



ATYPON

Mapreduce framework.

Sami Mohammed Mahmoud Shabaneh

Major: Fifth year of computer Engineering

University of Jordan.

Project's Features:

- 1- A simple easy to use GUI, with instant feedback to user.

The screenshot shows a web-based GUI titled "Map Reduce". It has a sidebar on the left with labels for different sections: "Number Of Mappers", "Number Of Reducers", "Text File Path", "Where to execute (swarm/locally?", "Custom Imports", "Mapper Function", "Reducer Function", and "OutPuts". The main area on the right contains input fields and code editors for each of these sections. The "Where to execute" field has a dropdown menu with the option "swarm/locally?". The "Custom Imports" field contains the text "//please list all your imports here.". The "Mapper Function" field contains the code `public static Map<?,?> mapping(String file){` followed by a closing brace. The "Reducer Function" field contains the code `public static Map<?,?> reduce(Map<Object, List<Object>> map){` followed by a closing brace. At the bottom right of the main area is an "Execute" button. Below the button, the "OutPuts" section displays the text "management server started at Thu Jan 30 00:37:56 EET 2020".

- 2- Scalable decoupled Workflow implementation which can be rolled back on fail. And easy to add to phases from XML parser.
- 3- Highly scalable run on swarm.

Contents

1. Clean Code:	4
1.1. Comments:.....	4
1.2. Environment	5
1.3. Functions	5
1.4. General:	6
1.5. JAVA.....	18
2. Effective JAVA:	21
3. Design Patterns:	34
3.1 Strategy patterns:	34
3.2. Command Pattern	35
3.3. Observer Pattern (Not Implemented).....	36
3.4. SINGLETONS:	36
4. Styling Guide:.....	37
5. DevOps:.....	37

1. Clean Code:

1.1. Comments:

C1: Inappropriate Information:

All comments are appropriate and reserved for technical notes about the code and design.

```
public static void startCollecting(int numberOfReducers) throws IOException {  
    allDataCollectedLatch = new CountDownLatch(numberOfReducers); // initialize the latch to let the method receive data from all reducers.  
    finalResult = new ConcurrentSkipListMap<>();  
    try (ServerSocket server = new ServerSocket(Constants.COLLECTOR_PORT)) {  
        AtomicInteger dataReceivedFromReducers = new AtomicInteger( initialValue: 0); // keep track of number of data received.  
        while (dataReceivedFromReducers.get() != numberOfReducers) {  
            Socket skt = server.accept();  
            dataReceivedFromReducers.getAndIncrement();  
        }  
    }  
}
```

C2: Obsolete Comment:

There are no obsolete Comments.

C3: Redundant Comment:

No redundant comments.

C4: Poorly Written Comment:

Comments are minimized. And explain exactly what they are doing.

C5: Commented-Out Code:

No codes are commented out.

1.2. Environment

Not applicable.

1.3. Functions

F1: Too Many Arguments:

All functions arguments are kept at a minimum with maximum of 3 args arguments. (because there are no other better way around)

```
public static <T> void sendObject(String address, int port, T dataToSend) throws IOException {  
    try (Socket sk = new Socket(address, port);  
        ObjectOutputStream objectOutput = new ObjectOutputStream(sk.getOutputStream())) {  
        objectOutput.writeObject(dataToSend);  
    } catch (Exception e) {  
    }  
}
```

F2: Output Arguments:

There are No output arguments.

F3: Flag Arguments:

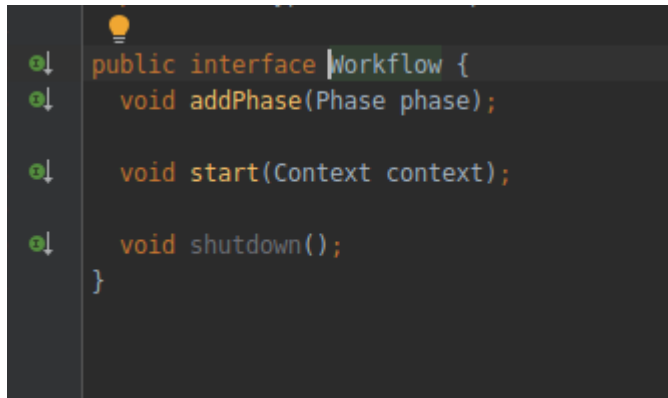
No Functions are taking flags.

F4: Dead Function:

All methods are being used. Unless there are in, to help for Later expanding.

Example of unused method is shutdown() this define how the workflow should shutdown if it has been called. However i did put it if there are future plan to define a shutdown process.

But it is not used in current Workflows.

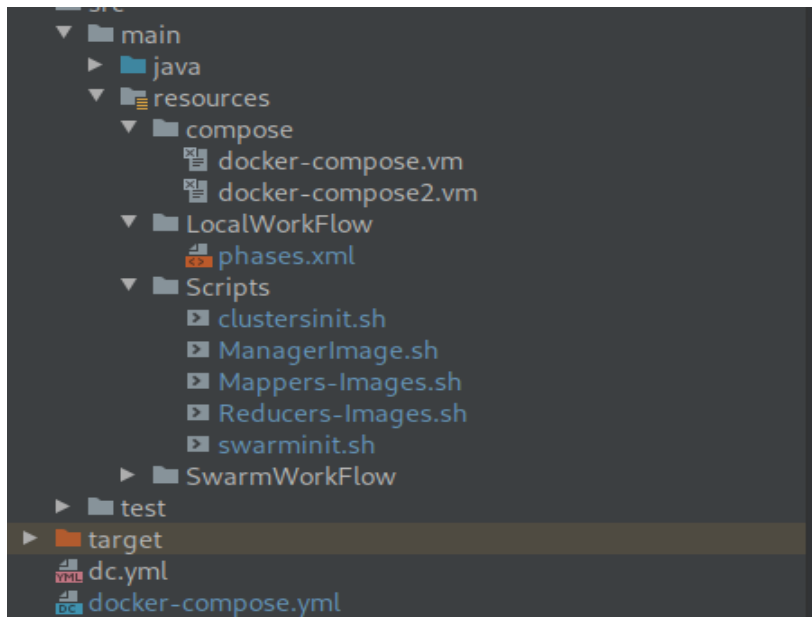


```
public interface Workflow {  
    void addPhase(Phase phase);  
  
    void start(Context context);  
  
    void shutdown();  
}
```

1.4. General:

G1: *Multiple Languages in One Source File:*

XML files , VM files and Java file , Scripts file are all separated in there directories. However this is not applicable on 1 command process calls. To avoid unnecessary complexity.



G2: Obvious Behavior Is Unimplemented:

All functions do what they are expected to do.

G3: Incorrect Behavior at the Boundaries:

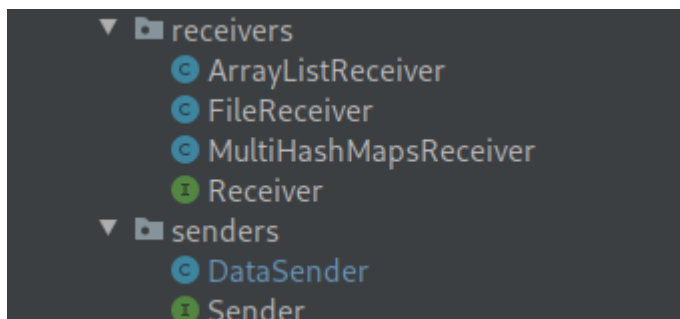
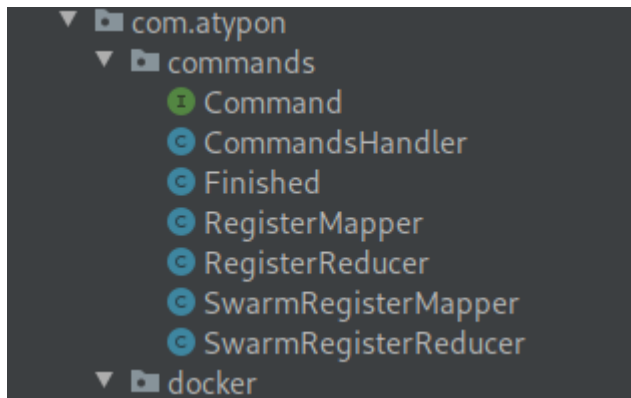
No automated testing, however the code Manually tested on multiple Mappers/Reducers multiple file sizes from 1mb to 4gb.

G4: Overridden Safeties:

Not applicable.

G5: Duplication:

Duplication has been eliminated using Commands/Strategy design patterns.



However some duplications are present in some phases. However this is not an issue since phases are separated from the code itself.

G6: Code at Wrong Level of Abstraction:

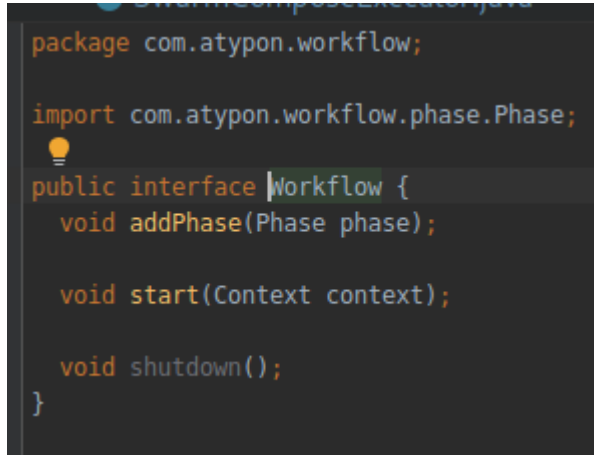
No codes are in there wrong level of Abstraction.

G7: Base Classes Depending on Their Derivatives:

Base classes doesn't depend on their Derivatives.

G8: Too Much Information:

Interfaces methods are kept at minimum, classes are as small as possible, Coupling is very low.

A screenshot of a code editor showing a Java interface definition. The code is as follows:

```
package com.atypon.workflow;  
  
import com.atypon.workflow.phase.Phase;  
  
public interface Workflow {  
    void addPhase(Phase phase);  
  
    void start(Context context);  
  
    void shutdown();  
}
```

G9: Dead Code:

No dead codes. If statements are reasonably used.

G10: Vertical Separation:

Variables are declared above there usage, same goes for private functions ..

```

public class ContainersHandler {

    private static ContainersHandler containersDataHandler = null;
    private ContainersDataTracker containersDataTracker;

    private ContainersHandler() { containersDataTracker = ContainersDataTracker.getInstance(); }

    public static ContainersHandler getInstance() {
        if (containersDataHandler == null) {
            containersDataHandler = new ContainersHandler();
        }

        return containersDataHandler;
    }

    public void sendReducerAddressesToMappers() throws InterruptedException {
        Thread.sleep( millis: 3000);
        for (String address : containersDataTracker.getMappersAddresses()) {
            Thread t =
                new Thread(
                    () -> {
                        try (Socket sk = new Socket(address, Constants.MAINSERVER_TO_MAPPERS_PORT)) {
                            ObjectOutputStream objectOutput = new ObjectOutputStream(sk.getOutputStream());
                            objectOutput.writeObject(containersDataTracker.getReducersAddresses());
                        } catch (Exception e) {
                            e.printStackTrace();
                        }
                    }
                );
            t.start();
            t.join();
        }
    }
}

```

G11: Inconsistency:

Kept at a minimum. Behaviours that been done in one place is done following other places..

G12: Clutter:

All constructors have implementations, unless it is private.

G13: Artificial Coupling:

Methods and variables are defined in there place.

G14: *Feature Envy:*

The containers handler class (contains method on containersData) doesn't manipulate Data in anyway.

G15: *Selector Arguments:*

Methods doesn't take flags.

G16: *Obscured Intent:*

Not applicable.

G17: *Misplaced Responsibility:*

Following single *responsibility* rules. I believe methods are in there correct places.

G18: *Inappropriate Static:*

Current Static functions have no reason to change.

G19: *Use Explanatory Variables:*

No explanatory variables however similar behaviour is implemented in phases and pipelining.

G20: *Function Names Should Say What They Do:*

All methods do what their names tell.

```
public static String readFileAsString(String filePath) {  
    String content = "";  
  
    try {  
        content = new String(Files.readAllBytes(Paths.get(filePath)));  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
  
    return content;  
}
```

G21: Understand the Algorithm:

I wrote the functions, I know how they work.

G22: Make Logical Dependencies Physical:

All dependent entities has been put in Constants class.

```

package com.atypon.utility;

public class Constants {

    public static final int MAIN_SERVER_PORT = 7777;
    public static final int MAPPERS_TO_REDUCERS_PORT = 8787;
    public static final int MAINSERVER_TO_MAPPERS_PORT = 9090;
    public static final int MAINSERVER_TO_REDUCERS_PORT = 9091;
    public static final int MAPPERS_FILE_RECEIVER_PORT = 6666;
    public static final int COLLECTOR_PORT = 9999;
    public static final int SWARM_FORWARDER_PORT = 7676;

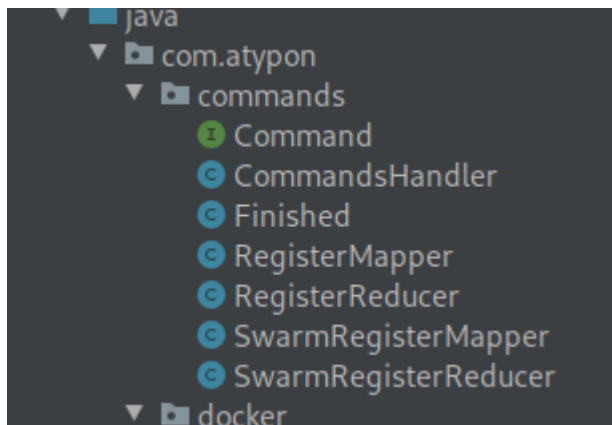
    public static final String MAIN_SERVER_IP = "192.168.8.100";

    private Constants() {
    }
}

```

G23: Prefer Polymorphism to If/Else or Switch/Case:

Solved with Command pattern, rather than switching the socket input and make a decision based on it.



G24: Follow Standard Conventions:

Google style guide for java has been used.

G25: Replace Magic Numbers with Named Constants:

All magic numbers are replaced.

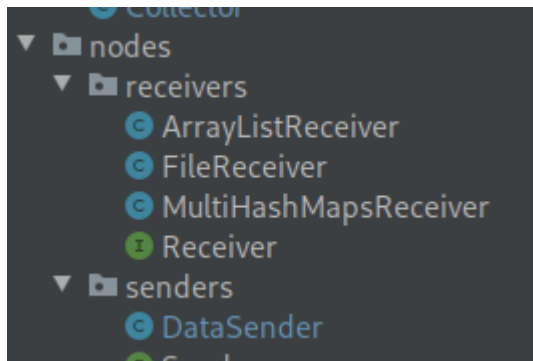
```
public class Constants {  
  
    public static final int MAIN_SERVER_PORT = 7777;  
    public static final int MAPPERS_TO_REDUCERS_PORT = 8787;  
    public static final int MAINSERVER_TO_MAPPERS_PORT = 9090;  
    public static final int MAINSERVER_TO_REDUCERS_PORT = 9091;  
    public static final int MAPPERS_FILE_RECEIVER_PORT = 6666;  
    public static final int COLLECTOR_PORT = 9999;  
    public static final int SWARM_FORWARDER_PORT = 7676;  
  
    public static final String MAIN_SERVER_IP = "192.168.8.100";  
  
    private Constants() {  
    }  
}
```

G26: Be Precise:

Race conditions and other has been solved. WorstCases are put in mind.

G27: Structure over Convention:

Strategy pattern used.



G28: Encapsulate Conditionals:

Not applicable, used 1 time to avoid unnecessary complexity.

```
try (ServerSocket server = new ServerSocket(Constants.COLLECTOR_PORT)) {
    AtomicInteger dataReceivedFromReducers = new AtomicInteger(0); // keep track of number of data received.
    while (dataReceivedFromReducers.get() != numberOfReducers) { //exit when the data has been received from all reducers
        Socket skt = server.accept();
        dataReceivedFromReducers.getAndIncrement();
        Thread t =
            new Thread(
```

G29: Avoid Negative Conditionals:

Avoided

```
if (FilesUtil.checkIfNotExist(context.getParam( paramName: "txtFilePath"))) {
    Main.appendText("TEXT FILE NOT FOUND, make sure it is readable");
    throw new PhaseExecutionFailed("TEXT File Not Found");
}
```

G30: Functions Should Do One Thing:

Based on Single Responsibility rule. All functions do one things. And what their names suggest. However the prepareUserCode is built with this in my mind. And it compiles the given user code. To work as one method.

G31: Hidden Temporal Couplings

The structure of methods is obvious.

```
@Override
public Workflow parse(String data) {
    Workflow workflow = new WorkflowImp();
    List<Phase> phases = createPhases(data);
    phases.forEach(workflow::addPhase);
    return workflow;
}
```

G32: Don't Be Arbitrary

No class scoped inside any other class.

G33: Encapsulate Boundary Conditions

Discussed in G28.

G35: Keep Configurable Data at High Levels


```

package com.atypon.utility;

public class Constants {

    public static final int MAIN_SERVER_PORT = 7777;
    public static final int MAPPERS_TO_REDUCERS_PORT = 8787;
    public static final int MAINSERVER_TO_MAPPERS_PORT = 9090;
    public static final int MAINSERVER_TO_REDUCERS_PORT = 9091;
    public static final int MAPPERS_FILE_RECEIVER_PORT = 6666;
    public static final int COLLECTOR_PORT = 9999;
    public static final int SWARM_FORWARDER_PORT = 7676;

    public static final String MAIN_SERVER_IP = "192.168.8.100";

    private Constants() {
    }
}

```

G36: Avoid Transitive Navigation

No object call more than one method.

```

@Override
public void execute() {
    containersDataTracker.addReducerAddress(socket.getInetAddress().toString().substring(1));
    containersDataTracker.incrementRunningContainers();
    containersDataTracker.incrementRunningReducers();
    Main.appendText(
        "Registered reducer "
        + containersDataTracker.getCurrentReducersRunning()
        + " / "
        + containersDataTracker.getNumOfReducer()
        + " "
        + socket.getInetAddress()
        + '\n');
}
}

```

Avoided.

1.5. JAVA

J1: Avoid Long Import Lists by Using Wildcards:

Not always avoided.

J2: Don't Inherit Constants

No constants are inherited.

J3: Constants versus Enums:

Not applicable. More on effective Java.

Names:

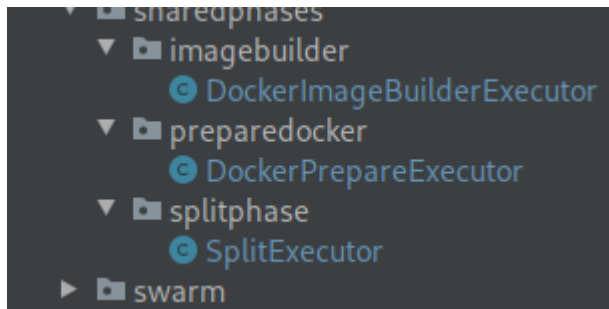
N1: *Choose Descriptive Names:*

```
private static ContainersDataTracker containersDataTracker = null;

@Setter
@Getter
private AtomicInteger numOfContainers;
@Setter
@Getter
private AtomicInteger numOfMappers;
@Setter
@Getter
private AtomicInteger numOfReducer;
@Setter
@Getter
private AtomicInteger runningContainers;|
@Getter
private AtomicInteger currentMappersRunning;
@Getter
private AtomicInteger currentReducersRunning;
```

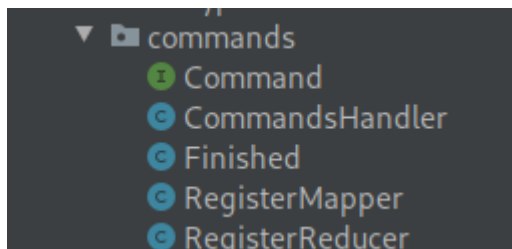
N2: Choose Names at the Appropriate Level of Abstraction:

Here i have named all phases executors with there appropriate name.



N3: Use Standard Nomenclature Where Possible:

Command pattern



N4: Unambiguous Names:

```
public void sendMappersAddressesToReducers() throws InterruptedException, IOException {
    Thread.sleep( millis: 3000);
    for (String address : containersDataTracker.getReducersAddresses()) {
        try (Socket sk = new Socket(address, Constants.MAINSERVER_TO_REDUCERS_PORT)) {
            ObjectOutputStream objectOutput = new ObjectOutputStream(sk.getOutputStream());
            objectOutput.writeObject(containersDataTracker.getMappersAddresses());
        } catch (Exception e) {
            throw e;
        }
    }
}
```

N5: Use Long Names for Long Scopes:

```
List<String> filesAbsPath = FilesUtil.GetFilesAbsPathInDirectory(rootDirectory);
int i = 0;
for (String fileAbsPath : filesAbsPath) {
    System.out.println("Sending : " + fileAbsPath);
    FilesUtil.fileUploader(containersDataTracker.getMappersAddresses().get(i), fileAbsPath);
    System.out.println(containersDataTracker.getMappersAddresses().get(i));
    i++;
}
```

N6: Avoid Encodings:

All names are written in english.

2. Effective JAVA:

ITEM 1: CONSIDER STATIC FACTORY METHODS INSTEAD OF CONSTRUCTORS:

Used in singletons.

```
public class ContainersHandler {  
  
    private static ContainersHandler containersDataHandler = null;  
    private ContainersDataTracker containersDataTracker;  
  
    private ContainersHandler() { containersDataTracker = ContainersDataTracker.getInstance(); }  
  
    public static ContainersHandler getInstance() {  
        if (containersDataHandler == null) {  
            containersDataHandler = new ContainersHandler();  
        }  
  
        return containersDataHandler;  
    }  
}
```

ITEM 2: CONSIDER  A BUILDER WHEN FACED WITH MANY CONSTRUCTOR ARGUMENTS:

Not applicable.

ITEM 3: ENFORCE SINGLETON PROPERTY WITH A PRIVATE CONSTRUCTOR OR AN ENUM TYPE.

```
public class ContainersHandler {  
  
    private static ContainersHandler containersDataHandler = null;  
    private ContainersDataTracker containersDataTracker;  
  
    private ContainersHandler() { containersDataTracker = ContainersDataTracker.getInstance(); }  
  
    public static ContainersHandler getInstance() {  
        if (containersDataHandler == null) {  
            containersDataHandler = new ContainersHandler();  
        }  
  
        return containersDataHandler;  
    }  
}
```

// TODO: replace with executioner

ITEM 4: ENFORCE NONINSTANTIABILITY WITH A PRIVATE CONSTRUCTOR.

Used in singleton and utility classes.

```
public class FilesUtil {  
  
    private FilesUtil() {  
    }  
  
    @ public static void copyToDir(File srcPath, File dstWithName) throws IOException {  
        Files.copy(srcPath.toPath(), dstWithName.toPath(), StandardCopyOption.REPLACE_EXISTING);  
    }  
  
    public static Boolean checkIfNotExist(String path) {  
        File file = new File(path);  
        return !file.exists();  
    }  
}
```

ITEM 5: PREFER DEPENDENCY INJECTION TO HARDWIRING RESOURCES.

```
import java.util.Map;

public class Context {

    private Map<String, Object> params;

    public Context(Map<String, Object> params) {
        this.params = params;
    }

    /unchecked/
    public <T> T getParam(String paramName) {
        return (T) params.get(paramName);
    }
}
```

ITEM 6: AVOID CREATING UNNECESSARY OBJECTS.

Avoided by singleton. And string builders.

ITEM 7 + 8:

Not applicable.

ITEM 9: PREFER TRY-WITH-RESOURCES TO TRY-FINALLY.

Always used.

```
public static void fileUploader(String address, String fileAbsPath, int port) throws IOException {  
    File f = new File(fileAbsPath);  
  
    try (Socket socket = new Socket(address, port);  
        InputStream in = new FileInputStream(f);  
        OutputStream out = socket.getOutputStream()) {  
  
        byte[] bytes = new byte[8192];  
        int count;  
        while ((count = in.read(bytes)) > 0) {  
            out.write(bytes, 0, count);  
        }  
    } catch (Exception e) {  
        throw e;  
    }  
}
```

ITEM 10 + 11 + 12 + 13 + 14:

Not applicable.

Classes and Interfaces

ITEM 15: MINIMIZE THE ACCESSIBILITY OF CLASSES AND MEMBERS.

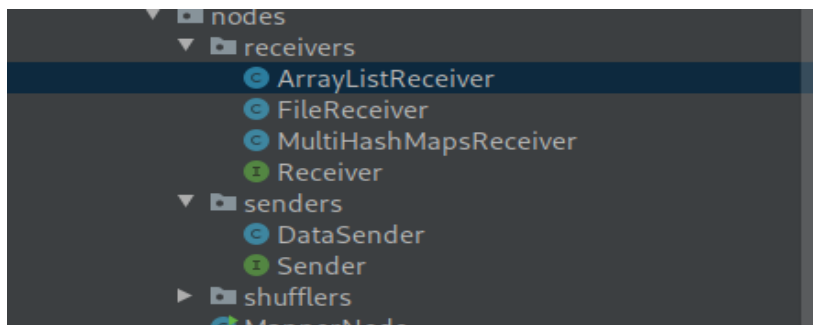
ITEM 16: IN PUBLIC CLASSES, USE ACCESSOR METHODS, NOT PUBLIC FIELDS.

```
public class ContainersDataTracker {  
  
    private static ContainersDataTracker containersDataTracker = null;  
  
    @Setter  
    @Getter  
    private AtomicInteger numOfContainers;  
    @Setter  
    @Getter  
    private AtomicInteger numOfMappers;  
    @Setter  
    @Getter  
    private AtomicInteger numOfReducer;  
    @Setter  
    @Getter  
    private AtomicInteger runningContainers;  
    @Getter  
    private AtomicInteger currentMappersRunning;  
    @Getter  
    private AtomicInteger currentReducersRunning;  
    @Getter  
    private AtomicInteger finishedMappers;  
  
    @Getter  
    private ArrayList<String> mappersAddresses;  
    @Getter  
    private ArrayList<String> reducersAddresses;  
  
    @Getter  
    private CountDownLatch finishedMappersLatch;  
    @Getter  
    private CountDownLatch waitForContainersLatch;  
}
```

ITEM 17: not applicable.

ITEM 18: FAVOR COMPOSITION OVER INHERITANCE.

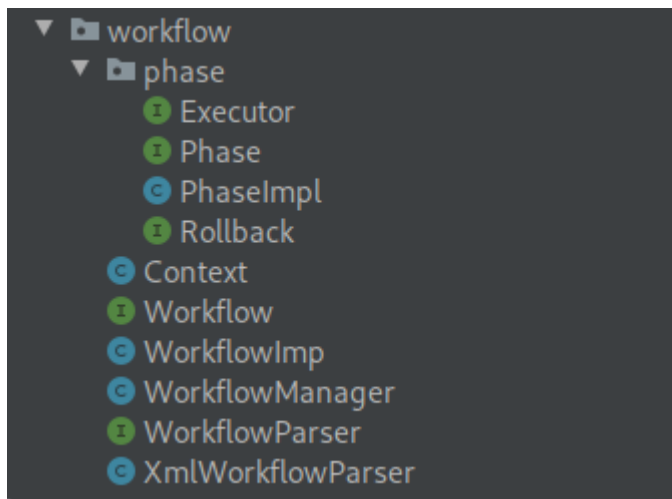
Favoured by using strategy pattern.



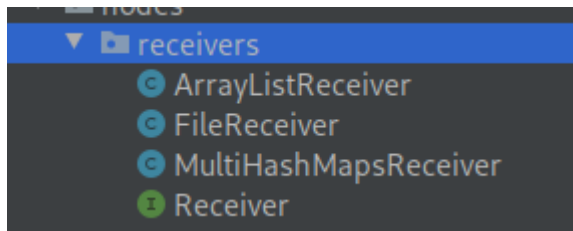
ITEM 19:

Not applicable.

ITEM 20: PREFER INTERFACES TO ABSTRACT CLASSES.



ITEM 21: DESIGN INTERFACES FOR POSTERITY.



ITEM22: USE INTERFACES ONLY TO DEFINE TYPE.

No interfaces used to define constants. All interfaces define types.

ITEM23:

Not applicable.

ITEM24: FAVOR STATIC MEMBER CLASSES OVER NONSTATIC.

Not applicable.

ITEM25: LIMIT SOURCE FILE TO A SINGLE TOP-LEVEL CLASS

All files have only one top level class.

ITEM26: DON'T USE RAW TYPES.

No raw types used.

```

private int numOfHashMapsToReceive;

private List<Map<Object, Object>> dataReceived;
private Lock lock = new ReentrantLock();

```

ITEM27: ELIMINATE UNCHECKED WARNINGS.

```

public class Context {

    private Map<String, Object> params;

    public Context(Map<String, Object> params) {
        this.params = params;
    }

    @SuppressWarnings("unchecked")
    public <T> T getParam(String paramName) {
        return (T) params.get(paramName);
    }

    public void put(String paramName, Object paramValue) {
        params.put(paramName, paramValue);
    }
}

```

ITEM28: PREFER LISTS TO ARRAYS.

Always preferred.

```

import ...

public class HashShuffler implements Shuffler {

    int numNodes;
    private List<Map<Object, Object>> shuffleResult;
    private Map<Object, Object> mapToShuffle;

    public HashShuffler(Map<?, ?> mapToShuffle, int numNodes) {

```

ITEM29+30: FAVOR GENERIC TYPES AND GENERIC METHODS.

```
}  
  
public static <T> void sendObject(String address, int port, T dataToSend) throws IOException {  
    try (Socket sk = new Socket(address, port);  
        ObjectOutputStream objectOutput = new ObjectOutputStream(sk.getOutputStream())) {  
        objectOutput.writeObject(dataToSend);  
    } catch (Exception e) {  
        throw e;  
    }  
}
```

```
import java.util.Map;  
  
public class Context {  
  
    private Map<String, Object> params;  
  
    public Context(Map<String, Object> params) {  
        this.params = params;  
    }  
  
    @SuppressWarnings("unchecked")  
    public <T> T getParam(String paramName) {  
        return (T) params.get(paramName);  
    }  
  
    public void put(String paramName, Object paramValue) {  
        params.put(paramName, paramValue);  
    }  
}
```

ITEM39: PREFER ANNOTATIONS TO NAMING PATTERNS.

Used lombok.

```
private static ContainersDataTracker containersD

@Setter
@Getter
private AtomicInteger numOfContainers;
@Setter
```

ITEM40: CONSISTENTLY USE THE OVERRIDE ANNOTATION.

Always used.

```
public ArrayListReceiver() {
}

@Override
/unchecked/
public ArrayList<Object> start(int port) {
    try (ServerSocket myServerSocket = new ServerSocket(port);
        Socket skt = myServerSocket.accept();
        ObjectInputStream objectInput = new ObjectInputStream(skt.getInputStream())) {
        Object object = objectInput.readObject();
        dataReceived = (ArrayList<Object>) object;
    } catch (Exception e) {
        e.printStackTrace();
    }
    return dataReceived;
}
```

ITEM42: PREFER LAMBDAS TO ANONYMOUS CLASSES.

```
new Thread(
    () -> {
        for (Workflow workflow : workflowManager.getWorkflows()) {
            workflow.start(data);
        }
    })
.start();
```

ITEM43: RETURN EMPTY COLLECTIONS OR ARRAYS, NOT NULLS.

No nulls returned.

ITEM57: MINIMIZE THE SCOPE OF LOCAL VARIABLES.

```
int i=0;
for (i = 0; i < numberOfReducers; i++) {

    try (Socket sk = new Socket(reducersAddresses.get(i), Constants.MAPPERS_TO_REDUCERS_PORT);
        ObjectOutputStream objectOutput = new ObjectOutputStream(sk.getOutputStream())) {
        objectOutput.writeObject(shuffleResult.get(i));
    } catch (Exception e) {
        System.out.println("Sending failed trying again in 3 seconds");
        Thread.sleep( millis: 3000);
        i--; //recover
    }
}
```

ITEM58: PREFER FOR-EACH LOOPS TO TRADITIONAL LOOPS.

Whenever available.

```
public void sendMappersAddressesToReducers() throws InterruptedException, IOException {
    Thread.sleep( millis: 3000);
    for (String address : containersDataTracker.getReducersAddresses()) {
        try (Socket sk = new Socket(address, Constants.MAINSERVER_TO_REDUCERS_PORT)) {
            ObjectOutputStream objectOutput = new ObjectOutputStream(sk.getOutputStream());
            objectOutput.writeObject(containersDataTracker.getMappersAddresses());
        } catch (Exception e) {
            throw e;
        }
    }
}
```

ITEM62 + 63 :

Replaced by string builder..

ITEM65: PREFER INTERFACES TO REFLECTION.

```
System.out.println("Receiving mappers addresses");  
Receiver mapperAddressesReceiver = new ArrayListReceiver();  
mappersAddresses = mapperAddressesReceiver.start(Constants.MAIN)
```

ITEM68: PREFER INTERFACES TO REFLECTION.

Google styling and clean code.

ITEM69: USE EXCEPTIONS ONLY FOR EXCEPTIONAL CONDITIONS.

There are no try abuses.

ITEM70+71: USE CHECKED FOR RECOVERABLE CONDITIONS AND RUNTIME EXCEPTIONS FOR PROGRAMMING ERRORS.

Im throwing e in most of methods. The reason for that it to handle it from the phase and throw and checked exception to rollback the phase.

ITEM78: SYNCHRONIZE ACCESS TO SHARED MUTABLE DATA.


```

@Setter
@Getter
private AtomicInteger numOfContainers;
@Setter
@Getter
private AtomicInteger numOfMappers;
@Setter
@Getter
private AtomicInteger numOfReducer;
@Setter
@Getter
private AtomicInteger runningContainers;
@Getter
private AtomicInteger currentMappersRunning;
@Getter
private AtomicInteger currentReducersRunning;
@Getter
private AtomicInteger finishedMappers;

@Getter
private ArrayList<String> mappersAddresses;
@Getter
private ArrayList<String> reducersAddresses;

@Getter
private CountDownLatch finishedMappersLatch;
@Getter
private CountDownLatch waitForContainersLatch;

private ContainerDataTracker() {

```

```

@synchronized
public void addMapperAddress(String address) {
    mappersAddresses.add(address);
}

@synchronized
public void addReducerAddress(String address) {
    reducersAddresses.add(address);
}

```

ITEM81: PREFER CONCURRENCY UTILITIES TO WAIT AND NOTIFY.

I used latches

Countdown latches are single-use barriers that allow one or more threads to wait for one or more other threads to do something. The sole constructor for CountDownLatch takes an int

that is the number of times the countDown method must be invoked on the latch before all waiting threads are allowed to proceed.

```
@Getter
private CountdownLatch finishedMappersLatch;
@Getter
private CountdownLatch waitForContainersLatch;

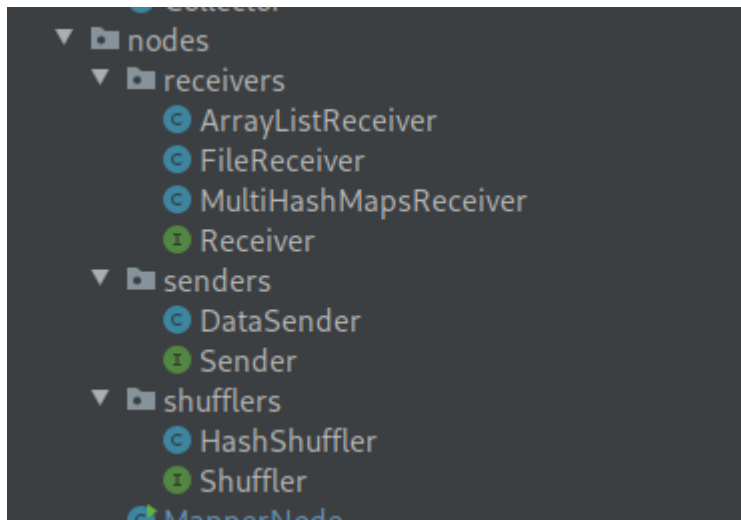
private ContainersDataTracker() {
```

3. Design Patterns:

3.1 Strategy patterns:

I have faced many method that receives multiple object's type in the nodes. Where each one has different algorithm.

However strategy pattern increased the usability of code and did eliminate duplicates and reduced coupling between objects.



Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.

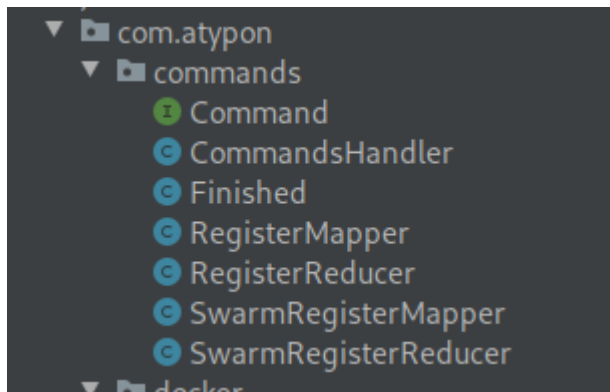
Capture the abstraction in an interface, bury implementation details in derived classes.

3.2. Command Pattern

- Encapsulate a request as an object, thereby letting us parametrize clients with different requests, queue or log requests, and support undoable operations.
- Promote "invocation of a method on an object" to full object status

Why?

Need to issue requests to objects without knowing anything about the operation being requested or the receiver of the request. Which reduce coupling and eliminates switch/if statements.



3.3. Observer Pattern (Not Implemented).

This is not implemented in the project however i wanted to stop the execution of the phases until all containers run. One way to approach this is to implement observer pattern to keep track of the number of containers running and release a lock. However i have used CountDownLatch for simplicity.

```
@Getter
private CountDownLatch finishedMappersLatch;
@Getter
private CountDownLatch waitForContainersLatch;

private ContainersDataTracker() {
```

3.4. SINGLETONS:

Application needs one, and only one, instance of data Tracker.

4. Styling Guide:

Google styling guide achieved by Google reformat plugin.

Data Structures used:

- 1- HashMaps for it's faster put/get access $O(1)$ + as a return type for users given codes.
- 2- ArrayLists -> preferred on arrays (effective java) and it is serializable + it has dynamic size.
- 4- ConcurrentSkipListMap -> ThreadSafe sorted version of Map.

5. DevOps:

Used docker to run mappers / reducers nodes. Why?

Docker provide easier Scalability. And insulation ! however the user code could be malicious so docker prevent it from running on the host machine.

Docker Swarm on docker-machine to simulate running the project at scale on multiple clusters.