.write(contents['code'])
        f.flush()
        f.close()

    return (['.'.join(packagename + (classname,))
             for classname in java_classes.keys()])
```

templates/arrays.py

```
from templating import partial

t_loop = """
<% declare_idx %>
<% declare_variables %>
for (idx = 0; idx < current_size; idx++) {
    <% body %>
}
"""

t_read = partial(
    t_loop,
    body="""
    <% variable_in %> = <% array_variable %>[idx];
    """)

t_write = partial(
    t_loop,
    body="""
    <% array_variable %>[idx] = <% element_literal %>; """)

t_init_nio = """
    <% type_declarations %>
    int localPersistentValue = 0;
    current_size /= 64;

"""

```

```
t_read_nio = partial(
    t_loop,
    body="""
    <% variable_in %> = <% array_variable %>.get<% type_name %>(idx);
    """)

t_write_nio = partial(
    t_loop,
    body="""
    <% array_variable %>.put<% type_name %>(idx, <% element_literal %>);
    """)

t_read_nio_as_view = partial(
    t_loop,
    body="""
    <% variable_in %> = <% array_variable %>.get(idx);
    """)

t_write_nio_as_view = partial(
    t_loop,
    body="""
    <% array_variable %>.put(idx, <% element_literal %>);
    """)

t_bulk_read = """
<% declare_variables %>
<% array_variable %>.clear();
<% array_variable %>.get(<% array_in %>, 0, current_size);
"""

t_bulk_write = """
<% declare_variables %>
<% array_variable %>.clear();
<% array_variable %>.put(<% array_in %>, 0, current_size);
"""
```

templates/c_jni_function.py

```
t = """

JNIEXPORT <% return_type %> JNICALL
Java_<% package %>_<% class_name %>_<% function %>
(JNIEnv *env, <% parameters %>) <%
    <% set_returnvalues %>
 %>

"""
```

templates/c_module.py

```
t = """
#include <jni.h>
#include <android/log.h>
#include <stdio.h>
#include "natives.h"
#include "native_benchmarks.h"
#include "returnvalues.h"

<% initialisers %>
<% jni_function_templates %>

"""

initialisers_t = """
static jmethodID mids[<% amount_of_methods %>];

static void init_methodids(JNIEnv *env) {
    jmethodID mid;
<% mid_inits %>
}


int check_interrupted(JNIEnv *env) {
    jobject current_thread = NULL;
    current_thread = (
        (*env)->CallStaticObjectMethod(env, thread_class, current_thread_mid));
    if (current_thread == NULL) {
        __android_log_write(ANDROID_LOG_ERROR, "check_interrupted",
            "Can't get current thread");
        return 1;
    }
    jboolean interrupted = (*env)->CallBooleanMethod(
        env, current_thread, is_interrupted_mid);
    (*env)->DeleteLocalRef(env, current_thread);
    if (interrupted == JNI_TRUE) {
        return 1;
    }
    return 0;
}

void throw_interrupted_exception(JNIEnv *env) {
    jclass newExcCls;
    newExcCls = (*env)->FindClass(env,
    "java/lang/InterruptedException");
    if (newExcCls == NULL) {
        /* Unable to find the exception class, give up. */
        return;
    }
    (*env)->ThrowNew(env, newExcCls, "thrown from C code");
}
```

```

JNIEXPORT void JNICALL
Java_fi_helsinki_cs_tituomin_nativebenchmark_BenchmarkRegistry_initNative
(JNIEnv *env, jclass cls, jlong reps, jlong interval, jclass javaCounterparts,
jobject javaCounterpartsInstance, jclass thread_cls) {
    repetitions = reps;
    interrupted = 0;

    CHECK_INTERRUPTED_INTERVAL = interval;

    jclass java_counterparts_class_global_ref = NULL;
    jclass thread_class_global_ref = NULL;
    jobject java_counterparts_object_global_ref = NULL;

    java_counterparts_class_global_ref = (*env)->NewGlobalRef(
        env, javaCounterparts);
    if (java_counterparts_class_global_ref == NULL) {
        __android_log_write(ANDROID_LOG_ERROR, "initNative",
            "Could not create global ref.");
        return;
    }
    java_counterparts_class = java_counterparts_class_global_ref;

    java_counterparts_object_global_ref = (*env)->NewGlobalRef(env,
        javaCounterpartsInstance);
    if (java_counterparts_object_global_ref == NULL) {
        __android_log_write(ANDROID_LOG_ERROR, "initNative",
            "Could not create global ref.");
        return;
    }
    java_counterparts_object = java_counterparts_object_global_ref;

    if (!(*env)->IsInstanceOf(env, java_counterparts_object,
        java_counterparts_class)) {
        __android_log_write(ANDROID_LOG_ERROR, "initNative",
            "JavaCounterparts instance or class is not correct.");
        return;
    }

    init_methodids(env);

    thread_class_global_ref = (*env)->NewGlobalRef(env, thread_cls);
    if (thread_class_global_ref == NULL) {
        __android_log_write(ANDROID_LOG_ERROR, "initNative",
            "Could not create global ref.");
        return;
    }
    thread_class = thread_class_global_ref;

    current_thread_mid = (*env)->GetStaticMethodID(env, thread_class,
        "currentThread", "()Ljava/lang/Thread;");
    if (current_thread_mid == NULL) {
        __android_log_write(ANDROID_LOG_ERROR, "initNative",
            "Could not find currentThread");
        return;
    }
    is_interrupted_mid = (*env)->GetMethodID(env, thread_class,
        "isInterrupted", "()Z");
    if (is_interrupted_mid == NULL) {
        __android_log_write(ANDROID_LOG_ERROR, "check_interrupted",
            "Can't get isInterrupted method");
        return;
    }

}

JNIEXPORT void JNICALL
Java_fi_helsinki_cs_tituomin_nativebenchmark_BenchmarkRegistry_setRepetitions
(JNIEnv *env, jclass cls, jlong reps) {
    repetitions = reps;
}

JNIEXPORT void JNICALL
Java_fi_helsinki_cs_tituomin_nativebenchmark_BenchmarkRegistry_interruptNative
(JNIEnv *env, jclass cls) {
    interrupted = 1;
}

JNIEXPORT void JNICALL
Java_fi_helsinki_cs_tituomin_nativebenchmark_BenchmarkRegistry_resetInterruptFlag
(JNIEnv *env, jclass cls) {
    interrupted = 0;
}

"""

mid_init_t = """
    mid = (*env)->Get<% static %>MethodID(
        env, java_counterparts_class, "<% method_name %>",
        "<% method_descriptor %>");
    if (mid == NULL) {
        __android_log_write(ANDROID_LOG_VERBOSE, "nativemethod",
            "<% method_descriptor %> not found.");
        return; /* method not found */
    }
    mids[<% seq_no %> - 1] = mid;

"""
```

templates/c_nativemethod.py

```
from templating import partial, put
import loop_code

# todo: templates confusingly named

t_run_method = """
JNIEXPORT <% return_type %> JNICALL
Java_<% packagename %>_<% classname %>_<% function %>
(JNIEnv *env, <% parameters %>) {
    <% parameter_declarations %>;
    <% parameter_initialisations %>;
    <% prebody %>
    <% body %>
}

"""

t = partial(
    t_run_method,
    body='return <% return_expression %>;',
    remove=['parameter_declarations',
            'parameter_initialisations',
            'prebody'])

# C to C
t_caller_native = partial(
    t_run_method,
    return_type='void',
    function='runInternal',
    parameters='jobject instance',
    prebody='',
    body=partial(
        loop_code.t_c_jni_call,
        benchmark_body=(
            '<% counterpart_method_name %>' +
            '(<% counterpart_method_arguments %>);')))

# C to J
t_caller_java = partial(
    t_run_method,
    return_type='void',
    function='runInternal',
    parameters='jobject instance',
    prebody='jmethodID mid = mids[<% seq_no %> - 1];')


def call_java_from_c(static=True, nonvirtual=False, **parameters):
    benchmark_body = ''
    if static:
        benchmark_body = (
            '(*env)->CallStatic<% java_method_type %>Method<% call_variant %>'
            '(env, java_counterparts_class, mid<% arguments %>);')
    elif nonvirtual:
        benchmark_body = (
            '(*env)->CallNonvirtual'
            '<% java_method_type %>Method<% call_variant %>'
            '(env, java_counterparts_object, java_counterparts_class,'
            ' mid<% arguments %>);')
    else:
        benchmark_body = (
            '(*env)->Call<% java_method_type %>Method<% call_variant %>' +
            '(env, java_counterparts_object, mid<% arguments %>);')

    parameters['body'] = put(
        loop_code.t_c_jni_call,
        benchmark_body=put(benchmark_body, **parameters))

    return partial(t_caller_java, **parameters)
```

templates/__init__.py

```

```

templates/java_benchmark.py

```
from templating import partial
import loop_code
import logging

t = """
package <% packagename %>;

import fi.helsinki.cs.tituomin.nativebenchmark.Benchmark;
import fi.helsinki.cs.tituomin.nativebenchmark.BenchmarkRegistry;
import fi.helsinki.cs.tituomin.nativebenchmark.BenchmarkParameter;
import fi.helsinki.cs.tituomin.nativebenchmark.measuringtool.BasicOption;
import fi.helsinki.cs.tituomin.nativebenchmark.MockObject;
<% imports %>
import android.util.Log;

public class <% classname %> <% class_relations %> extends Benchmark {

    public <% classname %> (BenchmarkParameter bp) {
        init(bp);
    }

    public String from() {
        return "<% from_language %>";
    }

    public String to() {
        return "<% to_language %>";
    }

    public int sequenceNo() {
        return <% seq_no %>;
    }

    public String id() {
        return "<% _id %>";
    }

    public boolean representative() {
        return <% representative %>;
    }

    public boolean dynamicParameters() {
        return <% has_dynamic_parameters %>;
    }

    public String description() {
        return "<% description %>";
    }

    public boolean isAllocating() {
        return <% is_allocating %>;
    }

    public boolean isNonvirtual() {
        return <% is_nonvirtual %>;
    }

    <% class_fields %>

    <% native_method %>

    <% run_method %>

}

"""

native_method_t = ('<% modifiers %> native <% return_type %> '
                   '<% name %> (<% parameters %>);')
dynamic_parameter_t = (
    'new BasicOption(BasicOption.VARIABLE, "<% variable %>")'.strip()
native_run_method_t = 'public native void runInternal();'

loop = partial(loop_code.t_java,
               benchmark_body=('<% counterpart_method_name %> '
                                  '(<% counterpart_method_arguments %>);'))

java_run_method_t   = partial("""

    public void runInternal() {
        JavaCounterparts counterpartInstance = JavaCounterparts.INSTANCE;
        <% parameter_declarations %>;
        <% parameter_initialisations %>;

        <% loop %>
    }

""", loop=loop)

java_run_method_inline_t = """

    public void runInternal() {
        <% init %>
        <% type_init %>
        <% loop %>
    }

"""
```

templates/java_counterparts.py

```
from templating import put

t = """
package <% packagename %>;

<% imports %>
import fi.helsinki.cs.tituomin.nativebenchmark.BenchmarkParameter;
import fi.helsinki.cs.tituomin.nativebenchmark.BenchmarkRunner;
import android.util.Log;


public enum JavaCounterparts {
    INSTANCE;

    <% return_value_declarations %>
    public int persistentValue;
    public static int staticpersistentValue = 0;

    private JavaCounterparts() {
        persistentValue = 0;
    }

    public static void initParams(BenchmarkParameter benchmarkParameter) {
        <% return_value_inits %>
    }

    <% counterpart_methods %>

}

"""

# todo     <% return_values %>


counterpart_t = """

<% privacy %> <% static %> <% return_type %> <% methodname %>(<% parameters %>) {
    <% static %>persistentValue = (<% static %>persistentValue + 1) % 10;
    return <% return_expression %>;
}

"""

return_value_t = (
    "private static <% actualtype %> = "
    "benchmarkParameter.retrieve<% typename %>(<% typespecs %>);")
```

templates/java_registry_init.py

```
from templating import put

t = """
package fi.helsinki.cs.tituomin.nativebenchmark;

import fi.helsinki.cs.tituomin.nativebenchmark.Benchmark;
import fi.helsinki.cs.tituomin.nativebenchmark.BenchmarkRegistry;
import fi.helsinki.cs.tituomin.nativebenchmark.BenchmarkParameter;
import fi.helsinki.cs.tituomin.nativebenchmark.benchmark.*;
import java.util.List;

public class BenchmarkInitialiser {

    public static void init(BenchmarkParameter bp) {
        List<Benchmark> benchmarks = BenchmarkRegistry.getBenchmarks();

        <% register_benchmarks %>
    }

}

"""
```

```


def inits(classname):
    return 'benchmarks.add(new {0} (bp));'.format(classname)
```

templates/loop_code.py

```
from templating import put, partial

t = """
    <% declare_counters %>
    <% additional_declaration %>

    <% init_counters %>
    division  = repetitions / interval + 1;
    remainder = repetitions % interval + 1;

    <% debug %>
    <% additional_init %>
    while (--division != 0) {
        <% init_counters %>
        interval = interval + 1;
        while (--interval != 0) {
            <% pre_body %>
            <% extra_debug %>
            <% benchmark_body %>
            <% post_body %>
        }
        if (<% test_interrupted %>) {
            <% debug_interrupted %>
            return;
        }
    }

    <% additional_init %>

    while (--remainder != 0) {
        <% pre_body %>
        <% benchmark_body %>
        <% post_body %>
    }

    <% removal_prevention %>
    <% finished %>

"""

jni_push_frame = """
if (refs == 0) {
    refs = LOCAL_FRAME_SIZE;
    if ((*env)->PushLocalFrame(env, LOCAL_FRAME_SIZE) < 0) {
        return;
    }
}
"""

jni_pop_frame = """
if (--refs == 0) {
    (*env)->PopLocalFrame(env, NULL);
}

"""


t_c_base = partial(
    t,
    declare_counters='jlong interval, division, remainder;',
    init_counters='interval = CHECK_INTERRUPTED_INTERVAL;',
    test_interrupted='interrupted')

t_c_jni_call = partial(
    t_c_base,
    additional_declaration='jlong refs;',
    additional_init='refs = 0;',
    remove=[
        'extra_debug',
        'debug',
     'debug_interrupted',
     'removal_prevention'],
    pre_body=jni_push_frame,
    post_body=jni_pop_frame)

t_c = partial(
    t_c_base,
    remove=['extra_debug', 'debug', 'debug_interrupted',
            'additional_declaration', 'additional_init',
            'pre_body', 'post_body', 'removal_prevention'])

t_java = partial(
    t,
    test_interrupted='Thread.currentThread().isInterrupted()',
    extra_debug='',  # ,'Log.v("Benchmark", division + " " + interval);',
    declare_counters='long interval, division, remainder;',
    init_counters='interval = BenchmarkRegistry.CHECK_INTERRUPTED_INTERVAL;',
    removal_prevention='repetitionsLeft = division * interval + remainder;',
    remove=['additional_declaration',
            'additional_init',
            'pre_body',
            'post_body'])
```

templating.py

```python
import string
import logging
formatter = string.Formatter()


class PartialDict(dict):

    def __missing__(self, key):
        return "<% " + key + " %>"


class PurgeDict(dict):

    def __missing__(self, key):
        return ""


def escape(string):
    string = string.replace('{', '__BEG__')
    string = string.replace('}', '__END__')
    string = string.replace('<% ', '{')
    string = string.replace(' %>', '}')
    return string


def unescape(string):
    string = string.replace('{', '<% ')
    string = string.replace('}', ' %>')
    string = string.replace('__BEG__', '{')
    string = string.replace('__END__', '}')
    return string


def put(template, remove=None, purge=True, **kwargs):
    try:
        template = escape(template)
        for k, v in kwargs.iteritems():
            if isinstance(v, str):
                kwargs[k] = escape(v)
            if v is None:
                kwargs[k] = ''

        if remove:
            for k in remove:
                kwargs[k] = ''

        if purge:
            fdict = PurgeDict(**kwargs)
        else:
            fdict = PartialDict(**kwargs)

        result = formatter.vformat(template, (), fdict)
        result = unescape(result)
        return result
    except ValueError as e:
        logging.error('error with template: ' + template)
        raise e
    return None


def partial(template, remove=None, **kwargs):
    return put(template, remove=remove, purge=False, **kwargs)
```

Python-komponentit (analyysi)

Hakemistossa .

analysis.py

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

from numpy import polyfit, reshape, polyval

def linear_fit_columns(x, y):
    p, residuals, rank, singular_values, rcond = polyfit(x, y, 1, full=True)
    ynorm = normalized(x, y, p)
    pnorm, residuals, rank, singular_values, rcond = polyfit(
        x, ynorm, 1, full=True)
    return p, residuals

def normalized(x, y, poly):
    # normali
    return (x - poly[1]) / poly[0]

def linear_fit(rows):
    columns = reshape(rows, len(rows)*len(rows[0]), order='F').reshape(
        (len(rows[0]), -1))
    x = columns[0]
    columns = columns[1:]
    residuals = [linear_fit_columns(x, col)[1][0] for col in columns]
    polys = [linear_fit_columns(x, col)[0] for col in columns]
    return x, polys, residuals

def estimate_measuring_overhead(rows):
    x, polys, residuals = linear_fit(rows)
    return [p[1] for p in polys]

def optimize_bins(x):
    """
    Created on Thu Oct 25 11:32:47 2012

    Histogram Binwidth Optimization Method

    Shimazaki and Shinomoto, Neural Comput 19 1503-1527, 2007
    2006 Author Hideaki Shimazaki, Matlab
    Department of Physics, Kyoto University
    shimazaki at ton.scphys.kyoto-u.ac.jp
    Please feel free to use/modify this program.

    Version in python adapted Érbet Almeida Costa

    Data: the duration for eruptions of
    the Old Faithful geyser in Yellowstone National Park (in minutes)
    or normal distribuition.
    Version in python adapted Érbet Almeida Costa
    Bugfix by Takuma Torii 2.24.2013

    """

    import numpy as np
    from numpy import mean, size, zeros, where, transpose
    from numpy.random import normal
    from matplotlib.pyplot import hist
    from scipy import linspace
    import array

    x_max = max(x)
    x_min = min(x)
    N_MIN = 4   #Minimum number of bins (integer)
                #N_MIN must be more than 1 (N_MIN > 1).
    N_MAX = 1000  #Maximum number of bins (integer)
    N = range(N_MIN,N_MAX) # #of Bins
    N = np.array(N)
    D = (x_max-x_min)/N    #Bin size vector
    C = zeros(shape=(size(D),1))

    #Computation of the cost function
    for i in xrange(size(N)):
        edges = linspace(x_min,x_max,N[i]+1) # Bin edges
        ki = hist(x,edges) # Count # of events in bins
        ki = ki[0]
        k = mean(ki) #Mean of event count
        v = sum((ki-k)**2)/N[i] #Variance of event count
        C[i] = (2*k-v)/((D[i])**2) #The cost Function
    #Optimal Bin Size Selection

    cmin = min(C)
    idx  = where(C==cmin)
    idx = int(idx[0])
    optD = D[idx]

    edges = linspace(x_min,x_max,N[idx]+1)

    return optD, edges
```

datafiles.py

```
#!/usr/bin/python

import re
from collections import OrderedDict as odict
```

```
import sys

SEPARATOR = ','
RE_EMPTY = re.compile('^\s*$')
RE_NUMERICAL = re.compile('^-?[0-9]+$')


def explode(line):
    return line.split(SEPARATOR)

def value(string, key=None):
    if key in ['start', 'end']:
        return string
    if key == 'class':
        return string.split('.')[-1]
    if string == '-' or RE_EMPTY.match(string):
        return None
    if RE_NUMERICAL.match(string):
        return int(string)
    else:
        return string

def empty_label():
    empty_label.cnt += 1
    return 'empty_{0}'.format(empty_label.cnt)

empty_label.cnt = 0

def read_datafiles(files, silent=False):
    if not silent:
        print 'Reading from %s files' % len(files)
    benchmarks = []
    #-1: there is an empty field at the end...

    keys_with_values = set()
    all_keys = set()

    lineno = 1
    for i, f in enumerate(files):
        line = f.readline()
        labels = explode(line)
        for i, l in enumerate(labels):
            # account for the fact that there might be an empty label
            # and corresponding column (usually the last)
            if RE_EMPTY.match(l):
                labels[i] = empty_label()

        all_keys.update(labels)

        line = f.readline()
        while line != '':
            exploded_line = explode(line)
            pad_amount = len(labels) - len(exploded_line)
            exploded_line.extend(['-'] * pad_amount)
            if len(labels) != len(exploded_line):
                 print ('missing values', f.name, 'line', lineno, 'labels',
                       len(labels), 'values', len(exploded_line))
                 exit(1)

            benchmark = dict()
            benchmark['lineno'] = lineno

            for key, string in zip(labels, exploded_line):
                benchmark[key] = value(string, key=key)

                if value(string, key=key) != None:
                    keys_with_values.add(key)

            #if benchmark['response_time'] != None:
            benchmarks.append(benchmark)

            line = f.readline()
            lineno += 1

    keys_without_values = all_keys - keys_with_values

    benchmark_keycount = None
    for benchmark in benchmarks:
        for key in keys_without_values:
            if key in benchmark:
                del benchmark[key]
        current_keycount = len(benchmark.keys())
        benchmark_keycount = benchmark_keycount or current_keycount
        if benchmark_keycount != current_keycount:
            print ("Benchmarks have different amount of data",
                   benchmark_keycount, current_keycount, "at line",
                   benchmark['lineno']
            exit(1)

    if not silent:
        print 'Read %d lines' % (lineno - 1)
    return benchmarks

def read_measurement_metadata(mfile, combine_compatibles):
    compatibles = odict()
    measurement = None
    line = None

    i = 0
    while line != '':
        skipped = False
        while line == "\n":
            line = mfile.readline()
            skipped = True

        if skipped:
            if measurement:
                if 'tools' in measurement:
                    measurement['tool'] = measurement['tools']
                revision = measurement.get('code-revision')
                checksum = measurement.get('code-checksum')
                repetitions = measurement.get('repetitions')
                tool = measurement.get('tool')
                cpufreq = measurement.get('cpu-freq')
                benchmark_set = measurement.get('benchmark-set')
                substring_filter = measurement.get('substring-filter')
                if measurement.get('rounds') == None:
                    measurement['rounds'] = 1

                if revision and repetitions:
                    if combine_compatibles:
                        key = (revision, checksum, repetitions, tool, cpufreq,
                               benchmark_set, substring_filter)
                    else:
                        key = i
                        i += 1
                    if key not in compatibles:
                        compatibles[key] = []
                    compatibles[key].append(measurement)
            measurement = {}

        if line != None:
            splitted = line.split()
            if len(splitted) > 1:
                key = splitted[0].rstrip(':')
                val = ' '.join(splitted[1:])
                measurement[key] = val.strip()

        line = mfile.readline()

    return compatibles
```

gnuplot.py

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import os
import uuid

INIT_PALETTE = """
# line styles for ColorBrewer Dark2
# for use with qualitative/categorical data
# provides 8 dark colors based on Set2
# compatible with gnuplot >=4.2
# author: Anna Schneider

# line styles
set style line 1 pt 7 lt 1 lc rgb '#1B9E77' # dark teal
set style line 2 pt 7 lt 1 lc rgb '#D95F02' # dark orange
set style line 3 pt 7 lt 1 lc rgb '#7570B3' # dark lilac
set style line 4 pt 7 lt 1 lc rgb '#000000' # black
set style line 5 pt 7 lt 1 lc rgb '#E7298A' # dark magenta
set style line 6 pt 7 lt 1 lc rgb '#66A61E' # dark lime green
set style line 7 pt 7 lt 1 lc rgb '#E6AB02' # dark banana
set style line 8 pt 7 lt 1 lc rgb '#A6761D' # dark tan
set style line 9 pt 7 lt 1 lc rgb '#666666' # dark gray
set style line 10 pt 7 lt 1 lc rgb '#1b70b3' # dark blue

set style line 11 pt 5 lt 2 lc rgb '#1B9E77' # dark teal
set style line 12 pt 5 lt 2 lc rgb '#D95F02' # dark orange
set style line 13 pt 5 lt 2 lc rgb '#7570B3' # dark lilac
set style line 14 pt 5 lt 2 lc rgb '#000000' # black
set style line 15 pt 5 lt 2 lc rgb '#E7298A' # dark magenta
set style line 16 pt 5 lt 2 lc rgb '#66A61E' # dark lime green
set style line 17 pt 5 lt 2 lc rgb '#E6AB02' # dark banana
set style line 18 pt 5 lt 2 lc rgb '#A6761D' # dark tan
set style line 19 pt 5 lt 2 lc rgb '#666666' # dark gray
set style line 20 pt 5 lt 2 lc rgb '#1b70b3' # dark blue


# palette
set palette maxcolors 8
set palette defined ( 0 '#1B9E77',\
                      1 '#D95F02',\
              2 '#7570B3',\
              3 '#E7298A',\
              4 '#66A61E',\
              5 '#E6AB02',\
              6 '#A6761D',\
              7 '#666666' )
"""

INIT_PLOTS_PDF = """
set terminal pdfcairo size 32cm,18cm {sizesuffix}
set size 1, 0.95
set output '{filename}'
"""

INIT_PLOTS_LATEX = """
set terminal epslatex input color \
header "\\\\caption{{{caption}}}\\\\label{{fig:{label}}}" {sizesuffix}
set pointsize 1.0
set format y "%4.2s%cs"
set output
"""

INIT_PLOTS_SVG = """
```

```
set terminal svg {sizesuffix}
set pointsize 1.0
set format y "%4.2s%cs"
set output
"""

INIT_PLOTS_COMMON = """
set grid
set xlabel "kutsuparametrien määrä"
"""

INIT_PLOT_LABEL_PDF = """
set label 1 "{bid}" at graph 0.01, graph 1.06
"""

TEMPLATES = {}
INIT_KEY = {}

TEMPLATES['binned_init'] = """
set title '{title}
binwidth={binwidth}
set boxwidth binwidth
set style fill solid 1.0
set xrange [{min_x}:{max_x}]
set yrange [0:{max_y}]
"""
# border lt -1
#bin(x,width)=width*floor(x/width) + width/2.0

TEMPLATES['binned_frame'] = """
#set label 2 "{datapoints}" at graph 0.8, graph 1.06
set bmargin 20
set tmargin 20
set rmargin 20
set lmargin 20
plot '-' using 1:2 notitle with boxes lt rgb "{color}"\n{values}\ne\n
#unset xlabel
#unset ylabel
#unset label 1
#unset title
unset xtics
unset ytics
"""

SET_TITLE_AND_PAGE_LABEL = """
set title '{title}'
set label 2 "{page}" at screen 0.9, screen 0.95
"""

INIT_KEY['simple_groups'] = """
set key {key_placement} box notitle width -3 height +1 vertical
"""

```

```
TEMPLATES['simple_groups'] = """
set ylabel "vasteaika {reps} toistolla"
set xlabel "{xlabel}"
plot for [I=2:{last_column}] '{filename}' index {index} \
using 1:I title columnhead with points ls I-1
"""

TEMPLATES['fitted_lines'] = """
set ylabel "vasteaika {reps} toistolla"
set xlabel "{xlabel}"
plot for [I=2:{last_real_column}] '{filename}' index {index} using 1:I \
title columnhead with points ls I-1, \
for [I={first_fitted_column}:{last_column}] '{filename}' index {index} \
using 1:I notitle with lines ls I-{first_fitted_column}+1
"""

TEMPLATES['named_columns'] = """
set yrange [0:*]
set xlabel "{xlabel}"
plot for [I=2:{last_column}] '{filename}' index {index} using I:xtic(1) \
title columnhead with linespoints
"""

TEMPLATES['histogram'] = """
#set xlabel "{xlabel}"
unset xlabel
#set xlabel "{xlabel}" rotate
unset ylabel
set y2label "vasteaika {reps} toistolla"
set size 1, 1
unset x2tics
#unset xtics
unset ytics

set y2tics format "%.00s%cs" rotate

set xtics out rotate
set key at graph 0.1, 0.9 width 2 height 8 notitle horizontal nobox samplen 0.2
set label 1 'C$\\rightarrow$Java' at graph 0.145, 0.78 left rotate by 90
set label 2 'Java$\\rightarrow$Java' at graph 0.205, 0.78 left rotate by 90
# set label 2 'Nowhere' at graph 0.09, 0.85 left rotate by 90
# set label 3 'Everywhere' at graph 0.2, 0.85 left rotate by 90
#set boxwidth 0.9 relative
# set style fill solid border lc rgbcolor "black"
set style data histograms
set style histogram clustered
#set style fill solid 1.0 border lt -1

plot [] [0:*] for [I=2:{last_column}] '{filename}' index {index} \
using I:xtic(1) every ::1 title " " with histogram fillstyle solid 1.0 border lt -1
"""

measurement_id = None
```

```
plot_directory = '/home/tituomin/gradu/paper/figures/plots'

def init(plotscript, filename, mid, output_type='pdf'):
    global measurement_id, plot_directory
    measurement_id = mid
    if output_type == 'pdf':
        plotscript.write(INIT_PLOTS_PDF.format(filename=filename))
        plotscript.write(INIT_PLOT_LABEL_PDF.format(bid=measurement_id))
    plotscript.write(INIT_PLOTS_COMMON)
    plotscript.write(INIT_PALETTE)

GROUPTITLES={
    'direction': 'kutsusuunta',
    'from': 'kieli'
}

def output_plot(data_headers, data_rows, plotpath,
                plotscript, title, specs, style, page,
                identifier,
                xlabel, additional_data=None, output='pdf',
                key_placement="inside top left", reps='XXX-fixme-XXX'):
    global plot_directory
    template = TEMPLATES[style]

    rowlen = len(data_rows[0]) - 1
    size = 'normal'
    if style == 'fitted_lines':
        rowlen /= 2
    if (page > 51 and rowlen > 7) or rowlen > 10:
        size = 'tall'
    if rowlen < 15:
        size = 'normal'
    if identifier in [
        'basic-call-all-types-j-j-fit',
        'basic-call-all-types-c-c-fit',
        'variable-argument-size-j-c',
        'special-calls-arrayelements-c-j-fit',
        'special-calls-arrayregion-c-j-fit']:
        size = 'tall'

    if output in ['latex', 'svg']:
        if output == 'latex':
            init_tmpl = INIT_PLOTS_LATEX
            file_suffix = 'tex'
        elif output == 'svg':
            init_tmpl = INIT_PLOTS_SVG
            file_suffix = 'svg'
        sizesuffix=''
        if size == 'tall':
            if output == 'svg':
                sizesuffix = 'size 1000,800'
            else:
                sizesuffix="size 15cm,13cm"
        else:
            if output == 'svg':
                sizesuffix="size 1000,600"
            else:
                sizesuffix="size 15cm,10cm"
        plotscript.write(
            init_tmpl.format(
                caption=title,
                label=identifier,
                sizesuffix=sizesuffix))
        if specs['variable'] == 'dynamic_size':
            plotscript.write("set xrange [0:512]\n")
            plotscript.write("set xtics 0, 64\n")
            plotscript.write("set format x \"%6.sB\"\n")
        else:
            plotscript.write("unset xtics\n")
            plotscript.write("set xtics autofreq\n")
            plotscript.write("set xrange [*:*]\n")
            plotscript.write("set format x \"%6.s\"\n")

        if size == 'tall':
            if identifier in ['special-calls-arrayelements-c-j-fit',
                              'special-calls-arrayregion-c-j-fit']:
                plotscript.write(
                    "set tmargin at screen 0.8\nset key above box "
                    "horizontal maxrows 8 maxcols 4 samplen 1 "
                    "spacing .5 font \",4\"\n");
            else:
                plotscript.write(
                    "set tmargin at screen 0.85\n"
                    "set key above nobox horizontal\n");
        else:
            plotscript.write("set tmargin at screen 0.95\n")
        plotscript.write("set output '{}'".format(
            os.path.join(plot_directory,
                         "plot-{}-{}.{}".format(
                             measurement_id, identifier, file_suffix))))

    if plotpath:
        # external data
        filename = os.path.join(plotpath, "plot-" + str(uuid.uuid4()) + ".data")
        plotdata = open(filename, 'w')
        specs['convert_to_seconds'] = False # (output == 'latex')
        if output == 'latex':
            specs['tinylabels'] = True
        if output == 'svg':
            specs['scriptlabels'] = True
        plotdata.write(print_benchmarks(data_headers, data_rows, title,
                                        **specs))

    miny = 0
    for row in data_rows:
        for cell in row[1:]:
            if cell < miny:
                miny = cell
    if miny == None:
        miny = '*'

    if output == 'pdf':
        plotscript.write(SET_TITLE_AND_PAGE_LABEL.format(page=identifier,
                                                         title=title))

    if style == 'binned':
        plotscript.write(template.format(
           title = title, page = identifier, filename = filename, index = 0,
            last_column = len(data_rows[0]),
           xlabel = xlabel, miny=miny, **additional_data))

    elif style == 'fitted_lines':
        length = len(data_headers) - 1
        last_real_column = 1 + length / 2
        first_fitted_column = last_real_column + 1
        plotscript.write(template.format(
           title = title, reps = reps, page = identifier, filename = filename,
            index = 0, last_column = len(data_rows[0]),
           xlabel = xlabel, miny=miny, last_real_column=last_real_column,
            first_fitted_column=first_fitted_column))

    elif style == 'simple_groups':
        grouptitle = GROUPTITLES.get(specs['group'], 'group')
        if key_placement is None:
            plotscript.write("\nunset key\n")
        elif size != 'tall':
            plotscript.write(INIT_KEY[style].format(
                key_placement=key_placement))

        plotscript.write(template.format(
            title = title, reps = reps, page = identifier, filename = filename,
            index = 0, last_column = len(data_rows[0]),
            xlabel = xlabel, miny=miny, grouptitle=grouptitle))

    else:
        grouptitle = GROUPTITLES.get(specs['group'], 'group')
        plotscript.write(template.format(
            title = title, page = identifier, filename = filename, index = 0,
            last_column = len(data_rows[0]),
            key_placement = key_placement, xlabel = xlabel, reps=reps,
            miny=miny, grouptitle=grouptitle))


def print_benchmarks(data_headers, data_rows, title, group=None, variable=None,
                     measure=None, convert_to_seconds=False, tinylabels=False,
                     scriptlabels=False):
    result = '#{0}\n'.format(title)
    if group and variable and measure:
        result = '#measure:{m} variable:{v} group:{g}'.format(
            m=measure, v=variable, g=group)

    prefix = ""
    suffix = ""
    if tinylabels:
        prefix = "\\\\tiny "
    elif scriptlabels:
        prefix = "\\\\tiny{"
        suffix = "}"
    result = " ".join([format_value("{}{}{}".format(prefix, k, suffix))
                            for k in data_headers])
    result += '\n'

    for row in data_rows:
        results = []
        for i, v in enumerate(row):
            convert = convert_to_seconds and i > 0
            results.append(format_value(v, convert_to_seconds=convert))
        result += ' '.join(results) + '\n'
    result += '\n\n'

    return result

def format_value(value, convert_to_seconds=False):
    if value == None:
        return "-500"
    if type(value) == str:
        return '"{0}"'.format(value)
    if type(value) == int:
        strval = str(value)
        if convert_to_seconds == False:
            return strval
        strval = strval.zfill(10)
        strlen = len(strval)
        return "{}.{}".format(
            strval[0:strlen-9],
            strval[strlen-9:])

    return str(value)

def hex_color_gradient(start, end, point):
    # start, end are tuples with r,g,b values (integer)
    # point is a point between 0 (start) and 1000 (end)
    return "#" + "".join(
        "{:0>2X}".format(
            int(start[i] +
                ((end[i] - start[i]) * (float(point))))
        for i in range(0,3))
```

plot_data.py

```python
#!/usr/bin/python
# -*- coding: utf-8 -*-

from collections import OrderedDict as odict
from itertools import groupby
from subprocess import call
from sys import argv
import functools
import pprint
import re
import os
import sys
import shutil
import uuid

import glob
import zipfile

import numpy
from numpy import array

from jni_types import primitive_type_definitions
from jni_types import object_type_definitions, array_types
from datafiles import read_datafiles, read_measurement_metadata
import analysis
from analysis import linear_fit, estimate_measuring_overhead
import gnuplot
import textualtable

FNULL = None

primitive_types = [
    t['java']
    for t in primitive_type_definitions
]

reference_types = [
    t['java']
    for t in array_types.itervalues()
]

reference_types.extend([
    t['java']
    for t in object_type_definitions
])

types = reference_types + primitive_types

plot_axes = {
    'description': 'operaatioiden määrä',
    'parameter_count': 'kutsuparametrien määrä',
    'dynamic_size': 'kohteen koko',
    'direction': 'kutsusuunta',
    'id': 'nimi'
}
pp = pprint.PrettyPrinter(depth=10, indent=4)

debugdata = open('/tmp/debug.txt', 'w')

def format_direction(fr, to, latex):
    if fr == 'J':
        fr = 'Java'
    if to == 'J':
        to = 'Java'
    if latex:
        SEPARATOR = '$\\\\rightarrow$'
    else:
        SEPARATOR = ' > '
    return "%s%s%s" % (fr, SEPARATOR, to)

DIRECTIONS = [('C', 'J'), ('J', 'C'), ('J', 'J'), ('C', 'C')]

def preprocess_benchmarks(benchmarks, global_values, latex=None):
    # For allocating benchmarks, the repetition count for individual benchmarks
    # come from the datafile. For non-allocating, it is a global value.
    keys = set([key for b in benchmarks for key in b.keys()])
    if 'repetitions' in keys:
        benchmarks = [b for b in benchmarks if b['repetitions'] is not None]
    for b in benchmarks:
        add_derived_values(b, latex=latex)
        add_global_values(b, global_values)
    return benchmarks

def add_derived_values(benchmark, latex=None):
    # migration - todo - remove
    if benchmark.get('response_time_millis') != None:
        benchmark['response_time'] = benchmark.get('response_time_millis')
        benchmark['time_unit'] = 'milliseconds'
        del benchmark['response_time_millis']
    if benchmark.get('dynamic_size') == None:
        benchmark['dynamic_variation'] = 0
        benchmark['dynamic_size'] = 0
    else:
        benchmark['dynamic_variation'] = 1
    if benchmark['no'] == -1:
        # Custom benchmark, do some name mapping:
        bid = benchmark['id']
        rename = True
        if bid == 'CopyUnicode':
            bid = 'GetStringRegion'
        elif bid == 'CopyUTF':
            bid = 'GetStringRegionUTF'
        elif bid == 'StringLength':
            bid = 'GetStringLength'
        elif bid == 'StringLengthUTF':
            bid = 'GetStringUTFLength'
        elif bid == 'ReadUnicode':
            bid = 'ReadString'
        elif bid == 'ReadUnicodeCritical':
            bid = 'ReadStringCritical'
        elif bid == 'ReadUTF':
            bid = 'ReadStringUTF'
        elif bid == 'ReadUtf':
            bid = 'ReadStringUTF'
        elif bid == 'ReadObjectArrayElement':
            bid = 'GetObjectArrayElement'
        elif bid == 'WriteObjectArrayElement':
            bid = 'SetObjectArrayElement'
        else:
            rename = False
        if rename:
            benchmark['id'] = bid

    single_type = None
    if (benchmark.get('parameter_count') == 0):
        single_type = 'any'
    elif (benchmark.get('parameter_type_count') == 1):
        for tp in types:
            if benchmark.get('parameter_type_{t}_count'.format(t=tp)) != None:
                single_type = tp
                break
    benchmark['direction'] = format_direction(
        benchmark['from'], benchmark['to'], latex)
    benchmark['single_type'] = single_type
    if 'Nio' in benchmark['id']:
        benchmark['nio'] = True
    else:
        benchmark['nio'] = False

def add_global_values(benchmark, global_values):
    for key, val in global_values.iteritems():
        if key not in benchmark or benchmark[key] == None:
            benchmark[key] = val
        elif key == 'multiplier' and benchmark[key] != None:
            benchmark[key] *= val


def extract_data(benchmarks,
                 group=None, variable=None, measure=None,
                 min_series_length=2, sort=None, min_series_width=None):

    # info == extra metadata not to be analyzed
    info = ['no', 'from', 'to', 'lineno', 'start', 'end']

    if 'class' in benchmarks[0]:
        info.append('class')
    if 'description' in benchmarks[0]:
        info.append('description')
    if re.match('parameter_type_.+count', variable):
```

```python
        info.append('parameter_count')
    if variable != 'id':
        info.append('id')

    # note: all the benchmarks have the same keyset
    all_keys = set(benchmarks[0].keys())

    # the actual keys of interest must have the least weight in sorting
    sort_last = [group, variable, measure] + info
    controlled_variables = all_keys - set(sort_last)
    sorted_keys = list(controlled_variables) + sort_last

    sorted_benchmarks = sorted(
        benchmarks,
        cmp=functools.partial(comp_function, sorted_keys))

    # 1. group benchmarks into a multi-dimensional list
    #    with the following structure:
    #    - compatible-measurements (controlled variables are equal)
    #      - plots (list of individual data series ie. plots)
    #        - multiple measurements ()
    benchmarks = group_by_keys(sorted_benchmarks, controlled_variables)
    for i, x in enumerate(benchmarks):
        benchmarks[i] = group_by_keys(x, [group])
        for j, y in enumerate(benchmarks[i]):
            benchmarks[i][j] = group_by_keys(y, [variable])

    # 2. statistically combine multiple measurements
    # for the exact same benchmark and parameters,
    # and store information about the roles of keys

    for i, compatibles in enumerate(benchmarks):
        for j, plotgroups in enumerate(compatibles):
            for k, measured_values in enumerate(plotgroups):

                plotgroups[k] = aggregate_measurements(
                    measured_values, measure, stat_fun=min)

            compatibles[j] = odict(
                (benchmark[variable], {
                    'fixed': dict(
                        (key, benchmark[key]) for key in controlled_variables),
                    'info': dict((key, benchmark[key]) for key in info),
                    'variable': variable,
                    'measure': measure,
                    'group': group,
                    variable: benchmark[variable],
                    measure: benchmark[measure],
                    group: benchmark[group]
                }) for benchmark in plotgroups)

        benchmarks[i] = odict(
            sorted(((bms.values()[0][group], bms)
                         for bms in benchmarks[i]),
                       key=lambda x: x[0]))

    return [x for x in benchmarks
            if len((x.values())[0]) >= min_series_length]


def group_by_keys(sorted_benchmarks, keyset):
    # todo make into generator?
    return [
        list(y) for x, y in groupby(
            sorted_benchmarks,
            key=lambda b: [b[k] for k in keyset])]


def aggregate_measurements(benchmarks, measure, stat_fun=min):
    values = []
    benchmark = None
    for benchmark in benchmarks:
        values.append(benchmark[measure])

    benchmark[measure] = stat_fun(values)

    if len(values) != benchmark['multiplier']:
        print ("Error: expecting", benchmark['multiplier'],
               "measurements, got", len(values))
        debugdata.write(pp.pformat(list(benchmarks)))
        exit(1)

    return benchmark


def comp_function(keys, left, right):
    for key in keys:
        if key not in left and key not in right:
            continue
        l, r = left[key], right[key]
        if l < r:
            return -1
        if l > r:
            return 1
    return 0


def without(keys, d):
    if keys == None:
        return d
    return dict(((key, val) for key, val in d.iteritems() if key not in keys))


def plot(
        benchmarks, gnuplot_script, plotpath, metadata_file,
        keys_to_remove=None, select_predicate=None,
        group=None, variable=None, measure=None,
        title=None, style=None, min_series_width=1,
        key_placement='inside top left',
        identifier=None,
        revision=None, checksum=None, output='pdf'):

    if len(benchmarks) > 0 and benchmarks[0].get('is_allocating'):
        identifier += '-alloc'
    if len(benchmarks) > 0:
        reps = benchmarks[0].get('repetitions')

    filtered_benchmarks = [
        without(keys_to_remove, x)
        for x in benchmarks
        if select_predicate(x)]

    variables = set([benchmark[variable] for benchmark in filtered_benchmarks])

    if len(variables) < 2:
        print 'Skipping plot without enough data variables', title
        return

    if len(filtered_benchmarks) == 0:
        print 'Error, no benchmarks for', title
        exit(1)

    print 'Plotting', title

    specs = {
        'group': group,
        'variable': variable,
        'measure': measure}

    data = extract_data(filtered_benchmarks, **specs)

    index = -1

    data_len = len([s for s in data if len(s.keys()) >= min_series_width])
    for series in data:
        if len(series.keys()) < min_series_width:
            # there are not enough groups to display
            continue
        index += 1

        plot.page += 1
        axes_label = plot_axes.get(variable, '<unknown variable>')

        headers, rows = make_table(
            series, group, variable, measure, axes_label)

        assert identifier is not None
        id_suffix = ""
        if data_len > 1:
            id_suffix = "-{}".format(index)

        gnuplot.output_plot(
            headers, rows, plotpath, gnuplot_script,
            title, specs, style, plot.page, identifier + id_suffix, axes_label,
            output=output, key_placement=key_placement, reps=reps
        )

        metadata_file.write("\n\n{0}\n{1}\n\n".format(
            title, identifier + id_suffix))

        keyvalpairs = series.values()[0].values()[0]['fixed'].items() + [
            ('variable', axes_label),
            ('measure', measure),
            ('grouping', group)]

        for k, v in keyvalpairs:
            if v != None:
                metadata_file.write("{k:<25} {v}\n".format(k=k, v=v))

        metadata_file.write(
            "\n" + textualtable.make_textual_table(headers, rows))

        id_headers, id_rows = make_table(
            series, group, variable, 'class', axes_label)

        def make_id(variable_value, item, variable):
            ret = "/".join([revision, item or '-'])
            if variable == 'dynamic_size':
                ret += "/" + str(variable_value)
            return ret

        id_rows = [
            [row[0]] +
            [make_id(row[0], item, variable) for item in row[1:]]
            for row in id_rows]

        ttable = textualtable.make_textual_table(id_headers, id_rows)
        metadata_file.write("\n" + ttable)

        if variable != 'direction' and variable != 'id':
            x, polys, residuals = linear_fit(rows)

            fitted_curves = []
            for i, xval in enumerate(x):
                current = [xval]
                current.extend(rows[i][1:])
                current.extend([numpy.polyval(polys[j], xval)
                                for j in range(0, len(rows[i]) - 1)])
                fitted_curves.append(current)

            plot.page += 1
            gnuplot.output_plot(
                headers + headers[1:], fitted_curves, plotpath, gnuplot_script,
                title, specs, 'fitted_lines', plot.page, identifier +
                id_suffix + '-fit', axes_label, output=output, reps=reps)

            def simplified_function(poly):
                return "{:.3g} * x {:+.3g}".format(poly[0], poly[1])
            metadata_file.write(
                "\npolynomial:\n" + textualtable.make_vertical_textual_table(
                    headers[1:], [map(simplified_function, polys)]))
            metadata_file.write(
                "\nresiduals:\n" + textualtable.make_vertical_textual_table(
                    headers[1:], [residuals]))
            metadata_file.write(
                "\nslope:\n" + textualtable.make_vertical_textual_table(
                    headers[1:], [map(lambda p: p[0], polys)]))
            metadata_file.write(
                "\nintercept:\n" + textualtable.make_vertical_textual_table(
                    headers[1:], [map(lambda p: p[1], polys)]))
    return data

plot.page = 0

def convert_to_seconds(value):
    if type(value) == int:
        strval = str(value)
        if convert_to_seconds == False:
            return strval
        strval = strval.zfill(10)
        strlen = len(strval)
        return float("{}.{}".format(
            strval[0:strlen-9],
            strval[strlen-9:]))
    return value

def make_table(series, group, variable, measure, axes_label):
    all_benchmark_variables_set = set()
    for bm_list in series.itervalues():
        all_benchmark_variables_set.update(bm_list.keys())

    all_benchmark_variables = sorted(list(all_benchmark_variables_set))

    rows = []

    headers = (
        [axes_label] +
        [k for k in series.iterkeys()]
    )

    for v in all_benchmark_variables:
        row = []
        row.append(v)
        for key, grp in series.iteritems():
            val = grp.get(v, {}).get(measure, None)
            if val is None:
                val = grp.get(v, {}).get('info', {}).get(measure, None)
            if measure == 'response_time':
                val = convert_to_seconds(val)
            row.append(val)
        rows.append(row)

    if variable == 'id':
        rows = sorted(rows, key=lambda x: x[1] or -1)

    return headers, rows


def binned_value(minimum, width, value):
    return width * (int(value - minimum) / int(width)) + minimum


def plot_distributions(
        all_benchmarks, output, plotpath, gnuplotcommands, bid,
        metadata_file, plot_type=None, latex=None, **kwargs):

    output_type = 'screen'
    if plot_type != 'animate':
        output_type = 'pdf'

    gnuplot.init(gnuplotcommands, output, bid, output_type=output_type)
    measure = 'response_time'

    keyset = set(all_benchmarks[0].keys()) - \
        set([measure, 'lineno', 'start', 'end'])
    comparison_function = functools.partial(comp_function, keyset)
    sorted_benchmarks = sorted(all_benchmarks, cmp=comparison_function)

    for group in group_by_keys(sorted_benchmarks, keyset):
        if plot_type != None:
            keyf = lambda x: x['lineno']
        else:
            keyf = lambda x: x[measure]

        frame_count = 1
        if plot_type != None:
            frame_count = 256

        current_frame = frame_count
        all_values = [b[measure] for b in sorted(group, key=keyf)]
        while current_frame > 0:

            if current_frame == frame_count:
                frame_ratio = 1
            else:
                frame_ratio = float(current_frame) / frame_count
            values = array(all_values[0:int(frame_ratio * len(all_values))])

            bin_width = 500
            min_x = numpy.amin(all_values)
            max_x = numpy.amax(all_values)

            bin_no = (max_x - min_x) / bin_width

            hgram, bin_edges = numpy.histogram(values, bins=max(bin_no, 10))

            mode = bin_edges[numpy.argmax(hgram)]
            min_x = mode - 100000
            max_x = mode + 100000

            if current_frame == frame_count:
                metadata_file.write(
                    'Direction {0}\n'.format(group[0]['direction']))

                gnuplotcommands.write(
                    gnuplot.templates['binned_init'].format(
                        title='%s %s' % (group[0]['id'], group[
                                         0]['direction']),
                        binwidth=bin_edges[1] - bin_edges[0],
                        min_x=min_x, max_x=max_x,
                        max_y=numpy.max(hgram)))

                if plot_type == 'animate':
                    gnuplotcommands.write('pause -1\n')

                elif plot_type == 'gradient':
                    gnuplotcommands.write("set multiplot\n")

            current_frame -= 1

            if plot_type == None:
                gnuplotcommands.write(
                    gnuplot.templates['binned_frame'].format(
                        datapoints='', color='#000033',
                        values='\n'.join(['{} {} {}'.format(val, count, val)
                                          for val, count in zip(
                                              bin_edges, hgram)])))

            elif plot_type == 'gradient':
                gnuplotcommands.write(
                    gnuplot.templates['binned_frame'].format(
                        datapoints='',
                        color=gnuplot.hex_color_gradient(
                            (125, 0, 0), (255, 255, 0), 1 - frame_ratio),
                        values='\n'.join(['{} {} {}'.format(val, count, val)
                                          for val, count in zip(
                                              bin_edges, hgram)])))

        gnuplotcommands.write("set xtics\n")
        gnuplotcommands.write("set ytics\n")

```

```

def plot_benchmarks(
        all_benchmarks, output, plotpath, gnuplotcommands, bid, metadata_file,
        plot_type=None, revision=None, checksum=None, latex=None):

    output_type = 'pdf'
    if latex == 'plotlatex':
        output_type = 'latex'
    elif latex == 'plotsvg':
        output_type = 'svg'

    gnuplot.init(gnuplotcommands, output, bid, output_type=output_type)

    type_counts = ["parameter_type_{t}_count".format(t=tp) for tp in types]
    keys_to_remove = type_counts[:]
    keys_to_remove.extend(
        ['parameter_type_count', 'single_type', 'dynamic_variation'])

    benchmarks = [bm for bm in all_benchmarks if bm['no'] != -1]
    defaults = [benchmarks, gnuplotcommands, plotpath]

#    analysis.calculate_overheads()
    overhead_estimates = {}
    overhead_benchmarks = [
        bm for bm in all_benchmarks
        if bm['no'] == -1 and 'Overhead' in bm ['id']]
    for loop_type in ['AllocOverhead', 'NormalOverhead']:
        for from_lang in ['C', 'J']:
            language_name = from_lang
            if language_name == 'J': language_name = 'Java'
            overhead_estimates[from_lang] = {}
            overhead_data = plot(
                overhead_benchmarks, gnuplotcommands, plotpath, metadata_file,
                style='simple_groups',
                key_placement=None,
                title='Mittauksen perusrasite ({})'.format(language_name),
                identifier='{}-{}'.format(loop_type.lower(), from_lang.lower()),
                keys_to_remove=[],
                select_predicate=(
                        lambda x: x['from'] == from_lang and loop_type in x['id'])
                group='from',
                measure='response_time',
                variable='description',
                revision=revision,
                checksum=checksum,
                output=output_type)

            if overhead_data == None:
                continue
            if len(overhead_data) > 1:
                print ('Error, more loop types than expected.',
                       len(overhead_data))
                exit(1)

            series = overhead_data[0]
            headers, rows = make_table(series,
                                       'from',
                                       'description',
                                       'response_time',
                                       'workload')
            est = estimate_measuring_overhead(rows[1:])
            overhead_estimates[from_lang][loop_type] = est[0]
            metadata_file.write('Overhead ' + from_lang + ' ' + str(est[0]))

    for i, ptype in enumerate(types):
        plot(
            benchmarks, gnuplotcommands, plotpath, metadata_file,
            title='{}-tyyppiset kutsuparametrit'.format(ptype),
            identifier='basic-call-{}'.format(ptype),
            style='simple_groups',
            keys_to_remove=(
                keys_to_remove +
                ['dynamic_size'] +
                ['has_reference_types']),
            select_predicate=lambda x: (
                x['single_type'] in [ptype, 'any'] and
                x['dynamic_size'] == 0),
            group='direction',
            variable='parameter_count',
            measure='response_time',
            revision=revision, checksum=checksum, output=output_type)

    for fr, to in DIRECTIONS:
        direction = format_direction(fr, to, latex)
        plot(
            benchmarks, gnuplotcommands, plotpath, metadata_file,
            title='Vaihteleva argumentin koko kutsusuunnassa ' + direction,
            identifier='variable-argument-size-{}-{}'.format(fr.lower(),
                                                              to.lower()),
            style='simple_groups',
            keys_to_remove=type_counts,
            select_predicate=(
                lambda x: (
                    x['direction'] == direction and
                    x['has_reference_types'] == 1 and
                    x['single_type'] in reference_types and
                    x['parameter_count'] == 1)),
            group='single_type',
            variable='dynamic_size',
            measure='response_time',
            revision=revision, checksum=checksum, output=output_type)

    for fr, to in DIRECTIONS:
        direction = format_direction(fr, to, latex)
        plot(
            benchmarks, gnuplotcommands, plotpath, metadata_file,
            title='Vaihteleva paluuarvon koko kutsusuunnassa ' + direction,
            identifier='variable-return-value-size-{}-{}'.format(fr.lower(),
                                                                 to.lower()),
            style='simple_groups',
            keys_to_remove=type_counts,
            select_predicate=(
                lambda x: x['has_reference_types'] == 1
                and x['direction'] == direction
                and x['return_type'] != 'void'),
            group='return_type',
            variable='dynamic_size',
            measure='response_time',
            revision=revision, checksum=checksum, output=output_type)

    keys_to_remove = type_counts[:]
    keys_to_remove.append('has_reference_types')
    keys_to_remove.append('dynamic_variation')

    for fr, to in DIRECTIONS:
        direction = format_direction(fr, to, latex)
        plot(
            benchmarks, gnuplotcommands, plotpath, metadata_file,
            style='simple_groups',
            title='Parametrityyppien vertailu ' + direction,
            identifier='basic-call-all-types-{}-{}'.format(fr.lower(),
                                                          to.lower()),
            keys_to_remove=keys_to_remove,
            select_predicate=(
                lambda x: x['direction'] == direction),
            group='single_type',
            variable='parameter_count',
            measure='response_time',
            revision=revision, checksum=checksum, output=output_type)

    plot(
        benchmarks, gnuplotcommands, plotpath, metadata_file,
        style='named_columns',
        title='Paluuarvon tyypit',
        identifier='return-value-types',
        keys_to_remove=['has_reference_types', 'dynamic_variation'],
        select_predicate=(
            lambda x: x['dynamic_size'] == 0 and
            x['return_type'] != 'void'),
        group='return_type',
        measure='response_time',
        variable='direction',
        min_series_width=2,
        revision=revision, checksum=checksum, output=output_type)
    # had: sort 'response_time', min_series_width: 2 , unused?

    def utf(b):
        return 'UTF' in b['id'] or 'Utf' in b['id']

    filters = {
        'utf': utf,
        'arrayregion': lambda x: 'ArrayRegion' in x['id'],
        'bytebufferview': lambda x: 'ByteBufferView' in x['id'],
        'unicode': lambda b: not utf(b) and 'String' in b['id'],
        'arrayelements': (lambda x:
                              'ArrayElements' in x['id'] or
                              'ArrayLength' in x['id'] or
                              'ReadPrimitive' in x['id']),
    }
    def uncategorized(x):
        for f in filters.values():
            if f(x):
                return False
        return True

    benchmarks = {}
    for key, f in filters.iteritems():
        benchmarks[key] = [
            bm for bm in all_benchmarks
            if bm['no'] == -1 and f(bm)]

    benchmarks['uncategorized'] = [
        bm for bm in all_benchmarks
        if bm['no'] == -1 and 'Overhead' not in bm['id'] and uncategorized(bm)]

    custom_benchmarks = benchmarks['uncategorized']

    for fr, to in DIRECTIONS:
        direction = format_direction(fr, to, latex)
        plot(
            custom_benchmarks, gnuplotcommands, plotpath, metadata_file,
            style='simple_groups',
            title='Erityiskutsut suunnassa ' + direction,
            identifier='special-calls-{}-{}'.format(fr.lower(), to.lower()),
            select_predicate=(
                lambda x: (x['direction'] == direction and
                           x['dynamic_variation'] == 1)),
            group='id',
            measure='response_time',
            variable='dynamic_size',
            revision=revision, checksum=checksum, output=output_type)

        plot(
            benchmarks['arrayregion'], gnuplotcommands, plotpath,
            metadata_file,
            style='simple_groups',
            title='Erityiskutsut suunnassa ' + direction,
            identifier='special-calls-arrayregion-{}-{}'.format(fr.lower(),
                                                                to.lower()),
            select_predicate=(
                lambda x: (x['direction'] == direction and
                           x['dynamic_variation'] == 1)),
            group='id',
            measure='response_time',
            variable='dynamic_size',
            revision=revision, checksum=checksum, output=output_type)

        plot(
            benchmarks['arrayelements'], gnuplotcommands, plotpath,
            metadata_file,
            style='simple_groups',
            title='Erityiskutsut suunnassa ' + direction,
            identifier='special-calls-arrayelements-{}-{}'.format(fr.lower(),
                                                                  to.lower()),
            select_predicate=(
                lambda x: (x['direction'] == direction and
                           x['dynamic_variation'] == 1)),
            group='id',
            measure='response_time',
            variable='dynamic_size',
            revision=revision, checksum=checksum, output=output_type)

        plot(
            benchmarks['utf'], gnuplotcommands, plotpath, metadata_file,
            style='simple_groups',
            title='UTF-merkkijonot suunnassa ' + direction,
            identifier='special-calls-utf-{}-{}'.format(fr.lower(),
                                                        to.lower()),
            select_predicate=(
                lambda x: (x['direction'] == direction and
                           x['dynamic_variation'] == 1)),
            group='id',
            measure='response_time',
            variable='dynamic_size',
            revision=revision, checksum=checksum, output=output_type)

        plot(
            benchmarks['unicode'], gnuplotcommands, plotpath, metadata_file,
            style='simple_groups',
            key_placement='inside bottom left',
            title='Unicode-merkkijonot suunnassa ' + direction,
            identifier='special-calls-unicode-{}-{}'.format(fr.lower(),
                                                            to.lower()),
            select_predicate=(
                lambda x: (x['direction'] == direction and
                           x['dynamic_variation'] == 1)),
            group='id',
            measure='response_time',
            variable='dynamic_size',
            revision=revision, checksum=checksum, output=output_type)

        plot(
            benchmarks['bytebufferview'], gnuplotcommands, plotpath,
            metadata_file,
            style='simple_groups',
            title='Erityiskutsut suunnassa ' + direction,
            identifier='special-calls-bytebufferview-{}-{}'.format(fr.lower(),
                                                                  to.lower()),
            select_predicate=(
                lambda x: (x['direction'] == direction and
                           x['dynamic_variation'] == 1 and
                           'Bulk' not in x['id'])),
            group='id',
            measure='response_time',
            variable='dynamic_size',
            revision=revision, checksum=checksum, output=output_type)

        plot(
            benchmarks['bytebufferview'], gnuplotcommands, plotpath,
            metadata_file,
            style='simple_groups',
            title='Erityiskutsut suunnassa ' + direction,
            identifier='special-calls-bulk-bytebufferview-{}-{}'.format(
                fr.lower(), to.lower()),
            select_predicate=(
                lambda x: (x['direction'] == direction and
                           x['dynamic_variation'] == 1 and
                           'Bulk' in x['id'])),
            group='id',
            measure='response_time',
            variable='dynamic_size',
            revision=revision, checksum=checksum, output=output_type)

    plot(
        custom_benchmarks, gnuplotcommands, plotpath, metadata_file,
        style='histogram',
        title='Erityiskutsujen vertailu eri kutsusuunnissa',
        identifier='special-calls-non-dynamic',
        select_predicate=(
            lambda x: (
                x['dynamic_variation'] == 0 and
                'Field' in x['id'])),
        group='direction',
        measure='response_time',
        variable='id',
        revision=revision, checksum=checksum, output=output_type)


MEASUREMENT_FILE = 'measurements.txt'
DEVICE_PATH = '/sdcard/results'
PLOTPATH = '/tmp'
TOOL_NAMESPACE = 'fi.helsinki.cs.tituomin.nativebenchmark.measuringtool'


def sync_measurements(dev_path, host_path, filename, update=True):
    old_path = host_path + '/' + filename
    tmp_path = '/tmp/' + filename
    if not update and os.path.exists(old_path):
        print 'No sync necessary'
        return

    kwargs = {}
    if FNULL is not None:
        kwargs['stdout'] = FNULL
        kwargs['stderr'] = FNULL

    try:
        success = call(['adb', 'pull',
                        dev_path + '/' + filename,
                        tmp_path], **kwargs)
    except OSError:
        success = -1
    if success == 0:
        if os.path.exists(old_path):
            size_new = os.path.getsize(tmp_path)
            size_old = os.path.getsize(old_path)
            if size_new < size_old:
                print ("Warning: new file contains less data than "
                       "the old. Aborting.")
                exit(2)
        shutil.move(tmp_path, old_path)

    else:
        print "Could not get new measurements, continuing with old."

def render_perf_reports_for_measurement(identifier, measurements,
                                        measurement_path, output_path,
                                        output_command=False):
    path = identifier.split("/")
    if len(path) < 2:
        print 'Invalid identifier {}'.format(identifier)
        exit(1)
    if len(path) == 3:
        revision, class_, dynamic_size = path
    elif len(path) == 2:
        revision, class_ = path
        dynamic_size = None

    def match_measurement(measurement):
        m = measurement[0]
        return (m.get('code-revision') == revision and
                m.get('tool') == 'LinuxPerfRecordTool')

    def match_measurement_run(m):
        if m.get('class').lower() != class_.lower():
            return False
        if dynamic_size and m.get('dynamic_size') != int(dynamic_size):
            return False
        if 'Filename' not in m or m['Filename'] is None:
            return False
        return True

    datafiles = []
    for measurement in filter(match_measurement, measurements):
        mid = measurement[0].get('id')
        zpath = os.path.join(measurement_path, 'perfdata-{}.zip'.format(mid))
        try:
            measurement_zipfile = zipfile.ZipFile(zpath, 'r')
            datafiles.append({
                'zip': measurement_zipfile,
                'zip_path': zpath,
                'mid': mid,
                'csv': measurement_zipfile.open(
                    '{0}/benchmarks-{0}.csv'.format(mid))
            })
        except zipfile.BadZipfile:
            print 'Bad zip file %s' % zpath
        except IOError as e:
            print 'Problem with zip file %s' % zpath
            print e

    benchmarks = []
    for df in datafiles:
        benchmarks.append({
            'zip': df['zip'],
            'mid': df['mid'],
            'metadata': read_datafiles([df['csv']], silent=output_command)
        })

    matching_benchmarks = []
    for bm in benchmarks:
        for row in bm['metadata']:
            if match_measurement_run(row):
                matching_benchmarks.append({
                    'zip': bm['zip'],
                    'mid': bm['mid'],
                    'filename': row['Filename']
                })

    for record in matching_benchmarks:
        perf_file = record['zip'].extract('{}/{}'.format(record['mid'],
                                                         record['filename']),
                                          '/tmp')
        try:
            command_parts = [
                "perf report",
                "-i {}",
                "--header",
                "--symfs=/home/tituomin/droid-symbols",
                "--kallsyms=/home/tituomin/droid/linux-kernel/kallsyms"
            ]
            command_parts.extend([
                "-g graph,0,caller",
                "--stdio",
                "| c++filt",
                ">/tmp/out.txt"
            ])
            command = " ".join(command_parts).format(perf_file)
            if output_command:
                print command
                exit(0)
            else:
                call([command], shell=True)
        except OSError as e:
            print e.filename, e.message, e.args

    for f in datafiles:
        f['zip'].close()
    print "Profile for identifier", identifier
    with open('/tmp/out.txt', 'r') as f:
        print f.read()
    exit(0)

if __name__ == '__main__':
    if len(argv) < 4 or len(argv) > 6:
        print argv[0]
        print ("\n    Usage: %s input_path output_path "
               "limit [pdfviewer] [separate]\n").format(argv[0])
        exit(1)

    FNULL = open(os.devnull, 'w')

    method = argv[0]
    measurement_path = os.path.normpath(argv[1])
    output_path = argv[2]

    if 'plotlatex' in method:
        latex = 'plotlatex'
        method = 'curves'
    elif 'plotsvg' in method:
        latex = 'plotsvg'
        method = 'curves'
    else:
        latex = None

    output_command = False
    if len(argv) > 5:
        if argv[5] == 'show-command':
            output_command = True

    limit = argv[3]
    if len(argv) > 4:
        pdfviewer = argv[4]
    else:
        pdfviewer = None

    if len(argv) == 6:
        group = (not argv[5] == "separate")
    else:
        group = True

    if output_command:
        system_stdout = sys.stdout
        system_stderr = sys.stderr
        sys.stdout = FNULL
        sys.stderr = FNULL

    sync_measurements(DEVICE_PATH, measurement_path, MEASUREMENT_FILE)

    f = open(os.path.join(measurement_path, MEASUREMENT_FILE))

    try:
        measurements = read_measurement_metadata(f, group)
    finally:
        f.close()

    limited_measurements = (
        filter(lambda x: int(x[0].get('repetitions', 0)) >= int(limit),
               measurements.values()))

    # ID = revision/checksum/class[/dynamic_size]
    if 'perf_select' in method:
        identifier = argv[4]
        if output_command:
            sys.stdout = system_stdout
            sys.stderr = system_stderr
            FNULL.close()
        render_perf_reports_for_measurement(
            identifier, limited_measurements, measurement_path,
            output_path, output_command=output_command)
        exit(0)

    csv_files = set()
    for f in glob.iglob(measurement_path + '/benchmarks-*.csv'):
        try:
            csv_files.add(f.split('.csv')[0].split('benchmarks-')[1])
        except IndexError:
            pass

    if len(limited_measurements) > 20:
        i = len(limited_measurements) - 20 + 1
        splice = limited_measurements[-20:]
    else:
        i = 1
        splice = limited_measurements

    print "\nAvailable compatible measurements. Choose one"
    for m in splice:
        b = m[0]
        warning = ""
        if int(b.get('rounds')) == 0:
            warning = " <---- WARNING INCOMPLETE MEASUREMENT"
        print """
    [{idx}]:     total measurements: {num}
                           local: {local}
                     repetitions: {reps}
                     description: {desc}
                          rounds: {rounds}{warning}
                              id: {mid}
                        checksum: {ck}
                        revision: {rev}
                            tool: {tool}
                             cpu: {freq} KHz
                             set: {bset}
                          filter: {sfilter}
                           dates: {first} -
                                  {last}
    """.format(
            local=b.get('id') in csv_files,
            num=len(m),
            mid=b.get('id'),
            idx=i,
            warning=warning,
            last=m[-1]['end'],
            rounds=reduce(lambda x, y: y + x, [int(b['rounds']) for b in m]),
            reps=b.get('repetitions'),
            ck=b.get('code-checksum'),
            rev=b.get('code-revision'),
            tool=b.get('tool'),
            freq=b.get('cpu-freq'),
            bset=b.get('benchmark-set'),
            desc=b.get('description'),
            sfilter=b.get('substring-filter'),
            first=b.get('start')
        )

        i += 1

    try:
        response = raw_input("Choose set 1-{last} >> ".format(last=i - 1))
    except EOFError:
        print 'Exiting.'
        exit(1)

    benchmark_group = limited_measurements[int(response) - 1]

    filenames = []
    ids = []
    multiplier = 0
    for measurement in benchmark_group:
        if 'LinuxPerfRecordTool' in measurement['tool']:
            basename = "perfdata-{n}.zip"
        else:
            basename = "benchmarks-{n}.csv"
        filenames.append(
            basename.format(n=measurement['id']))
        if 'logfile' in measurement:
            filenames.append(measurement['logfile'])
        ids.append(measurement['id'])
        multiplier += int(measurement['rounds'])

    files = []
    for filename in filenames:
        sync_measurements(DEVICE_PATH, measurement_path,
                          filename, update=False)
        if filename not in [m.get('logfile') for m in benchmark_group]:
            files.append(open(os.path.join(measurement_path, filename)))

    first_measurement = benchmark_group[0]

    global_values = {
        'repetitions': first_measurement['repetitions'],
        'is_allocating': first_measurement['benchmark-set'] == 'ALLOC',
        'multiplier': multiplier
    }

    perf = False
    if 'LinuxPerfRecordTool' in first_measurement['tool']:
        print 'Perf data downloaded.'
        perf = True
    if not perf:
        try:
            benchmarks = read_datafiles(files)

        finally:
            for f in files:
                f.close()

        benchmark_group_id = os.getenv('PLOT_ID', str(uuid.uuid4()))
        plot_prefix = 'plot-{0}'.format(benchmark_group_id)

        if latex is not None:
            output_filename = os.path.join(output_path, plot_prefix)
        else:
            output_filename = os.path.join(output_path, plot_prefix + '.pdf')
        plot_filename = plot_prefix + '.gp'

        plotfile = open(os.path.join(output_path, plot_filename), 'w')
        metadata_file = open(os.path.join(
            output_path, plot_prefix + '-metadata.txt'), 'w')

        measurement_ids = " ".join(ids)
        metadata_file.write("-*- mode: perf-report; -*-\n\n")
        metadata_file.write("id: {0}\n".format(benchmark_group_id))
        metadata_file.write("measurements: {0}\n".format(measurement_ids))

        benchmarks = preprocess_benchmarks(benchmarks, global_values,
                                           latex=latex)

        animate = False
        if pdfviewer == 'anim':
            plot_type = 'animate'
            pdfviewer = None
        elif pdfviewer == 'gradient':
            plot_type = 'gradient'
            pdfviewer = None
        else:
            plot_type = None

    if 'curves' in method:
        function = plot_benchmarks
    elif 'distributions' in method:
        function = plot_distributions
    if perf or not function:
        exit(0)

    function(
        benchmarks,
        output_filename,
        PLOTPATH,
        plotfile,
        benchmark_group_id,
        metadata_file,
        plot_type=plot_type,
        revision=first_measurement['code-revision'],
        checksum=first_measurement['code-checksum'],
        latex=latex)

    plotfile.flush()
    plotfile.close()
    if plot_type == 'animate':
        print "Press enter to start animation."
    call(["gnuplot", plotfile.name])
    if pdfviewer:
        call([pdfviewer, str(output_filename)])
    print "Final plot",
    if 'animate' != plot_type:
        print str(output_filename)
    else:
        print str(plot_filename)
    print(benchmark_group_id)
    exit(0)
```

textualtable.py

```python
#/usr/bin/python

def make_textual_table(headers, rows):
    result = ""
    max_widths = []

    for x in headers:
        max_widths.append(len(str(x)))

    for row in rows:
        for i, x in enumerate(row):
            l = len(str(x))
            if max_widths[i] < l:
                max_widths[i] = l

    row_format = ["{{:>{w}}}   ".format(w=w) for w in max_widths]
    row_format = "".join(row_format) + "\n"

    result += row_format.format(*headers)
    for row in rows:
        result += row_format.format(*row)
    return result

def make_vertical_textual_table(headers, elements):
    result = ""
    max_width = max((len(x) for x in headers))

    header_format = "{{:>{w}}}".format(w=max_width)

    for i in range(0, len(headers)):
        result += header_format.format(headers[i])
        for group in elements:
            result += "    "
            result += str(group[i])
        result += "\n"

    return result
```