```java
package stockProvider;

import java.util.Iterator;
import java.util.UUID;
import java.util.ArrayList;
import java.lang.Math;
import java.lang.System;
import java.util.Random;
import java.util.concurrent.TimeoutException;
import java.io.IOException;
import java.util.Random;

import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.Channel;
import org.apache.commons.lang3.SerializationUtils;

import stockManager.StockDB;
import common.Product;
import configParser.ConfigParser;
import logger.Logger;
import logger.LogLevel;

public class MainClass {
    public MainClass(String[] argv) throws IllegalArgumentException,
                                           IOException {
        config_ = ConfigParser.getInstance();
        logger_ = Logger.getInstance();

        config_.init(argv[1]);
        this.initLogger(argv[0]);

        String stockDBFile = config_.get("STOCK", "stock-db-file");
        stockDB_ = new StockDB(stockDBFile);
    }

    public static void main(String[] argv) {
        ConfigParser config = ConfigParser.getInstance();
        Logger logger = Logger.getInstance();
        try {
            MainClass app = new MainClass(argv);
            app.increaseStock();
        }
        catch (IllegalArgumentException e) {
            // We couldn't open the logger. Just exit
            System.out.println(e);
            System.exit(-1);
        }
        catch (IOException e) {
             logger.log(LogLevel.ERROR, e.toString());
        }
    }

    private void initLogger(String processNumber)
    throws IllegalArgumentException {
        String logFileName = config_.get("MAIN", "log-file");
        String logLevel = config_.get("MAIN", "log-level");

        Logger logger = Logger.getInstance();
        logger.init(logFileName, LogLevel.parse(logLevel));
        logger.setPrefix("[STOCK_PROVIDER " + processNumber + "]");
        logger.log(LogLevel.DEBUG, "Process started");
    }

    public void increaseStock() throws IllegalArgumentException,
                                       IOException {
        long globalIncrease = Long.parseLong(config_.get("STOCK-PROVIDER",
                                                         "global-increase"));
        long type1Increase = Long.parseLong(config_.get("STOCK-PROVIDER",
                                                        "type-1-increase"));
        long type2Increase = Long.parseLong(config_.get("STOCK-PROVIDER",
                                                        "type-2-increase"));
        long type3Increase = Long.parseLong(config_.get("STOCK-PROVIDER",
```

```java
                                                        "type-3-increase"));
        long type4Increase = Long.parseLong(config_.get("STOCK-PROVIDER",
                                                        "type-4-increase"));
        long type5Increase = Long.parseLong(config_.get("STOCK-PROVIDER",
                                                        "type-5-increase"));

        stockDB_.increaseStock(Product.TYPE_1, globalIncrease + type1Increase);
        stockDB_.increaseStock(Product.TYPE_2, globalIncrease + type2Increase);
        stockDB_.increaseStock(Product.TYPE_3, globalIncrease + type3Increase);
        stockDB_.increaseStock(Product.TYPE_4, globalIncrease + type4Increase);
        stockDB_.increaseStock(Product.TYPE_5, globalIncrease + type5Increase);
    }

    private Logger logger_;
    private ConfigParser config_;
    private StockDB stockDB_;
}
```

```java
package stockManager;

import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Consumer;
import com.rabbitmq.client.DefaultConsumer;
import com.rabbitmq.client.Envelope;
import com.rabbitmq.client.AMQP;
import org.apache.commons.lang3.SerializationUtils;

import configParser.ConfigParser;
import common.Order;
import common.OrderState;
import logger.Logger;
import logger.LogLevel;
import stockManager.StockDB;

import java.io.IOException;


public class StockManager extends DefaultConsumer {
    public StockManager(Channel channel) throws IllegalArgumentException,
                                                IOException {
        super(channel);
        logger_ = Logger.getInstance();
        config_ = ConfigParser.getInstance();
        this.initQueues();

        String stockDBFile = config_.get("STOCK", "stock−db−file");
        stockDB_ = new StockDB(stockDBFile);
    }

    @Override
    public void handleDelivery(String consumerTag,
                               Envelope envelope,
                               AMQP.BasicProperties properties,
                               byte[] body) throws IOException {
        Order newOrder = (Order) SerializationUtils.deserialize(body);
        logger_.log(LogLevel.TRACE, "Order received: " + newOrder.toString());


        elapsedTime_ = System.currentTimeMillis();
        boolean enoughStock = stockDB_.decreaseStock(newOrder.productType(),
                                                     newOrder.amount());
        elapsedTime_ = System.currentTimeMillis() − elapsedTime_;
        logger_.log(LogLevel.NOTICE, "decreaseStock. Time: "
            + elapsedTime_ + " ms.");

        if (enoughStock) {
            newOrder.state(OrderState.ACCEPTED);
        }
        else {
            newOrder.state(OrderState.REJECTED);
        }

        logger_.log(LogLevel.INFO, "Order processed: "
            + newOrder.toStringShort() + ". Sending it to the OrderManager.");

        body = SerializationUtils.serialize(newOrder);
        this.getChannel().basicPublish("", orderManagerQueueName_, null, body);
    }

    /**
     * @brief Declare the queues. This is necessary because maybe they have not
     * been created yet
     */
    private void initQueues() throws IOException {
        Channel channel = this.getChannel();

        orderManagerQueueName_ = config_.get("QUEUES", "order−manager−queue");
        channel.queueDeclare(orderManagerQueueName_,
                             false,
```

```java
                             false,
                             false,
                             null);
    }

    private Logger logger_;
    private ConfigParser config_;
    private String orderManagerQueueName_;
    private StockDB stockDB_;

    // For performance stats
    private long elapsedTime_;
}
```

```java
package stockManager;

import java.nio.channels.OverlappingFileLockException;
import java.lang.System;
import java.io.EOFException;
import java.nio.ByteBuffer;
import java.nio.channels.FileLock;
import java.util.Map;
import java.io.IOException;
import java.io.RandomAccessFile;
import java.nio.channels.FileChannel;
import java.io.FileOutputStream;
import java.io.File;
import java.util.HashMap;

import common.Product;
import logger.Logger;
import logger.LogLevel;

public class StockDB {
    public StockDB(String dbFilePath) throws IOException {
        logger_ = Logger.getInstance();

        File file = new File(dbFilePath);
        // Taken from the JavaDocs
        // http://docs.oracle.com/javase/7/docs/api/java/io/File.
        // html#createNewFile()
        // Atomically creates a new, empty file named by this abstract
        // pathname if and only if a file with this name does not yet
        // exist. The check for the existence of the file and the creation
        // of the file if it does not exist are a single operation that is
        // atomic with respect to all other filesystem activities that
        // might affect the file.
        file.createNewFile();

        // http://docs.oracle.com/javase/7/docs/api/java/io/
        // RandomAccessFile.html#mode
        // The "rwd" mode can be used to reduce the number of I/O operations
        // performed. Using "rwd" only requires updates to the file's content
        // to be written to storage; using "rws" requires updates to both the
        // file's content and its metadata to be written, which generally
        // requires at least one more low-level I/O operation.
        file_ = new RandomAccessFile(dbFilePath, "rwd");
        FileLock lock = file_.getChannel().lock();

        if (file.length() ≡ 0) {
            this.createEmptyStockFile();
            logger_.log(LogLevel.WARNING,
                "StockDB file doesn't exists."
                + " Proceed to create it. StockDB file: " + dbFilePath);
        }
        lock.release();
    }

    private void createEmptyStockFile() throws IOException {
        HashMap<Product, Long> map = new HashMap<Product, Long>();
        for (Product product : Product.values()) {
            map.put(product, new Long(10000));
        }

        for (Map.Entry<Product, Long> entry : map.entrySet()) {
            String key = String.format("%-10s", entry.getKey().toString());
            file_.write(key.getBytes());

            ByteBuffer b = ByteBuffer.allocate(8);
            b.putLong(entry.getValue());
            file_.write(b.array());
        }
    }

    // This method is very long, but don't split it in functions because
    // to performance problems
    public boolean decreaseStock(Product product, Long amount)
```

```java
    throws IOException {
        byte[] buffer = new byte[PRODUCT_KEY_MAX_SIZE];
        file_.seek(0);

        try {
            while(true) {
                int readBytes = file_.read(buffer, 0, PRODUCT_KEY_MAX_SIZE);
                if (readBytes ≡ -1) {
                    // lock.release();
                    return false;
                }

                Product key = Product.valueOf(new String(buffer).trim());
                if (key ≠ product) {
                    // Jump to the next entry
                    file_.skipBytes(Long.BYTES);
                    continue;
                }

                // Product found, proceed to update value
                // Read the amount of the stock to update it, and go back to
                // the same position
                file_.read(buffer, 0, Long.BYTES);
                file_.seek(file_.getFilePointer() - Long.BYTES);

                // Check if there is stock of the file
                ByteBuffer b = ByteBuffer.wrap(buffer);
                long productStock = b.getLong();
                if (productStock < amount) {
                    logger_.log(LogLevel.WARNING, "Order cannot be accepted. "
                        + "Not enough stock of product " + product.toString()
                        + ". ProductStock: " + productStock
                        + " - OrderAmount: " + amount);
                    return false;
                }

                // There is stock, update the StockDB
                long newStock = productStock - amount;
                ByteBuffer longBuf = ByteBuffer.allocate(Long.BYTES);
                longBuf.putLong(newStock);

                // Just lock the part of the file to me modified
                FileLock lock = file_.getChannel().lock(file_.getFilePointer(),
                                                        Long.BYTES,
                                                        false);
                file_.write(longBuf.array());
                lock.release();

                logger_.log(LogLevel.DEBUG, "Decreasing stock of product "
                    + product.toString() + ". PreviousStock: "
                    + productStock + " - UpdatedStock: " + newStock);
                break;
            }
        }
        catch (EOFException e) {
            // If this happen, then the product does not exists and we have
            // a bug in the system. ABORT!
            logger_.log(LogLevel.ERROR, "Product does not exists. Product: "
                + product.toString());
            System.exit(-1);
        }

        // lock.release();
        return true;
    }


    // This method is very long, but don't split it in functions because
    // to performance problems
    public boolean increaseStock(Product product, Long amount)
    throws IOException {
        byte[] buffer = new byte[PRODUCT_KEY_MAX_SIZE];
        file_.seek(0);

        try {
            while(true) {
                int readBytes = file_.read(buffer, 0, PRODUCT_KEY_MAX_SIZE);
                if (readBytes == -1) {
                    // lock.release();
                    return false;
                }

                Product key = Product.valueOf(new String(buffer).trim());
                if (key != product) {
                    // Jump to the next entry
                    file_.skipBytes(Long.BYTES);
                    continue;
                }

                // Product found, proceed to update value
                // Read the amount of the stock to update it, and go back to
                // the same position
                file_.read(buffer, 0, Long.BYTES);
                file_.seek(file_.getFilePointer() - Long.BYTES);

                // Check if there is stock of the file
                ByteBuffer b = ByteBuffer.wrap(buffer);
                long productStock = b.getLong();

                // There is stock, update the StockDB
                long newStock = productStock + amount;
                ByteBuffer longBuf = ByteBuffer.allocate(Long.BYTES);
                longBuf.putLong(newStock);

                // Just lock the part of the file to me modified
                FileLock lock = file_.getChannel().lock(file_.getFilePointer(),
                                                        Long.BYTES,
                                                        false);
                file_.write(longBuf.array());
                lock.release();

                logger_.log(LogLevel.NOTICE, "Increasing stock of product "
                    + product.toString() + ". PreviousStock: "
                    + productStock + " - UpdatedStock: " + newStock);
                break;
            }
        }
        catch (EOFException e) {
            // If this happen, then the product does not exists and we have
            // a bug in the system. ABORT!
            logger_.log(LogLevel.ERROR, "Product does not exists. Product: "
                + product.toString());
            System.exit(-1);
        }

        // lock.release();
        return true;
    }

    private Logger logger_;
    private RandomAccessFile file_;
    private static final int PRODUCT_KEY_MAX_SIZE = 10;
}
```

```java
package stockManager;

// Program includes
import configParser.ConfigParser;
import logger.Logger;
import logger.LogLevel;

// External libraries includes
import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Consumer;
import com.rabbitmq.client.DefaultConsumer;
import com.rabbitmq.client.Envelope;
import com.rabbitmq.client.AMQP;

// Java includes
import java.lang.IllegalArgumentException;
import java.io.IOException;
import java.util.concurrent.TimeoutException;

public class MainClass {
    public static void main(String[] argv) {
        ConfigParser config = ConfigParser.getInstance();

        Logger logger = Logger.getInstance();

        try {
            MainClass app = new MainClass();
            config.init(argv[1]);
            app.initLogger(config, argv[0]);

            ConnectionFactory factory = new ConnectionFactory();
            factory.setHost(config.get("MAIN", "server-address", "localhost"));
            Connection connection = factory.newConnection();
            Channel channel = connection.createChannel();

            String stockQueue = config.get("QUEUES", "stock-manager-queue");
            // To secure fairness between the processes
            channel.basicQos(1);
            channel.queueDeclare(stockQueue,
                                 false,
                                 false,
                                 false,
                                 null);

            Consumer consumer = new StockManager(channel);
            channel.basicConsume(stockQueue, true, consumer);
        }
        catch (IllegalArgumentException e) {
            // We couldn't open the logger. Just exit
            System.out.println(e);
            System.exit(-1);
        }
        catch (TimeoutException e) {
            logger.log(LogLevel.ERROR, e.toString());
        }
        catch (IOException e) {
            logger.log(LogLevel.ERROR, e.toString());
        }
    }

    private void initLogger(ConfigParser config, String processNumber)
    throws IllegalArgumentException {
        String logFileName = config.get("MAIN", "log-file");
        String logLevel = config.get("MAIN", "log-level");

        Logger logger = Logger.getInstance();
        logger.init(logFileName, LogLevel.parse(logLevel));
        logger.setPrefix("[STOCK_MANAGER " + processNumber + "]");
        logger.log(LogLevel.DEBUG, "Process started");
    }
}
```

```
package requestDispatcher;

import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Consumer;
import com.rabbitmq.client.DefaultConsumer;
import com.rabbitmq.client.Envelope;
import com.rabbitmq.client.AMQP;
import org.apache.commons.lang3.SerializationUtils;

import configParser.ConfigParser;
import common.Order;
import common.OrderState;
import logger.Logger;
import logger.LogLevel;

import java.io.IOException;


public class RequestDispatcher extends DefaultConsumer {
    public RequestDispatcher(Channel channel) throws IllegalArgumentException,
                                                     IOException {
        super(channel);
        logger_ = Logger.getInstance();
        config_ = ConfigParser.getInstance();
        this.initQueues();
    }

    @Override
    public void handleDelivery(String consumerTag,
                               Envelope envelope,
                               AMQP.BasicProperties properties,
                               byte[] body) throws IOException {
        Order newOrder = (Order) SerializationUtils.deserialize(body);

        newOrder.state(OrderState.RECEIVED);
        body = SerializationUtils.serialize(newOrder);

        this.getChannel().basicPublish("", orderManagerQueueName_, null, body);
        logger_.log(LogLevel.DEBUG, "Order received: " + newOrder.stringID());

        // Send the order received to the auditLog
        this.getChannel().basicPublish("", auditLogQueueName_, null, body);

        // Send the order to the Stock Manager
        this.getChannel().basicPublish("", stockManagerQueueName_, null, body);
    }

    /**
     * @brief Declare the queues. This is necessary because maybe they have not
     * been created yet
     */
    private void initQueues() throws IOException {
        Channel channel = this.getChannel();
        channel.basicQos(1);

        auditLogQueueName_ = config_.get("QUEUES", "audit-log-queue");
        channel.queueDeclare(auditLogQueueName_,
                             false,
                             false,
                             false,
                             null);

        orderManagerQueueName_ = config_.get("QUEUES", "order-manager-queue");
        channel.queueDeclare(orderManagerQueueName_,
                             false,
                             false,
                             false,
                             null);

        stockManagerQueueName_ = config_.get("QUEUES", "stock-manager-queue");
        channel.queueDeclare(stockManagerQueueName_,
```

```
                             false,
                             false,
                             false,
                             null);
    }

    private Logger logger_;
    private ConfigParser config_;
    private String auditLogQueueName_;
    private String stockManagerQueueName_;
    private String orderManagerQueueName_;
}
```

```java
package requestDispatcher;

// Program includes
import configParser.ConfigParser;
import logger.Logger;
import logger.LogLevel;
import requestDispatcher.RequestDispatcher;

// External libraries includes
import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Consumer;
import com.rabbitmq.client.DefaultConsumer;
import com.rabbitmq.client.Envelope;
import com.rabbitmq.client.AMQP;

// Java includes
import java.lang.IllegalArgumentException;
import java.io.IOException;
import java.util.concurrent.TimeoutException;

public class MainClass {
    public static void main(String[] argv) {
        ConfigParser config = ConfigParser.getInstance();

        Logger logger = Logger.getInstance();

        try {
            MainClass app = new MainClass();
            config.init(argv[1]);
            app.initLogger(config, argv[0]);

            ConnectionFactory factory = new ConnectionFactory();
            factory.setHost(config.get("MAIN", "server−address", "localhost"));
            Connection connection = factory.newConnection();
            Channel channel = connection.createChannel();

            String clientQueue = config.get("QUEUES", "client−queue");
            // To secure fairness between the processes
            channel.basicQos(1);
            channel.queueDeclare(clientQueue,
                                 false,
                                 false,
                                 false,
                                 null);

            Consumer consumer = new RequestDispatcher(channel);
            channel.basicConsume(clientQueue, true, consumer);
        }
        catch (IllegalArgumentException e) {
            // We couldn’t open the logger. Just exit
            System.out.println(e);
            System.exit(−1);
        }
        catch (TimeoutException e) {
            logger.log(LogLevel.ERROR, e.toString());
        }
        catch (IOException e) {
            logger.log(LogLevel.ERROR, e.toString());
        }
    }

    private void initLogger(ConfigParser config, String processNumber)
    throws IllegalArgumentException {
        String logFileName = config.get("MAIN", "log−file");
        String logLevel = config.get("MAIN", "log−level");

        Logger logger = Logger.getInstance();
        logger.init(logFileName, LogLevel.parse(logLevel));
        logger.setPrefix("[REQUEST_DISPATCHER " + processNumber + "]");
        logger.log(LogLevel.DEBUG, "Process started");
    }
```

```java
}
```

```java
package querySolver;

import java.util.UUID;
import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Consumer;
import com.rabbitmq.client.DefaultConsumer;
import com.rabbitmq.client.Envelope;
import com.rabbitmq.client.AMQP;
import org.apache.commons.lang3.SerializationUtils;

import configParser.ConfigParser;
import common.Order;
import common.OrderState;
import common.OrderDB;
import logger.Logger;
import logger.LogLevel;

import java.io.IOException;


public class QuerySolver extends DefaultConsumer {
    public QuerySolver(Channel channel) throws IllegalArgumentException,
                                               IOException {
        super(channel);
        logger_ = Logger.getInstance();
        config_ = ConfigParser.getInstance();
        orderDB_ = new OrderDB(config_.get("ORDER", "order−db−directory"));
    }

    @Override
    public void handleDelivery(String consumerTag,
                               Envelope envelope,
                               AMQP.BasicProperties properties,
                               byte[] body) throws IOException {
        UUID orderKey = (UUID) SerializationUtils.deserialize(body);
        logger_.log(LogLevel.DEBUG, "Query received. Key: "
            + orderKey.toString());

        Order order = orderDB_.get(orderKey);
        if (order ≠ null) {
            logger_.log(LogLevel.NOTICE, "Order " + orderKey.toString()
                + " − State: " + order.state().toString());
        }
        else {
            logger_.log(LogLevel.WARNING, "Order " + orderKey.toString()
                + " was not processed yet.");
        }
    }

    private Logger logger_;
    private ConfigParser config_;
    private OrderDB orderDB_;
}
```

```java
package querySolver;

// Program includes
import configParser.ConfigParser;
import logger.Logger;
import logger.LogLevel;
import querySolver.QuerySolver;

// External libraries includes
import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Consumer;
import com.rabbitmq.client.DefaultConsumer;
import com.rabbitmq.client.Envelope;
import com.rabbitmq.client.AMQP;

// Java includes
import java.lang.IllegalArgumentException;
import java.io.IOException;
import java.util.concurrent.TimeoutException;

public class MainClass {
    public static void main(String[] argv) {
        ConfigParser config = ConfigParser.getInstance();

        Logger logger = Logger.getInstance();

        try {
            MainClass app = new MainClass();
            config.init(argv[1]);
            app.initLogger(config, argv[0]);

            ConnectionFactory factory = new ConnectionFactory();
            factory.setHost(config.get("MAIN", "server−address", "localhost"));
            Connection connection = factory.newConnection();
            Channel channel = connection.createChannel();

            String queryQueue = config.get("QUEUES", "query−queue");
            // To secure fairness between the processes
            channel.basicQos(1);
            channel.queueDeclare(queryQueue,
                                 false,
                                 false,
                                 false,
                                 null);

            Consumer consumer = new QuerySolver(channel);
            channel.basicConsume(queryQueue, true, consumer);
        }
        catch (IllegalArgumentException e) {
            // We couldn’t open the logger. Just exit
            System.out.println(e);
            System.exit(−1);
        }
        catch (TimeoutException e) {
            logger.log(LogLevel.ERROR, e.toString());
        }
        catch (IOException e) {
            logger.log(LogLevel.ERROR, e.toString());
        }
    }

    private void initLogger(ConfigParser config, String processNumber)
    throws IllegalArgumentException {
        String logFileName = config.get("MAIN", "log−file");
        String logLevel = config.get("MAIN", "log−level");

        Logger logger = Logger.getInstance();
        logger.init(logFileName, LogLevel.parse(logLevel));
        logger.setPrefix("[QUERY_SOLVER " + processNumber + "]");
        logger.log(LogLevel.DEBUG, "Process started");
    }
}
```

```java
package orderManager;

import java.lang.System;
import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Consumer;
import com.rabbitmq.client.DefaultConsumer;
import com.rabbitmq.client.Envelope;
import com.rabbitmq.client.AMQP;
import org.apache.commons.lang3.SerializationUtils;

import configParser.ConfigParser;
import common.Order;
import common.OrderDB;
import common.OrderState;
import logger.Logger;
import logger.LogLevel;

import java.io.IOException;


public class OrderManager extends DefaultConsumer {
    public OrderManager(Channel channel) throws IllegalArgumentException,
                                                IOException {
        super(channel);
        logger_ = Logger.getInstance();
        config_ = ConfigParser.getInstance();

        orderDB_ = new OrderDB(config_.get("ORDER", "order−db−directory"));
        this.initQueues();

    }

    @Override
    public void handleDelivery(String consumerTag,
                               Envelope envelope,
                               AMQP.BasicProperties properties,
                               byte[] body) throws IOException {
        Order newOrder = (Order) SerializationUtils.deserialize(body);
        logger_.log(LogLevel.DEBUG, "Order received: " + newOrder.stringID());

        OrderState state = newOrder.state();
        elapsedTime_ = System.currentTimeMillis();

        switch(newOrder.state()) {
            case RECEIVED:
                // Add the order to the DB
                orderDB_.add(newOrder);
                elapsedTime_ = System.currentTimeMillis() − elapsedTime_;
                logger_.log(LogLevel.NOTICE, "OrderDB::add. Time: "
                    + elapsedTime_ + " ms.");
                break;
            case DELIVERED:
            case REJECTED:
                orderDB_.alter(newOrder);
                elapsedTime_ = System.currentTimeMillis() − elapsedTime_;
                logger_.log(LogLevel.NOTICE, "OrderDB::alter. Time: "
                    + elapsedTime_ + " ms.");
                break;
            case ACCEPTED:
                orderDB_.alter(newOrder);
                elapsedTime_ = System.currentTimeMillis() − elapsedTime_;
                logger_.log(LogLevel.NOTICE, "OrderDB::alter. Time: "
                    + elapsedTime_ + " ms.");
                this.getChannel().basicPublish("",
                                               deliveryQueueName_,
                                               null,
                                               body);
                break;
        }

        logger_.log(LogLevel.INFO, "Order processed: " + newOrder.stringID());
    }

    /**
     * @brief Declare the queues. This is necessary because maybe they have not
     * been created yet
     */
    private void initQueues() throws IOException {
        Channel channel = this.getChannel();
        channel.basicQos(1);

        deliveryQueueName_ = config_.get("QUEUES", "delivery−queue");
        channel.queueDeclare(deliveryQueueName_,
                             false,
                             false,
                             false,
                             null);
    }

    private Logger logger_;
    private ConfigParser config_;
    private OrderDB orderDB_;
    private long elapsedTime_;
    private String deliveryQueueName_;
}
```

```
package orderManager;

// Program includes
import configParser.ConfigParser;
import logger.Logger;
import logger.LogLevel;
import orderManager.OrderManager;

// External libraries includes
import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Consumer;
import com.rabbitmq.client.DefaultConsumer;
import com.rabbitmq.client.Envelope;
import com.rabbitmq.client.AMQP;

// Java includes
import java.lang.IllegalArgumentException;
import java.io.IOException;
import java.util.concurrent.TimeoutException;

public class MainClass {
    public static void main(String[] argv) {
        ConfigParser config = ConfigParser.getInstance();

        Logger logger = Logger.getInstance();

        try {
            MainClass app = new MainClass();
            config.init(argv[1]);
            app.initLogger(config, argv[0]);

            ConnectionFactory factory = new ConnectionFactory();
            factory.setHost(config.get("MAIN", "server−address", "localhost"));
            Connection connection = factory.newConnection();
            Channel channel = connection.createChannel();

            String orderQueue = config.get("QUEUES", "order−manager−queue");
            // To secure fairness between the processes
            channel.basicQos(1);
            channel.queueDeclare(orderQueue,
                                 false,
                                 false,
                                 false,
                                 null);

            Consumer consumer = new OrderManager(channel);
            channel.basicConsume(orderQueue, true, consumer);
        }
        catch(SecurityException e) {
            logger.log(LogLevel.ERROR, "Cannot create OrderDB Directory. "
                + "Change folder permissions or point out to another path.");
        }
        catch (IllegalArgumentException e) {
            // We couldn’t open the logger. Just exit
            System.out.println(e);
            System.exit(−1);
        }
        catch (TimeoutException e) {
            logger.log(LogLevel.ERROR, e.toString());
        }
        catch (IOException e) {
            logger.log(LogLevel.ERROR, e.toString());
        }
    }

    private void initLogger(ConfigParser config, String processNumber)
    throws IllegalArgumentException {
        String logFileName = config.get("MAIN", "log−file");
        String logLevel = config.get("MAIN", "log−level");

        Logger logger = Logger.getInstance();
```

```java
        logger.init(logFileName, LogLevel.parse(logLevel));
        logger.setPrefix("[ORDER_MANAGER " + processNumber + "]");
        logger.log(LogLevel.DEBUG, "Process started");
    }
}
```

```java
package logger;

public enum LogLevel {
    ERROR(1),
    CRITIC(2),
    WARNING(3),
    NOTICE(4),
    INFO(5),
    DEBUG(6),
    TRACE(7);

    LogLevel(int level) {
        if (this.isLevelValid(level)) {
            this.level_ = level;
        }
        else {
            throw new IllegalArgumentException("LogLevel is out of bounds.");
        }
    }

    public static LogLevel parse(String level) {
        String lowerLogLevel = level.toLowerCase().trim();

        if (lowerLogLevel.equals("error")) {
            return LogLevel.ERROR;
        }
        else if (lowerLogLevel.equals("critic")) {
            return LogLevel.CRITIC;
        }
        else if (lowerLogLevel.equals("warning")) {
            return LogLevel.WARNING;
        }
        else if (lowerLogLevel.equals("info")) {
            return LogLevel.INFO;
        }
        else if (lowerLogLevel.equals("notice")) {
            return LogLevel.NOTICE;
        }
        else if (lowerLogLevel.equals("debug")) {
            return LogLevel.DEBUG;
        }
        else if (lowerLogLevel.equals("trace")) {
            return LogLevel.TRACE;
        }

        throw new IllegalArgumentException("Invalid LogLevel introduced: "
            + level);
    }

    public int level() {
        return level_;
    }

    public Boolean isLevelValid(int level) {
        // TODO: This enum are shit. I try, but this is the best a can do
        return level ≥ 1 ∧ level ≤ 7;
        /*return level >= LogLevel.ERROR &&
                level <= LogLevel.TRACE;*/
    }

    public String prefix(LogLevel verbosity) {
        int level = verbosity.level();

        if (this.isLevelValid(level)) {
            return PREFIX_ARRAY[level − 1];
        }

        throw new IllegalArgumentException("LogLevel is out of bounds.");
    }

    private final int level_;
    private final String PREFIX_ARRAY[] = {"[ERROR]",
                                           "[CRITIC]",
                                                            "[WARNING]",
                                                            "[NOTICE]",
                                                            "[INFO]",
                                                            "[DEBUG]",
                                                            "[TRACE]"};
}
```

```java
package logger;

import java.io.*;
import java.nio.channels.FileLock;
import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
import java.util.Date;

import logger.LogLevel;



public class Logger {
    private Logger() {}

    public static Logger getInstance() {
        if (logger_ ≡ null) {
            logger_ = new Logger();
        }

        return logger_;
    }

    public void init(String filePath, LogLevel verbosity) {
        verbosity_ = verbosity;
        dateFormat_ = new SimpleDateFormat("yyyy/MM/dd HH:mm:ss");
        lock_ = new ReentrantLock();

        try {
            // Open file in append mode
            fstream_ = new FileOutputStream(filePath, true);
        }
        catch(IOException e) {
            System.err.println("[LOGGER] Error calling init() method.");
            System.err.println(e);
        }
    }

    public void setPrefix(String prefix) {
        try {
            lock_.lock();
            prefix_ = prefix;
        }
        finally {
            lock_.unlock();
        }
    }

    public void terminate() throws IOException {
        FileLock lock = null;
        try {
            lock = fstream_.getChannel().lock();
            fstream_.close();
        }
        catch(IOException e) {
            System.err.println("[LOGGER] Error calling terminate() method.");
            System.err.println(e);
            System.exit(−1);
        }
        finally {
            lock.release();
        }
    }

    public void log(LogLevel verbosity, String msg) {
        if (verbosity_.level() ≥ verbosity.level()) {
            this.write(verbosity_.prefix(verbosity) + " " + msg);
        }
    }

    private void write(String msg) {
        try {
            Date date = new Date();
            msg = dateFormat_.format(date) + " " + prefix_ + " " + msg + "\n";

            FileLock lock = fstream_.getChannel().lock();
            fstream_.write(msg.getBytes());
            fstream_.flush();
            // Remove sync to enhace performance
            // fstream_.getFD().sync();
            lock.release();
        }
        catch(IOException e) {
            System.err.println("[LOGGER] Error calling write() method.");
            System.err.println(e);
            System.exit(−1);
        }
    }

    private static Logger logger_ = null;
    private DateFormat dateFormat_;
    private LogLevel verbosity_;
    private FileOutputStream fstream_;
    private Lock lock_;
    private String prefix_;
}
```

```
import subprocess
import sys
import os
import ConfigParser


class Launcher(object):
    def __init__(self):
        self._config = ConfigParser.RawConfigParser()
        self._config.read('launcher.ini')
        self._absolute_path = self._config.get("MAIN", "absolute−path")
        self._processes_pid_list = []
        self._manual_processes = {}
        self._processes_config_file = self._config.get("MAIN",
                                                       "processes−config−file")
        self._common_classpath = ""
        self._common_classpath = self.process_classpath("MAIN")

    def process_classpath(self, section):
        # Process the classpath
        classpath = self._config.get(section, "classpath")

        ret_classpath = ""
        for lib in classpath.split(":"):
            ret_classpath += self._absolute_path + lib + ":"
        return self._common_classpath + ret_classpath

    def init_system_processes(self):
        for section in self._config.sections():
            if section ≡ "MAIN":
                continue

            if ¬ self._config.getboolean(section, "run"):
                continue

            classpath = self.process_classpath(section)
            classname = self._config.get(section, "class−name")
            amount_processes = self._config.getint(section, "amount")

            for i in range(1, amount_processes + 1):
                print "Proceed to execute instance with ID {0} of program {1}"\
                    .format(str(i), classname.split(".")[0])

                call_args = []
                call_args.extend(["java",
                                  "−cp",
                                  classpath[:−1],
                                  classname,
                                  str(i),
                                  self._processes_config_file])
                process = subprocess.Popen(call_args, shell=False)
                if self._config.getboolean(section, "kill"):
                    self._processes_pid_list.append(process)

    def wait_for_events(self):
        # Wait for an input
        prompt = "Write 'STOP' to terminate the "\
                 "system. Write the section name "\
                 "of a process to run a instance of it.\n"

        while 1:
            user_input = raw_input(prompt)
            input_args = user_input.split(" ")

            if input_args[0] ≡ "STOP" ∨ user_input ≡ "STOP":
                if len(input_args) ≡ 3:
                    # Try to kill a specific process
                    self.kill_process(input_args[1:])
                else:
                    # Kill all scheduled processes
                    self.kill_processes()
                    break
            elif user_input in self._config.sections():
                # Process created who will terminate by their own
                self.run_process(user_input)
            elif len(input_args) > 1 ∧ input_args[0]\
            in self._config.sections():
                self.run_process(input_args[0], input_args[1])


    def run_process(self, section, key=None):
        classpath = self.process_classpath(section)
        classname = self._config.get(section, "class−name")
        print "Proceed to run program " + classname

        call_args = []
        call_args.extend(["java",
                          "−cp",
                          classpath[:−1],
                          classname,
                          "X" if key ≡ None else key,
                          self._processes_config_file])

        process = subprocess.Popen(call_args, shell=False)
        if key ≠ None:
            self._manual_processes[section, key] = process

    def kill_processes(self):
        for process in self._processes_pid_list:
            print "Killing process with PID " + str(process.pid)
            os.system("kill −15 " + str(process.pid))

    def kill_process(self, key):
        key = tuple(key)
        if self._manual_processes[key] ≠ None:
            print "Killing process " + key[0] + " with PID " +\
            str(self._manual_processes[key].pid)
            os.system("kill −15 " + str(self._manual_processes[key].pid))
            del self._manual_processes[key]
        else:
            print "Process was not registered: " + key


def main():
    launcher = Launcher()
    launcher.init_system_processes()
    launcher.wait_for_events()


if __name__ ≡ '__main__':
    main()
```

```
package employer;

// Program includes
import java.lang.Thread;
import java.lang.Runtime;
import configParser.ConfigParser;
import logger.Logger;
import logger.LogLevel;
import requestDispatcher.RequestDispatcher;

// External libraries includes
import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Consumer;
import com.rabbitmq.client.DefaultConsumer;
import com.rabbitmq.client.Envelope;
import com.rabbitmq.client.AMQP;

// Java includes
import java.lang.IllegalArgumentException;
import java.io.IOException;
import java.util.concurrent.TimeoutException;

public class MainClass extends Thread {
    public MainClass() {
        logger_ = Logger.getInstance();
    }

    public static void main(String[] argv) {
        ConfigParser config = ConfigParser.getInstance();

        Logger logger = Logger.getInstance();

        try {
            MainClass app = new MainClass();
            Runtime.getRuntime().addShutdownHook(app);

            config.init(argv[1]);
            app.initLogger(config, argv[0]);

            ConnectionFactory factory = new ConnectionFactory();
            factory.setHost(config.get("MAIN", "server−address", "localhost"));
            Connection connection = factory.newConnection();
            Channel channel = connection.createChannel();

            String deliveryQueue = config.get("QUEUES", "delivery−queue");
            // To secure fairness between the processes
            channel.basicQos(1);
            channel.queueDeclare(deliveryQueue,
                                 false,
                                 false,
                                 false,
                                 null);

            Consumer consumer = new Employer(channel, deliveryQueue);
            channel.basicConsume(deliveryQueue, true, consumer);
        }
        catch (IllegalArgumentException e) {
            // We couldn't open the logger. Just exit
            System.out.println(e);
            System.exit(−1);
        }
        catch (TimeoutException e) {
            logger.log(LogLevel.ERROR, e.toString());
        }
        catch (IOException e) {
            logger.log(LogLevel.ERROR, e.toString());
        }
    }

    private void initLogger(ConfigParser config, String processNumber)
    throws IllegalArgumentException {
        String logFileName = config.get("MAIN", "log−file");
        String logLevel = config.get("MAIN", "log−level");

        Logger logger = Logger.getInstance();
        logger.init(logFileName, LogLevel.parse(logLevel));
        logger.setPrefix("[EMPLOYER " + processNumber + "]");
        logger.log(LogLevel.DEBUG, "Process started");
    }

    public void run() {
        logger_.log(LogLevel.NOTICE, "Program finished by signal.");
    }

    private Logger logger_;
}
```

```
package employer;

import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Consumer;
import com.rabbitmq.client.DefaultConsumer;
import com.rabbitmq.client.Envelope;
import com.rabbitmq.client.AMQP;
import org.apache.commons.lang3.SerializationUtils;

import configParser.ConfigParser;
import common.Order;
import common.OrderState;
import logger.Logger;
import logger.LogLevel;

import java.io.IOException;
import java.util.concurrent.TimeoutException;

public class Employer extends DefaultConsumer {
    public Employer(Channel channel,
                    String channelName) throws IllegalArgumentException,
                                               IOException {
        super(channel);
        logger_ = Logger.getInstance();
        config_ = ConfigParser.getInstance();
        this.initQueues();

        channelName_ = channelName;
        channelClosed_ = false;

        amountOrdersToProcess_ =
            Integer.parseInt(config_.get("EMPLOYER",
                                         "amount−orders−to−process"));
    }

    @Override
    public void handleDelivery(String consumerTag,
                               Envelope envelope,
                               AMQP.BasicProperties properties,
                               byte[] body) throws IOException {
        if (channelClosed_) {
            return;
        }

        Order newOrder = (Order) SerializationUtils.deserialize(body);
        // Change the state a we should have process the order and be ready
        // to deliver it
        newOrder.state(OrderState.DELIVERED);
        body = SerializationUtils.serialize(newOrder);

        logger_.log(LogLevel.DEBUG, "Order delivered: " + newOrder.stringID());
        this.getChannel().basicPublish("", orderManagerQueueName_, null, body);
    }

    /**
     * @brief Declare the queues. This is necessary because maybe they have not
     * been created yet
     */
    private void initQueues() throws IOException {
        Channel channel = this.getChannel();
        channel.basicQos(1);

        orderManagerQueueName_ = config_.get("QUEUES", "order−manager−queue");
        channel.queueDeclare(orderManagerQueueName_,
                             false,
                             false,
                             false,
                             null);
    }

    private Logger logger_;
```

```java
    private ConfigParser config_;
    private String orderManagerQueueName_;
    private String channelName_;
    private int amountOrdersToProcess_;
    private boolean channelClosed_;
}
```

```java
package configParser;

import java.lang.IllegalArgumentException;
import java.io.FileReader;
import java.io.IOException;

// External imports
import org.ini4j.Ini;

import logger.Logger;
import logger.LogLevel;


public class ConfigParser {
    private ConfigParser() {}

    public void init(String filePath) {
        // TODO: Receive the config file from an argument
        String configFileName = filePath;
        config_ = new Ini();

        try {
            config_.load(new FileReader(configFileName));
        }
        catch(IOException e) {
            System.err.println("[CONFIGPARSER] Could not open config file.");
            System.err.println(e);
            System.exit(−1);
        }
    }

    public static ConfigParser getInstance() {
        if (configParser_ ≡ null) {
            configParser_ = new ConfigParser();
        }

        return configParser_;
    }

    public String get(String section, String key, String defaultValue) {
        String value = config_.get(section, key);
        if (value ≠ null) {
            return value;
        }

        Logger.getInstance().log(LogLevel.INFO,
            "[CONFIGPARSER] Key (" + section + "," + key
            + ") was not found. Using default value: " + defaultValue);
        return defaultValue;
    }

    public String get(String section, String key) {
        String value = config_.get(section, key);;
        if (value ≠ null) {
            return value;
        }

        String msg = "Value doesn’t exists in Config File. Section: "
            + section + " − Key: " + key;
        throw new IllegalArgumentException(msg);
    }

    private static ConfigParser configParser_ = null;
    private Ini config_;
}
```

Sep 20, 15 1:19 **Product.java** Page 1/1

```java
package common;

import java.util.Arrays;
import java.util.List;
import java.util.Collections;
import java.util.Random;

public enum Product {
    TYPE_1,
    TYPE_2,
    TYPE_3,
    TYPE_4,
    TYPE_5;

    private static final List<Product> VALUES =
        Collections.unmodifiableList(Arrays.asList(values()));
    private static final int SIZE = VALUES.size();
    private static final Random RANDOM = new Random();

    public static Product randomProduct() {
        return VALUES.get(RANDOM.nextInt(SIZE));
    }

}
```

Sep 20, 15 12:22 **OrderState.java** Page 1/1

```java
package common;

public enum OrderState {
    TO_BE_PROCESSED,
    RECEIVED,
    REJECTED,
    ACCEPTED,
    DELIVERED;
}
```

```java
package common;

import common.OrderState;
import common.Product;

import java.io.Serializable;
import java.util.UUID;


public class Order implements Serializable {
    public Order(Product productType, Long amount) {
        state_ = OrderState.TO_BE_PROCESSED;
        productType_ = productType;
        amount_ = amount;
        uuid_ = UUID.randomUUID();
    }

    public Order(UUID uuid,
                 Product productType,
                 Long amount,
                 OrderState state) {
        uuid_ = uuid;
        productType_ = productType;
        amount_ = amount;
        state_ = state;
    }

    public UUID id() {
        return uuid_;
    }

    public OrderState state() {
        return state_;
    }

    public void state(OrderState state) {
        state_ = state;
    }

    public String stringID() {
        return uuid_.toString();
    }

    public Product productType() {
        return productType_;
    }

    public Long amount() {
        return amount_;
    }

    public String toString() {
        String aux = "";
        // Reduce the size of the UUID to better log size comprehension
        aux += "Order ID: " + uuid_.toString().substring(0,6) + " − ";
        aux += "State: " + state_.toString() + " − ";
        aux += "Product Type: " + productType_.toString() + " − ";
        aux += "Amount: " + amount_;
        return aux;
    }

    /**
     * @brief Return a representation of the order just with a truncated
     * UUID and the state of the order
     */
    public String toStringShort() {
        String aux = "";
        // Reduce the size of the UUID to better log size comprehension
        aux += "Order ID: " + uuid_.toString().substring(0,6) + " − ";
        aux += "State: " + state_.toString() + " − ";
        aux += "Product Type: " + productType_.toString() + " − ";
        aux += "Amount: " + amount_;
        return aux;
```

```java
    }

    public String toStringFull() {
        String aux = "";
        // Reduce the size of the UUID to better log size comprehension
        aux += "Order ID: " + uuid_.toString() + " − ";
        aux += "State: " + state_.toString() + " − ";
        aux += "Product Type: " + productType_.toString() + " − ";
        aux += "Amount: " + amount_;
        return aux;
    }

    private OrderState state_;
    private final UUID uuid_;
    // FIXME: Create a enum or something like that to represent this
    private Product productType_;
    private Long amount_;
    public static final long serialVersionUID = 123L;
}
```

```java
package common;

import java.lang.System;
import java.io.EOFException;
import java.util.UUID;
import java.nio.ByteBuffer;
import java.nio.channels.FileLock;
import java.io.RandomAccessFile;
import common.Order;
import common.OrderDBEntry;
import common.Product;
import logger.Logger;
import logger.LogLevel;

import java.lang.IllegalArgumentException;
import java.io.IOException;
import java.io.RandomAccessFile;
import java.io.File;
import java.lang.SecurityException;

public class OrderDB {
    public OrderDB(String dirPath) throws SecurityException,
                                          IOException,
                                          IllegalArgumentException {
        logger_ = Logger.getInstance();

        dirPath_ = dirPath;
        File file = new File(dirPath);
        if (¬ file.isDirectory()) {
            // Use mkdirs instead of mkdir, to create intermediate
            // directories if they does not exists
            file.mkdirs();
        }
    }

    public void add(Order order) throws IOException {
        RandomAccessFile file = this.getOrderFile(order, "rwd");
        FileLock lock = file.getChannel().lock();
        OrderDBEntry entry = new OrderDBEntry(order);

        // Go to the end of the file
        file.seek(file.length());
        file.write(entry.getBytes());
        lock.release();
        file.close();
    }

    public void alter(Order order) throws IOException {
        RandomAccessFile file = this.getOrderFile(order, "rwd");
        FileLock lock = file.getChannel().lock();
        long offset = this.getOffsetToEntry(file, order.id());

        // Sanity check
        if (offset ≡ −1) {
            // This should not happen. Stop program
            logger_.log(LogLevel.ERROR, "Order doesn’t exists in alter");
            System.exit(−1);
        }

        // Do a have to do this or the file is in the correct offset?
        file.seek(offset);
        OrderDBEntry entry = new OrderDBEntry(order);
        file.write(entry.getBytes());

        lock.release();
        file.close();
    }

    public Order get(UUID orderKey) throws IOException {
        RandomAccessFile file = this.getOrderFile(orderKey, "rwd");
        FileLock lock = file.getChannel().lock();
        long offset = this.getOffsetToEntry(file, orderKey);

```

```java
        if (offset ≡ −1) {
            lock.release();
            file.close();
            return null;
        }

        byte[] entryBuffer = new byte[OrderDBEntry.ENTRY_SIZE];
        file.seek(offset);
        file.read(entryBuffer, 0, OrderDBEntry.ENTRY_SIZE);
        OrderDBEntry entry = new OrderDBEntry(entryBuffer);

        lock.release();
        file.close();

        return entry.order();
    }

    private long getOffsetToEntry(RandomAccessFile file, UUID orderKey)
    throws IOException {
        byte[] buffer = new byte[OrderDBEntry.UUID_SIZE];
        try {
            file.seek(0);
            while(true) {
                int readBytes = file.read(buffer, 0, OrderDBEntry.UUID_SIZE);
                if (readBytes ≡ −1) {
                    // EOF reached
                    return −1;
                }

                // Create a UUID
                ByteBuffer bb = ByteBuffer.wrap(buffer);
                UUID uuid = new UUID(bb.getLong(), bb.getLong());

                if (uuid.equals(orderKey)) {
                    break;
                }

                // Jump to the next entry
                file.skipBytes(OrderDBEntry.ENTRY_SIZE −
                               OrderDBEntry.UUID_SIZE);
            }
        }
        catch (EOFException e) {
            // If this happen, then the product does not exists and we have
            // a bug in the system. ABORT!
            logger_.log(LogLevel.ERROR, "Order does not exists in OrderDB. "
                + "Order key: " + orderKey.toString());
            System.exit(−1);
        }

        // We must sustract the key that was read in the last comparison
        return file.getFilePointer() − OrderDBEntry.UUID_SIZE;
    }

    /**
     * @brief Get the file where the order must be stored in the DB
     **/
    private RandomAccessFile getOrderFile(Order order, String mode)
    throws IOException {
        String subUuid = order.stringID().substring(0, 2);
        return this.getOrderFile(subUuid, mode);

    }

    private RandomAccessFile getOrderFile(UUID uuid, String mode)
    throws IOException {
        String subUuid = uuid.toString().substring(0, 2);
        return this.getOrderFile(subUuid, mode);
    }

    private RandomAccessFile getOrderFile(String subUuid, String mode)
    throws IOException {
        String fileName = dirPath_ + "/" + subUuid;
```

```java

            // Again, we cannot check if the file exists. Just try to create it
            File orderFile = new File(fileName);
            orderFile.createNewFile();
            return new RandomAccessFile(fileName, mode);
        }

        private String dirPath_;
        private Logger logger_;
    }
```

```java
package common;

import java.nio.ByteBuffer;
import java.util.UUID;
import common.Order;
import common.OrderState;
import common.Product;

/**
 * DB ENTRY serialization structure:
 * UUID − 16 bytes
 * PRODUCT_TYPE − 10 bytes (pad with spaces)
 * AMOUNT − 8 bytes (long type)
 * ORDER_STATE − 15 bytes (pad with spaces)
 */

public class OrderDBEntry {
    public OrderDBEntry(Order order) {
        order_ = order;
    }

    /**
     * @brief Receives a DB entry in bytes, and deserialize it to get an Order
     */
    public OrderDBEntry(byte[] entry) {
        ByteBuffer bb = ByteBuffer.wrap(entry);

        // UUID;
        UUID uuid = new UUID(bb.getLong(), bb.getLong());

        // Product
        byte[] productBuffer = new byte[PRODUCT_SIZE];
        bb.get(productBuffer, 0, PRODUCT_SIZE);
        Product product = Product.valueOf(new String(productBuffer).trim());

        // Amount
        long amount = bb.getLong();

        // Order State
        byte[] stateBuffer = new byte[STATE_SIZE];
        bb.get(stateBuffer, 0, STATE_SIZE);
        OrderState state = OrderState.valueOf(new String(stateBuffer).trim());

        order_ = new Order(uuid, product, amount, state);
    }

    /**
     * @brief Serialiazes the Order stored as it should be stored in the DB
     * @return The order serialized as it would be stored in the DB
     */
    public byte[] getBytes() {
        ByteBuffer bb = ByteBuffer.allocate(ENTRY_SIZE);

        // UUID
        bb.putLong(order_.id().getMostSignificantBits());
        bb.putLong(order_.id().getLeastSignificantBits());

        // Product
        String product = String.format("%−10s",
                                       order_.productType().toString());
        bb.put(product.getBytes());

        // Amount
        bb.putLong(order_.amount());

        // Order State
        String state = String.format("%−15s", order_.state().toString());
        bb.put(state.getBytes());

        return bb.array();
    }

    public Order order() {
        return order_;
    }

    private Order order_;
    // 16 = UUID size (This shouldn’t change)
    public static final int UUID_SIZE = 16;
    public static final int PRODUCT_SIZE = 10;
    // 8 = Long size (I don’t expect this to change)
    private static final int AMOUNT_SIZE = 8;
    private static final int STATE_SIZE = 15;
    public static final int ENTRY_SIZE = PRODUCT_SIZE +
                                         STATE_SIZE +
                                         UUID_SIZE +
                                         AMOUNT_SIZE;
}
```

```
package client;

import java.util.concurrent.locks.ReentrantLock;
import java.util.concurrent.locks.Lock;
import java.lang.Thread;
import java.lang.Runtime;
import java.util.Iterator;
import java.util.UUID;
import java.util.ArrayList;
import java.lang.Math;
import java.lang.System;
import java.util.Random;
import java.util.concurrent.TimeoutException;
import java.io.IOException;
import java.util.Random;

import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.Channel;
import org.apache.commons.lang3.SerializationUtils;

import common.Order;
import common.OrderState;
import common.Product;
import configParser.ConfigParser;
import logger.Logger;
import logger.LogLevel;

public class MainClass extends Thread {
    public MainClass(String[] argv) {
        randomGenerator_ = new Random(System.currentTimeMillis());
        config_ = ConfigParser.getInstance();
        logger_ = Logger.getInstance();
        lock_ = new ReentrantLock();
        ordersKeys_ = new ArrayList<UUID>();

        config_.init(argv[1]);
        this.initLogger(argv[0]);
    }

    public static void main(String[] argv) throws InterruptedException {
        ConfigParser config = ConfigParser.getInstance();
        Logger logger = Logger.getInstance();
        try {
            MainClass app = new MainClass(argv);
            Runtime.getRuntime().addShutdownHook(app);

            app.initRabbit();
            logger.log(LogLevel.INFO,
                "Proceed to create and send orders");
            app.sendOrders();

            int sleepTime = Integer.parseInt(config.get("CLIENT",
                "sleep−between−orders−and−queries", "0"));

            if (sleepTime > 0) {
                logger.log(LogLevel.INFO,
                    "Proceed to sleep before send queries to the system");
                Thread.sleep(sleepTime * 1000);
            }

            logger.log(LogLevel.INFO,
                "Proceed to send queries associated with the orders created");
            app.queryOrders();
            app.terminate();
        }
        catch (IllegalArgumentException e) {
            // We couldn’t open the logger. Just exit
            System.out.println(e);
            System.exit(−1);
        }
        catch (TimeoutException e) {
            logger.log(LogLevel.ERROR, e.toString());
```

```java
            }
            catch (IOException e) {
                logger.log(LogLevel.ERROR, e.toString());
            }
        }

        private void initLogger(String processNumber)
        throws IllegalArgumentException {
            String logFileName = config_.get("MAIN", "log−file");
            String logLevel = config_.get("MAIN", "log−level");

            Logger logger = Logger.getInstance();
            logger.init(logFileName, LogLevel.parse(logLevel));
            logger.setPrefix("[CLIENT " + processNumber + "]");
            logger.log(LogLevel.DEBUG, "Process started");
        }

        public void initRabbit() throws IOException,
                                        TimeoutException,
                                        IllegalArgumentException {
            ConnectionFactory factory = new ConnectionFactory();
            factory.setHost(config_.get("MAIN", "server−address", "localhost"));
            connection_ = factory.newConnection();
            channel_ = connection_.createChannel();

            clientQueue_ = config_.get("QUEUES", "client−queue");
            channel_.queueDeclare(clientQueue_,
                                  false,
                                  false,
                                  false,
                                  null);

            queryQueue_ = config_.get("QUEUES", "query−queue");
            channel_.queueDeclare(queryQueue_,
                                  false,
                                  false,
                                  false,
                                  null);
        }

        public void terminate() throws IOException, TimeoutException {
            channel_.close();
            connection_.close();
        }

        public void sendOrders() throws IOException {
            int ordersToCreate =
                Integer.parseInt(config_.get("CLIENT",
                                             "amount−orders−to−simulate",
                                             "1"));
            logger_.log(LogLevel.DEBUG, "Orders to simulate: "
                + ordersToCreate);

            for (int i = 0; i < ordersToCreate; ++i) {
                Order order = this.generateRandomOrder();
                byte[] data = SerializationUtils.serialize(order);

                // Store the UUID generated to then make a query to the system
                ordersKeys_.add(order.id());

                logger_.log(LogLevel.DEBUG, "Sending order: " + order.stringID());
                channel_.basicPublish("", clientQueue_, null, data);
            }
        }

        /**
         * @brief Sends as much queries as the parameter amount−queries−to−simulate
         * @details If amount−queries−to−simulate is bigger than
         * amount−orders−to−simulate, a round−robin algorithm is used to keep
         * querying orders
         */
        public void queryOrders() throws IOException {
            int amountQueries =
```

```java
                Integer.parseInt(config_.get("CLIENT",
                                             "amount−queries−to−simulate",
                                             "1"));

            Iterator<UUID> it = ordersKeys_.iterator();
            while (amountQueries > 0) {
                −−amountQueries;

                UUID key = it.next();
                byte[] data = SerializationUtils.serialize(key);
                logger_.log(LogLevel.DEBUG, "Querying order: " + key.toString());
                channel_.basicPublish("", queryQueue_, null, data);

                if (¬ it.hasNext()) {
                    it = ordersKeys_.iterator();
                }
            }
        }

        private Order generateRandomOrder() {
            long amount = Math.abs(randomGenerator_.nextInt() % 10) + 1;
            return new Order(Product.randomProduct(), amount);
        }

        public void run() {
            try {
                this.terminate();
            }
            catch (TimeoutException e) {
            }
            catch(IOException e) {
            }
        }

        private Logger logger_;
        private ConfigParser config_;
        private Random randomGenerator_;
        private Channel channel_;
        private Connection connection_;
        private String clientQueue_;
        private String queryQueue_;
        private Lock lock_;
        private ArrayList<UUID> ordersKeys_;
    }
```

```
package auditLogger;

import java.lang.Runtime;
import java.lang.Thread;
import java.util.concurrent.TimeoutException;
import java.io.IOException;

// External libraries includes
import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Consumer;
import com.rabbitmq.client.DefaultConsumer;
import com.rabbitmq.client.Envelope;
import com.rabbitmq.client.AMQP;
import org.apache.commons.lang3.SerializationUtils;

import auditLogger.AuditLogger;
import common.Order;
import configParser.ConfigParser;
import logger.Logger;
import logger.LogLevel;


public class MainClass {
    public static void main(String[] argv) {
        ConfigParser config = ConfigParser.getInstance();

        Logger logger = Logger.getInstance();

        try {
            MainClass app = new MainClass();
            config.init(argv[1]);
            app.initLogger(config, argv[0]);

            Channel channel = app.initRabbit(config);
            String auditLogFile = config.get("AUDIT", "audit-log-file");
            String auditLogQueue = config.get("QUEUES", "audit-log-queue");
            Consumer consumer = new AuditLogger(channel, auditLogFile);
            channel.basicConsume(auditLogQueue, true, consumer);
        }
        catch (IllegalArgumentException e) {
            // We couldn't open the logger. Just exit
            System.out.println(e);
            System.exit(-1);
        }
        catch (TimeoutException e) {
            logger.log(LogLevel.ERROR, e.toString());
        }
        catch (IOException e) {
            logger.log(LogLevel.ERROR, e.toString());
        }
    }

    private void initLogger(ConfigParser config, String processNumber)
    throws IllegalArgumentException {
        String logFileName = config.get("MAIN", "log-file");
        String logLevel = config.get("MAIN", "log-level");

        Logger logger = Logger.getInstance();
        logger.init(logFileName, LogLevel.parse(logLevel));
        logger.setPrefix("[AUDIT_LOG " + processNumber + "]");
        logger.log(LogLevel.DEBUG, "Process started");
    }

    private Channel initRabbit(ConfigParser config)
    throws IllegalArgumentException,
           IOException,
           TimeoutException {
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost(config.get("MAIN", "server-address", "localhost"));
        Connection connection = factory.newConnection();
        Channel channel = connection.createChannel();
```

```

        String auditLogQueue = config.get("QUEUES", "audit-log-queue");
        // To secure fairness between the processes
        channel.basicQos(1);
        channel.queueDeclare(auditLogQueue,
                             false,
                             false,
                             false,
                             null);

        return channel;
    }
}
```

```java
package auditLogger;

import java.text.SimpleDateFormat;
import java.util.Date;
import java.text.DateFormat;
import java.io.File;
import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Consumer;
import com.rabbitmq.client.DefaultConsumer;
import com.rabbitmq.client.Envelope;
import com.rabbitmq.client.AMQP;
import org.apache.commons.lang3.SerializationUtils;

import common.Order;
import logger.Logger;
import logger.LogLevel;

import java.io.IOException;
import java.io.FileWriter;


public class AuditLogger extends DefaultConsumer {
    public AuditLogger(Channel channel,
                       String auditLogFile) throws IOException {
        super(channel);
        logger_ = Logger.getInstance();
        // Open log file in append mode
        File file = new File(auditLogFile);
        dateFormat_ = new SimpleDateFormat("yyyy/MM/dd HH:mm:ss");

        if (¬ file.exists()) {
            logger_.log(LogLevel.WARNING,
                "Audit log doesn’t exists."
                + " Proceed to create it. AuditLogFile: " + auditLogFile);
            file.createNewFile();
        }
        writer_ = new FileWriter(auditLogFile, true);
    }

    @Override
    public void handleDelivery(String consumerTag,
                               Envelope envelope,
                               AMQP.BasicProperties properties,
                               byte[] body) throws IOException {
        Order newOrder = (Order) SerializationUtils.deserialize(body);
        logger_.log(LogLevel.DEBUG, "Order received: " + newOrder.stringID());
        writer_.write(this.generateAuditEntry(newOrder) + "\n");
        writer_.flush();
    }

    private String generateAuditEntry(Order order) {
        Date date = new Date();
        String entry = dateFormat_.format(date) + " − ";
        entry += "Order ID: " + order.stringID();
        return entry;
    }

    private Logger logger_;
    private FileWriter writer_;
    private DateFormat dateFormat_;
}
```

```ini
[MAIN]
absolute−path = /media/Datos/Facultad/75.61 Taller III/Order_Dispatcher/
processes−config−file = /media/Datos/Facultad/75.61 Taller III/Order_Dispatcher/configuration.ini
classpath = lib/ini4j−0.5.4/ini4j−0.5.4.jar:lib/rabbitmq−java−client−bin−3.5.4/rabbitmq−client.jar:lib/ini4j−0.5.4/ini4j−0.5.4.jar:lib/commons−lang3−3.4/commons−lang3−3.4.jar

# All the paths in the classpaths classes must be relative paths to the
# absolute−path given in the main section
# Also, the paths who have spaces must not be escaped. The app will do it
[REQUEST−DISPATCHER]
classpath = build/jar/RequestDispatcher.jar
class−name = requestDispatcher.MainClass
run = true
kill = true
amount = 1

[CLIENT]
classpath = build/jar/Client.jar
class−name = client.MainClass
run = false
kill = true
amount = 1

[EMPLOYER]
classpath = build/jar/Employer.jar
class−name = employer.MainClass
run = false
kill = true
amount = 1

[AUDIT−LOGGER]
classpath = build/jar/AuditLogger.jar
class−name = auditLogger.MainClass
run = true
kill = true
# This cannot be a value higher than one
amount = 1

[STOCK−MANAGER]
classpath = build/jar/StockManager.jar
class−name = stockManager.MainClass
run = true
kill = true
amount = 1

[ORDER−MANAGER]
classpath = build/jar/OrderManager.jar
class−name = orderManager.MainClass
run = true
kill = true
amount = 1

[QUERY−SOLVER]
classpath = build/jar/QuerySolver.jar
class−name = querySolver.MainClass
run = true
kill = true
amount = 1
```

Sep 22, 15 23:58 **configuration.ini** Page 1/1

```
[MAIN]
server-address = localhost
log-file = /tmp/OrderDispatcher.log
log-level = DEBUG

[QUEUES]
client-queue = CLIENT-QUEUE
audit-log-queue = LOG-QUEUE
stock-manager-queue = STOCK-QUEUE
order-manager-queue = ORDER-QUEUE
query-queue = QUERY-QUEUE
delivery-queue = DELIVERY-QUEUE

[AUDIT]
audit-log-file = /tmp/Audit.log

[STOCK]
stock-db-file = /tmp/Stock.db

[CLIENT]
amount-orders-to-simulate = 1000
amount-queries-to-simulate = 1000
# In seconds
sleep-between-orders-and-queries = 20

[ORDER]
order-db-directory = /tmp/OrderDB

[STOCK-PROVIDER]
global-increase = 10
type-1-increase = 1
type-2-increase = 2
type-3-increase = 3
type-4-increase = 4
type-5-increase = 5

[EMPLOYER]
amount-orders-to-process = 100
```

Sep 24, 15 1:24 **Table of Content** Page 1/1

Table of Contents