Assignment 2: Group Report

Heriot-Watt University

Advanced Software Engineering

21/03/2019

Group 2:

Athanasiou, Ioannis ia50@hw.ac.uk

Nanduri, Ram rnn1@hw.ac.uk

Schatton, Bartosz bps2@hw.ac.uk

Triffterer, Thomas tt63@hw.ac.uk

Repository:

https://github.com/trz-maier/hwu-ase-group2

Jar:

https://github.com/trz-maier/hwu-ase-group2/raw/master/hwu-ase-cw2-group2.jar

Contributions:

Ioannis: Design contributions; scaffolding; Priority orders implementation; random queue ordering fixes; code quality maintenance; report alignment with new processes;

Ram: Log class; OrderQueue; Log methods, formatting and writing to Log file (singleton design pattern); Pausing and resuming the order processing (program);

Bartosz: ServerFrame; QueueFrame; OrderQueue threading; Server pausing/resuming; Server adding/removing; random Order addition; Order processing speed adjustment; Bug fixes; Kanban board setup and management;

Thomas: Design contributions; Server; ServerController; ApplicationCloseTask; Bug fixes; Server pausing/resuming;

Objective

Java café order processing simulation with multiple queues and servers using threading

Program status

Program meets core specifications fully and runs without errors

Known issues

* None noted

Known limitations

* Order processing is not shown in great detail
* When last item in an Order is being processed Server thread can be stopped
* Program cannot be closed until Orders are processed

Design choices

* It has been decided to use separate ServerFrame windows for each of the Servers to add and remove Servers freely. Nesting Servers within the main frame would mean the total number of Servers would have to be limited.

Enhancements

* Priority queue
* Pausing and resuming program
* Putting servers on break and resuming
* Adding and removing servers
* Order processing speed adjustment
* Randomized order addition

Priority queue

The priority queue feature was achieved by simply having a single Java BlockingQueue. The PriorityBlockingQueue implementation was utilized, replacing of the previous LinkedBlockingQueue. Once that was done, the Order class had to be given a priority attribute, along with a Comparable interface implementation.

At that point the issue was mostly resolved. There were a few extra steps taken to ensure that all the structures behaved correctly in a multi-threaded environment, such as moving to Vectors from ArrayLists.

Putting Servers on-break and resuming

Using Server statuses FREE and BUSY along with TO_BE_PAUSED and PAUSED allowed to create a logic whereby a server that was requested to go on break, but is still processing orders, to be paused when it is done processing the current order. This is achieved by using synchronized wait() and notify() function calls in the server thread along with server status information. The status changes can be seen in the respective ServerFrame and in the log so the process is easy to follow.

```
@Override
public void startOrderProcess() {
    if (serverThread == null) {
        serverThread = new Thread(new ServerRunnable());
        serverThread.setName(name);
        serverThread.start();
    } else {
        System.out.println(getName() + "already started");
    }
}

@Override
public void pauseOrderProcess() {
    // this does not pause the Status immediately but waits until current order is processed
    shouldPause = true;
    synchronized (serverThread) {
        serverThread.interrupt();
    }
}

@Override
public void restartOrderProcess() {
    shouldPause = false;
    clearStatus();
    synchronized (serverThread) {
        serverThread.notify();
    }
}
```

Pausing and resuming program

Pausing and resuming a program included two components. The first was pausing/restarting all servers and the second was pausing/restarting the Order Producer. This was our agreed definition for pausing and resuming a program.

To pause and restart servers the methods for pause and unpause servers were called for all servers within the method to pause and restart (as shown below). Then pause/restart for the order queue is called.

```
public void startProcessing() {
    for (ServerController serverController : serverList) {
        serverController.unPause();
    }
    orderProducer.restartOrderProcess();
}

public void pauseProcessing() {
    for (ServerController serverController : serverList) {
        serverController.pause();
    }
    orderProducer.pauseOrderProcess();
}
```

Synchronised wait and notify are used to implement pausing and restarting the Order Producer. Synchronised notify() in the restartOrderProcess() and a synchronised wait() in the run(). A variable shouldPause will determine whether to pause or not.

```
@Override
public void run() {
    for (int i = 0; i < this.loadedOrders.size(); i++) {
        synchronized (currentThread) {
            while (shouldPause) {
                try {
                    currentThread.wait();
                } catch (InterruptedException e) {
                }
            }
        }

        int delay = (int) (Math.random() * maxDelayTime);
        Order order = loadedOrders.get(i);
        try {
            Thread.sleep(delay);
            increaseCounter();
            order.setCreationOrder(this.counter);
            listener.onOrderProduced(order);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

public void pauseOrderProcess() {
    shouldPause = true;
}

public void restartOrderProcess() {
    shouldPause = false;
    synchronized (currentThread) {
        currentThread.notify();
    }
}
```

Adding and removing Servers

Similarly, to pausing and resuming Servers they can also be easily added and removed on request. This process uses the same Server status whereby a newly created Server is set to be FREE and ready to pick up new Orders, while a Server that is requested to be closed waits until the current order is processed with a status TO_BE_STOPPED before the Thread is stopped and ServerFrame disposed of. Adding a Server simply involves calling its constructor through the Controller which also starts an associated ServerFrame. Servers are stored by the Controller in a list for easy access.

```
public void addServer() {
    synchronized (serverList) {
        if (!applicationClosing) {
// If the application is currently closing(if the factory is done and the
// queue is empty) it is not allowed to add new servers
            currentServerNumber = this.serverList.size() + 1;
            ServerController sc = new ServerController(
                                currentServerNumber, queueFrame, queuedOrders, this);
            this.serverList.add(sc);
            sc.setOrderProcessTime(BASE_PROCESSING_TIME);

        }
    }
}

public void removeServer() {
    synchronized (serverList) {
        if (!applicationClosing) {
// If the application is currently closing(if the factory is done and the queue
// is empty) it is not allowed to remove servers
            if (serverList.size() > 0) {
                ServerController closingServer = serverList.get(serverList.size() - 1);
                closingServer.stop();
                serverList.remove(serverList.size() - 1);
            }
        }
    }
}
```

Order processing speed adjustment

JSlider in the QueueFrame is used to adjust the speed factor which is used to alter the number if milliseconds in the .wait(milliseconds) function call on both production and processing of orders. The slider range is 1 to 19 which is then transformed into a speed factor 0.1 to 1.9 and multiplied by the starting processing time in milliseconds. Both OrderQueue and Server have respective methods to amend the processing time while the ChangeListener interface implemented into QueueFrame ensures user interaction with the slider is responded to immediately.

```
@Override
public void stateChanged(ChangeEvent e) {
    if (e.getSource() == speedSlider) {
        if (!speedSlider.getValueIsAdjusting()){
            oc.setProcessingSpeed((20-speedSlider.getValue())/10.0);
        }
    }
}


public void setProcessingSpeed(double factor) {
    for (ServerController serverController : serverList) {
        int time = (int) (BASE_PROCESSING_TIME * factor);
        serverController.setOrderProcessTime(time);
        orderProducer.setMaxDelayTime(time);
    }
}
```

Random order addition

Both priority and non-priority orders can be added to the queue during run-time by using order randomizer. Such orders are appended to the queue list making listener calls to inform the QueueFrame that a new order has been added so that it updates and logs this event. The use of listeners ensures the additions are immediately reflected in the QueueFrame and displayed to the user.

```
public void addRandomOrder(boolean priority) throws InvalidCustomerIdException {
    String customerId = "C" + randomSeed.nextInt(1000000, 9999999 + 1);
    int noOfItems = randomSeed.nextInt(1, 5);
    Order order = new Order(customerId, priority);
    for (int itemCount = 0; itemCount < noOfItems; itemCount++) {
        Object[] itemKeys = stockItems.keySet().toArray();
        int randomIndex = randomSeed.nextInt(0, itemKeys.length);
        Item item = stockItems.get(itemKeys[randomIndex]);
        order.addOrderItem(item);
    }
    orderProducer.increaseCounter();
    order.setCreationOrder(orderProducer.getCounter());
    onOrderProduced(order);
    this.extraRandomOrdersAdded++;
}
```

Agile methodology

Timeline

[1] During our first meeting we decided how to leverage our code from Phase 1 and transform it into a program that fulfills the requirements of Phase 2. We decided to do a bit of clean up and amend the order processing functionality to fit the new objective. We also had a brief design discussion and agreed on using Sprints agile techniques. We divided the workload into 4 Sprints with clear objectives. Kanban board was set up with appropriate cards and labelling.

[2] The second meeting took place after Sprint 1 was concluded where we discussed and added into our Project planner all of the core requirements as well as added a few enhancement ideas. Each of us through a few into the basket and we decided to select some for development later on.

[3] During the main development sprints we worked on core and enhanced features mostly individually however we have put a lot of effort in reviewing each other's codes and challenging design choices. Many pull requests had detailed discussions that often resulted in a number of changes before merge commits. We have also met on several occasions to discuss issues and help each other with implementation.

[4] Our final meeting took place before the final Sprint where we discussed all the remaining issues and how to fix them. This was an important meeting due to the complexity of some of the issues encountered and it allowed us to share ideas and test solutions before deciding on the correct fixes.

Development method

Sprints

Sprints

Sprint 1 [due: 04-03-2019]: Created new classes and adjusted existing in preparation for further work.

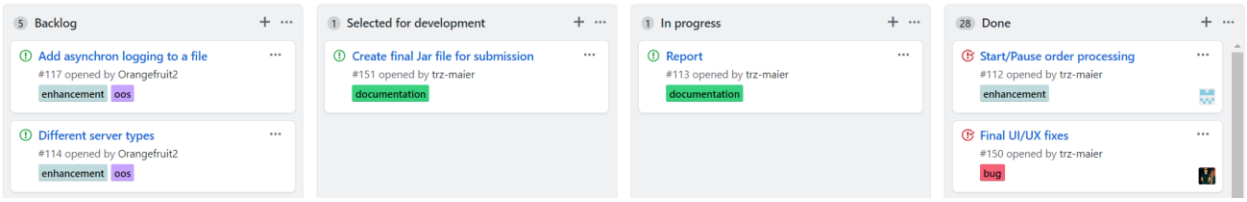
Sprint 2 [due: 11-03-2019]: Implemented all core requirements.

Sprint 3 [due: 16-03-2019]: Implemented enhancements and fixed issues relating to core requirements.

Sprint 4 [due: 20-03-2019]: Fixed all remaining issues. Prepared submission files.

Management

Kanban board with custom cards using GitHub Project functionality. [Link here](#).



Cards

[Backlog] – proposed ideas and newly reported issues
[Selected for development] – features, issues and tasks to be worked on in current sprint
[In progress] – elements currently worked on
[Done] – completed elements

Labels

[core] – core features as per requirements
[enhancement] – additional features developed as enhancements
[bug] – bugs and issues found in code
[documentation] – project documentation and main submission report
[testing] – unit testing tasks
[duplicate] – duplicate issue or feature
[help wanted] – asked for help in fixing an issue or developing a feature
[oos] – out of scope issue or feature
[invalid] – incorrect issue or feature

Development experience

Thomas

Through the sprints a better workflow was provided. There was more knowledge sharing than in the first task. Agile allowed us to implement new ideas/improvements which we had during the project. One problem was that sometimes different design ideas existed how to solve the same problem. Since we have not had have a fixed plan for the alliterations, this led to programming algorithms, which in the end were not needed. All in all, agile is a great way how to program, but there should be always a clear and understandable final project aim.

Ioannis

Agile proved to be vastly different from a plan-driver approach. Both cases had their merits, such as the steady, reliable pace of work found in the latter. However, it would be impossible to deny that in the 2nd Stage many more features were realized, while also doing a more complex, multi-threaded implementation. Partially this difference perplexes the matter of comparing these methodologies as, ideally, they would be used by the same or similar teams to perform identical work.

Development under agile enabled more frequent communication and flexibility in development. While a lot of features were developed, the pace was less consistent, and the team was lacking a "common vision". This led to the need for further talks, clarifications & revisions to parts of code that, although functional in a way, might not have been what other members of the team were visualizing. This great disparity in quality, styles and ideas in the software probably highlights the need for a more vibrant leading and decision-making pillar in the team for defining goals and direction.

In retrospect, this lack of central authority was perhaps a mistake, perhaps necessary in the scope of a university assignment. One does though need to note, that the main criticism of the eXtreme Programming agile methodology has been for years this same problem. And this problem is what led to the creation of the more modern, popular agile approaches like Kanban & Scrum, placing a stronger emphasis on the managerial side of things.

Ram

In the first Stage we found that we were often debating designs to implement a solution. However, this meant that we all had a clear idea of what we were doing and knew what our final solution would look like. Although our approach was thorough and well thought out, we found that we needed to make small considerations and changes which are difficult to foresee. I also felt that considerable time had elapsed before we made a significant implementation. Also, with lack of any "leader", this only added to the debate and the time elapsed. At times it felt like we were running around in circles.

In comparison, in the second stage initial discussion and implementation was very quick. Agile also meant that we were managing our time more effectively. However, I felt from a personal perspective that I did

not have that clear an idea of what our end implementation was going to look like. I also felt that although we discussed what we were each going to be doing, we had left out exactly how they would go about implementing this. This did not matter in the first part as we were each responsible for different packages of the code. For the second stage however we were each implementing certain features instead. The clarity needed was missing. This led to bugs (duplication of OrderQueue thread). Resolving bugs often meant meeting with a team mate to discuss their implementation or support from them in the form of GitHub comments on Pull Requests. This however can be argued to be the result of the way in which work was shared.

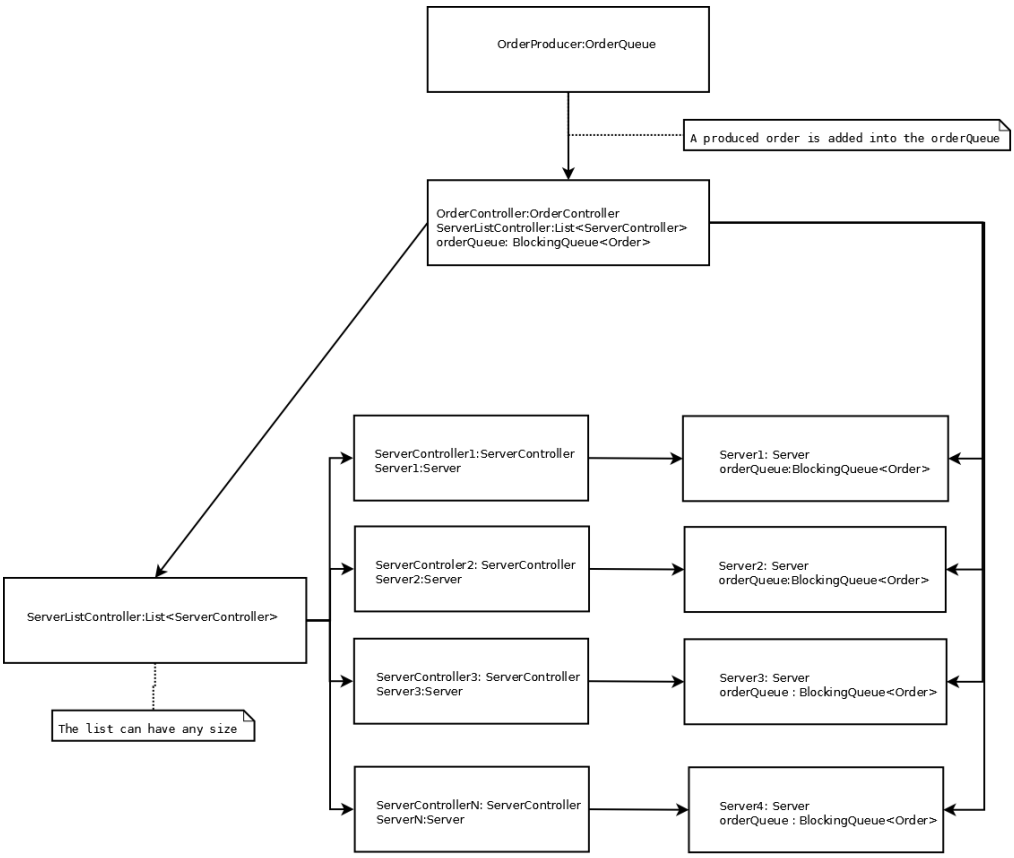
Bartosz

In comparison to Stage 1 where we had designed the entire program before we started coding, Stage 2 seemed a bit more chaotic. I appreciate that the objective changed, the program we were to develop was more complex, however I felt that being able to discuss the design in detail first made the work smoother and the development more predictable. I did however find that Agile development gave us more freedom in designing the features the way each of us thought was right and the group worked closely together to challenge and improve upon these ideas. Generally speaking, discussing ideas was more effective in Stage 2. I found that appropriately managing the workload using Kanban boards to be of a great help in driving feature development and bug fixing to close. The group worked collaboratively and although at times chaotic we delivered a program that meets the specifications and introduced several good enhancements. On the flipside, not discussing design choices at the beginning of the development meant that there has been a bit of frustration in the process where ideas were challenged in some heated discussions. That being said Agile offers quite a lot of different techniques so it is flexible enough to be able to satisfy both the project and group characteristics.

Design patterns

Producer-consumer

We use the producer consumer pattern to produce the orders with a delay inside the OrderQueue class and consume them in the Server class.

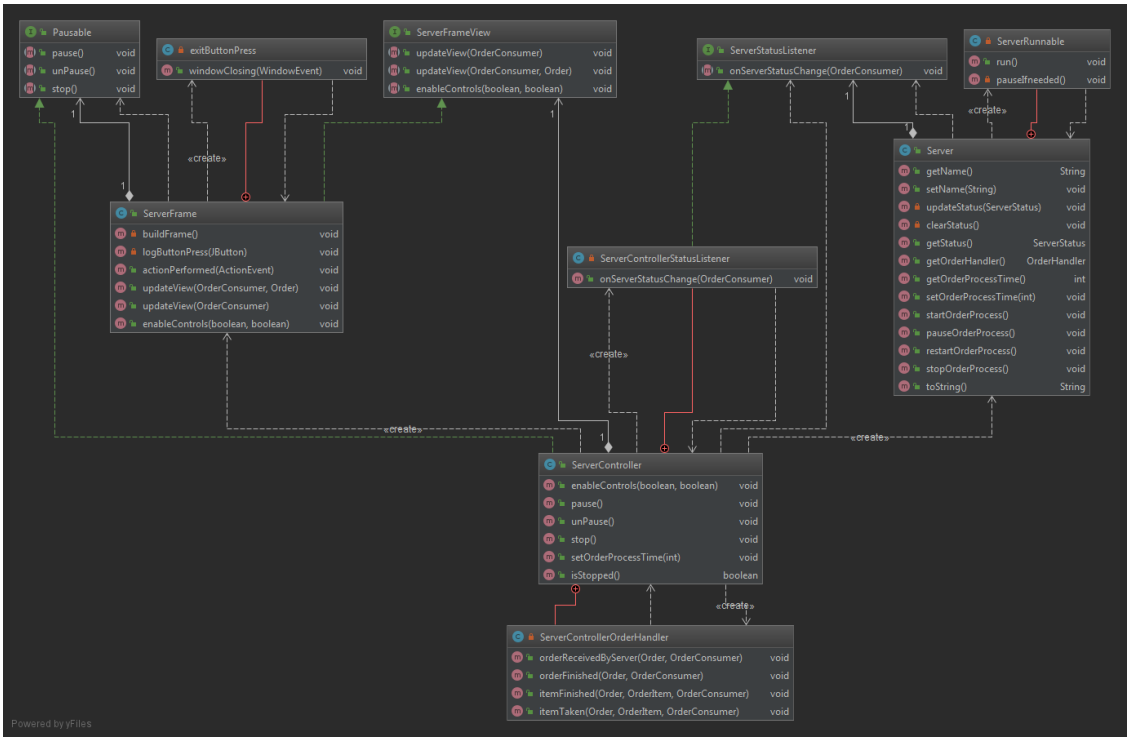


A class diagram cannot show the Producer-consumer pattern suitable, because the classes are connected via a shared object. Therefore, we decided to show this pattern as Object diagram.

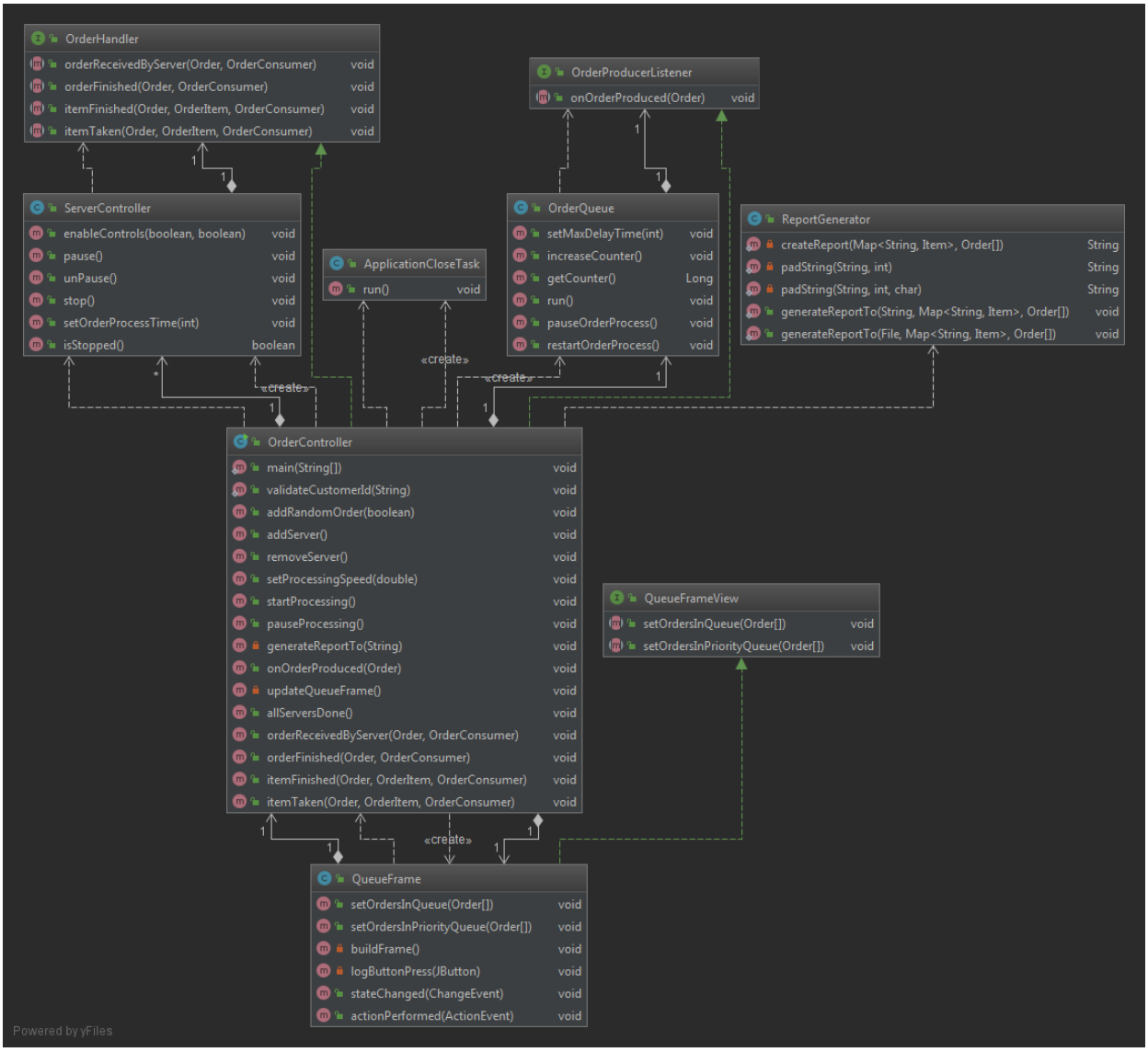
MVP

A partial MVP approach was used in Stage 1. For Stage 2 we moved to full MVP pattern, which was achieved in a number of ways. The model package, Frame classes which are only concerned with the GUI, and control package were retained. However, we added View interfaces to be used by old and new controllers. This was done to offer better further abstraction and a much better support for unit testing things like simulated GUI events.

We use MVP two times. One MVP is designed to show the status of a server inside a window, whereas another MVP is used to show the current order queue.



The Server MVP



The orderController mvp

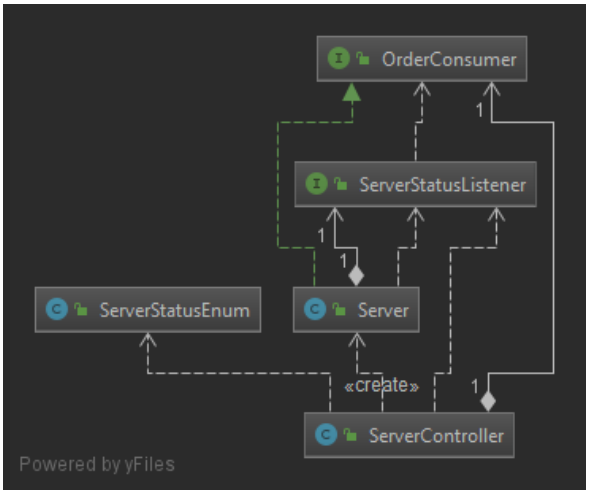
Singleton

Our Log class was implemented using the Singleton pattern. It was achieved through the usage of a private constructor and a single, withheld instance that would be returned to any caller of the static getLogger() method. The getLogger() uses double checked locking. Two other methods can be called on the instance of this class. One to log an event - log() and the other to write the log to a file - writeToLogFile().

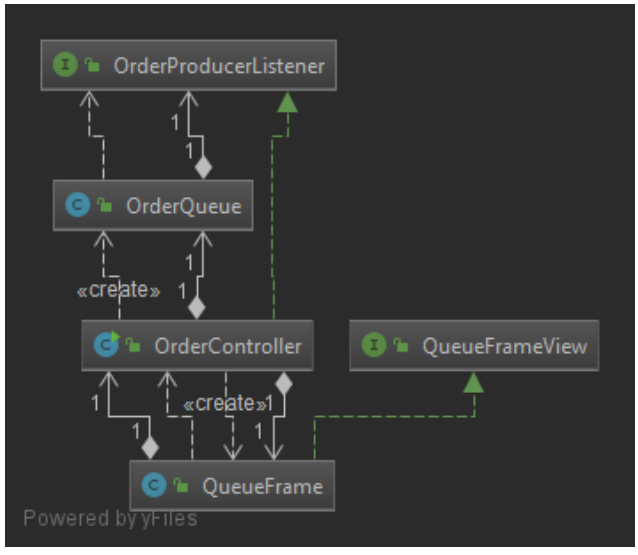
The benefit of a Singleton is that only a single instance is created, without wasting memory for instances whose variety would not matter for a logger feature set. Singleton is also valuable in terms of constraints where having a single entity is important for state consistency, this is particularly important for writing all events to a Log file on the exit of the application. It must be ensured that all the events belong to the same log instance. Therefore, the Singleton design pattern is the most obvious choice in implementing a Log class. We chose to place the Log class in its own log package outside model, view, control and other packages.

Listener

Listeners are used inside the MVP and Producer-consumer patterns. ServerStatusListener was created to inform the program of Server status change and update elements accordingly. OrderProducerListener is called each time an Order is produced to update the QueueFrame in real-time. ChangeListener and ActionListener are interfaces implemented by the QueueFrame to respond to the user interacting with GUI.



The serverStatus listener



The orderQueue listener

Threads

We use a Thread to produce the order Queue and multiple Threads to consume the order Queue. Each Server is one consumer which run inside an own thread.

For each Server who is stopped, a stopping thread is created. Because a server will first finish pending order, the Thread which stops the Server is blocked. If we had stopped the server inside the UI thread, this would have blocked the UI Thread.

One additional Thread is created which waits for the order producer to produce all Orders and for the server to consume all orders. After all orders were produced and consumed, this Thread stops the application.

Screen shot – Main Program