Technical University of Cluj-Napoca

Faculty of Automation and Computer Science

Computer Science Department

Year 2015-2016 Semester II

Programming Techniques

*3rd Project*

Student: Dan Toderici

Group: 30422

[Project specification 3](#_Toc448230096)

[Problem Analysis 4](#_Toc448230097)

[Modeling 4](#_Toc448230098)

[Use Case Diagram 4](#_Toc448230099)

[Design 5](#_Toc448230100)

[Relational Diagram 5](#_Toc448230101)

[Class Structure 7](#_Toc448230102)

[Implementation 7](#_Toc448230103)

[Model 7](#_Toc448230104)

[Task Class 7](#_Toc448230105)

[Simulator Class 8](#_Toc448230106)

[Scheduler Class 8](#_Toc448230107)

[Server Class 8](#_Toc448230108)

[1:1 (kernel-level threading) 11](#_Toc448230109)

[N:1 (user-level threading) 12](#_Toc448230110)

[M:N (hybrid threading) 12](#_Toc448230111)

[Views 12](#_Toc448230112)

[Server Panel 12](#_Toc448230113)

[Server Frame 12](#_Toc448230114)

[Conclusion and further developments 12](#_Toc448230115)

[References 13](#_Toc448230116)

# ****Project specification****

Propose is to design and implement a system that simulates an queue simulation application for processing clients tasks. It is going to be just like a shop implementation but simplified and the objective for this project isn’t the shop itself, but processing task in parallel.

The application, that resulted from design such system, can perform 11 main operations:

1) Insertion of number of queues maximum allowed to be opened at once

2) Insertion of threshold for each queue

3) Creation of tasks, which have a random duration

4) Distribution of tasks to a queue that has not reached its threshold

5) Opening of new queues when the thresholds for the already opened queues were succeeded

6) Execution of task at server level

7) Computing the average waiting time at a queue

8) Computing the average service time at a queue

9) Computing the peak hour

10) Displaying the distribution of task to each queue and the opening of new queues

11) Log events in the console for a better understanding of threads processes

However, all these operations must be executed based on user input. Therefore, a graphical user interface is needed. The UI must provide the user methods to insert the number of queues and the threshold desired. The will not be many views and controllers, but a main controller is required.

# Problem Analysis

Splitting the problem into smaller parts is the first main problem. Starting from the design of the Task to the Server itself, it is to be considering a strong and trustworthy design. Many multithreading problems may appear due to poor synchronizations or requests without pause, when there is no room left for another task. However no patterns where specified in implementing this project and therefore pattern-wise this is not a difficult project. However the architecture was key in solving the problem.

# Modeling

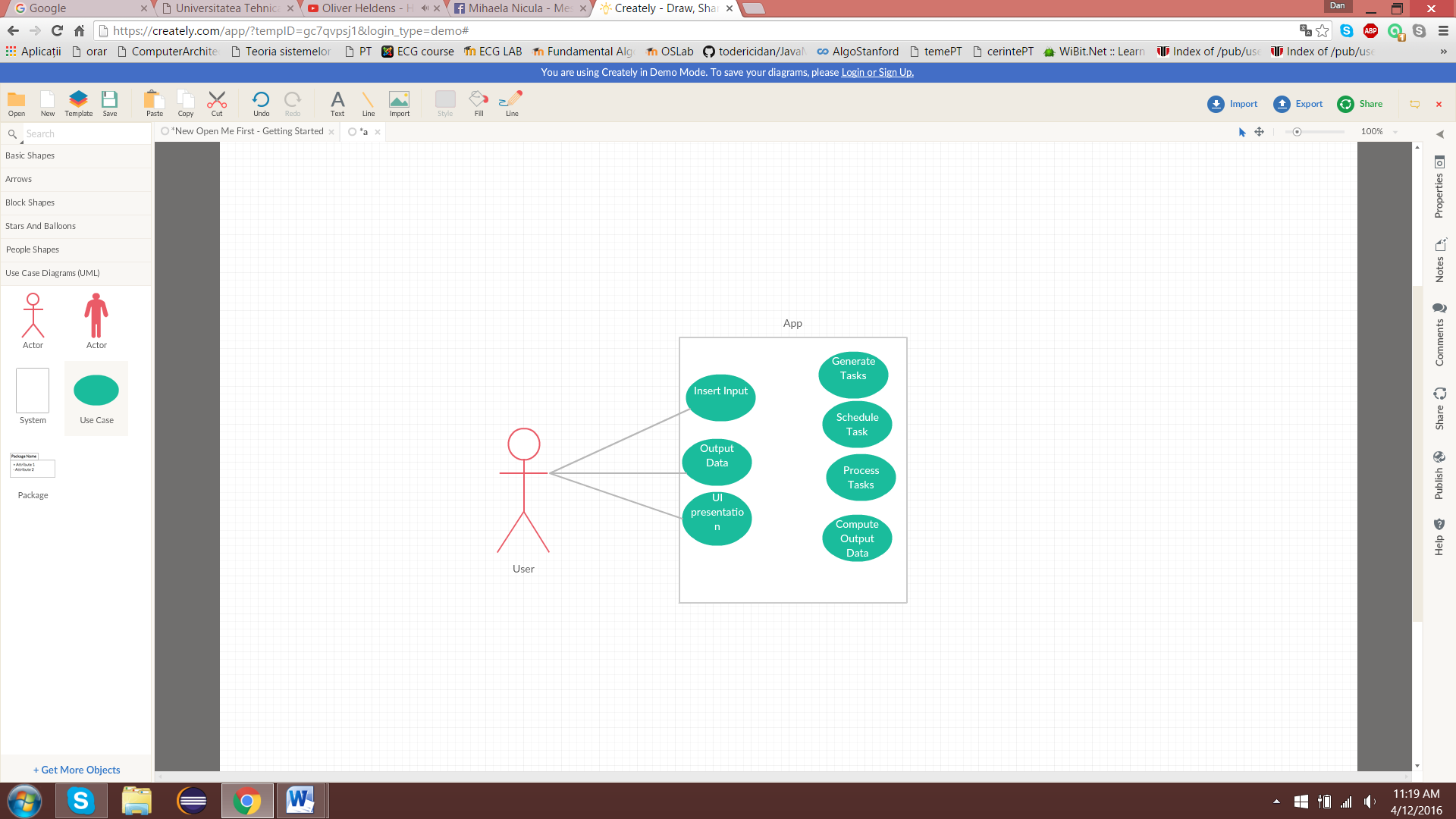
In order to keep things as basic and clean as possible a structure of fetching data from user and computing the simulation data based on them. Afterwards the simulation is quite easy. Tasks are inserted in a Server or sent to a Server however you want to call it and due to a BlockingQueue they are placed there for waiting to be processed. Simulator must first generate them and send them off to the Scheduler. The Scheduler based on the info gathered from the servers, if they have reached threshold or not.

## Use Case Diagram

This type of diagrams have an important role in showing how an user may interact with some features of a certain application. In software and systems engineering, a use case is a list of actions or steps, that define the interaction between a role and a sytem( in the Unified Modeling Language the role is also known as an actor) in order to achieve a goal.

In this case, there are many use case, this chapter will only display and explain some of them (for example inserting number of servers and threshold, generate tasks, schedule task, process tasks, compute tasks and present the data to the user).

As far as the operation are concerned the, the user has to input the before mention info and will see the simulation afterwards until a number of clients enter and finish their tasks.



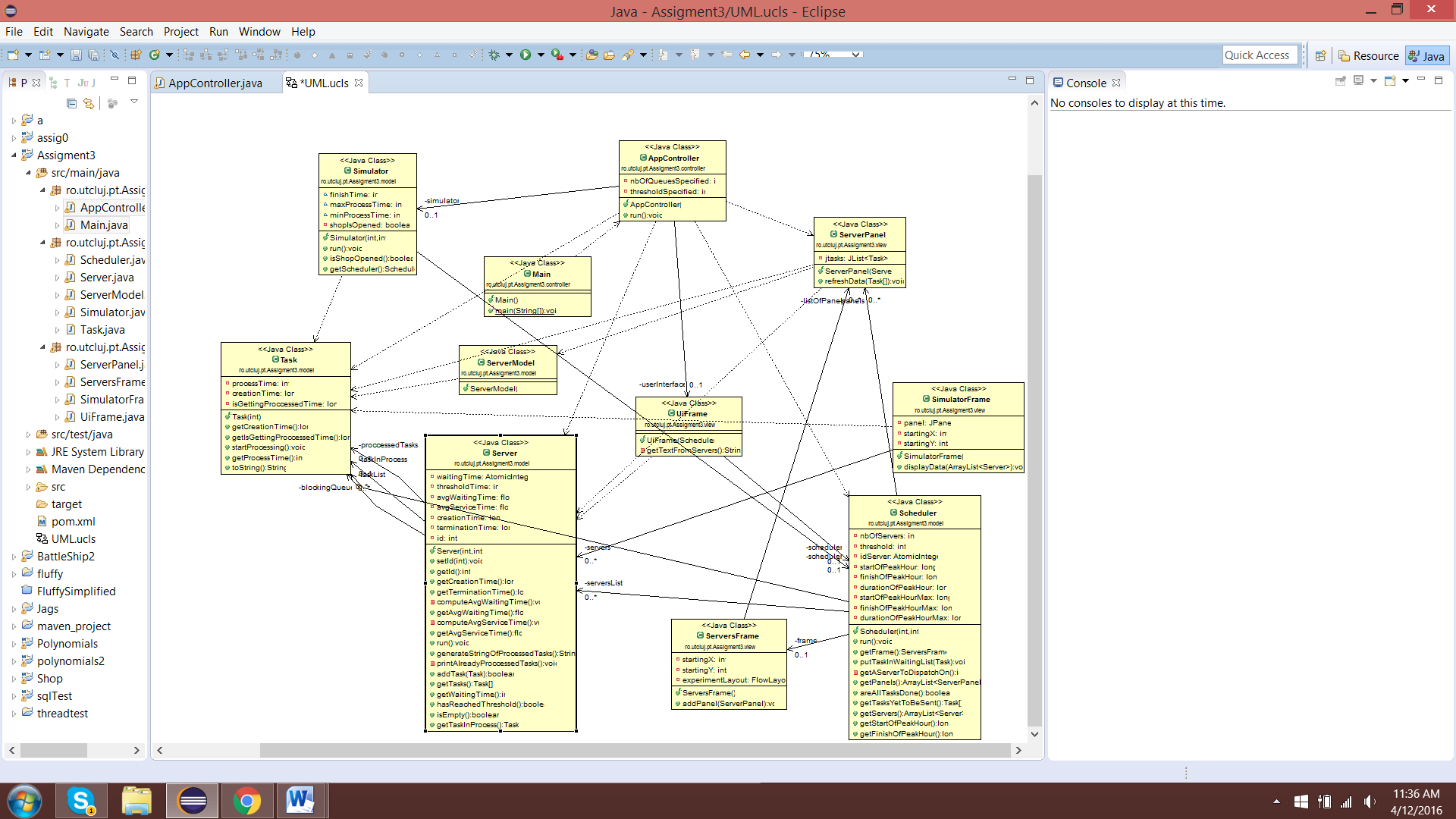
# Design

## Relational Diagram

The entity-relational model is a data model that describes aspects of a business domain or its process requirements, in such an abstract way that it can lead to being implemented in a database such as a relational database. The main components of ER models are entities and the relationships that can exist between them.

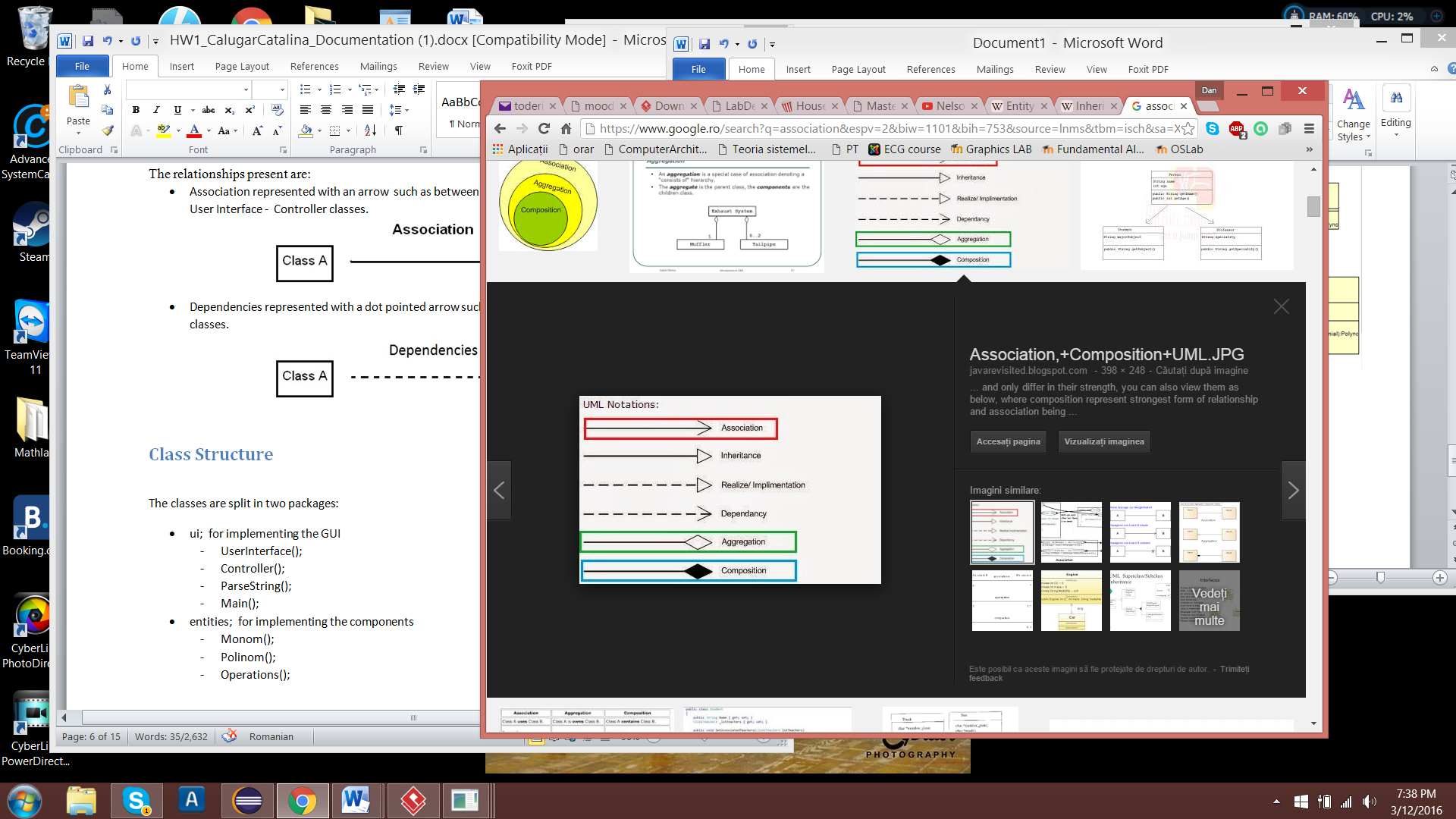
In most cases the ER design is the first step in solving any OOP problem. I have chosen to implement each operation into a separate class and also to add a class for testing. However, testing is needed only for verifying if the persistence classes work properly, because of the dimensions of the project and the number of operations done on the database, we must make sure that those classes are perfect. Also controllers, can have or not a hierarchic structures.

This is the relationship diagram containing associations and dependencies type relations:

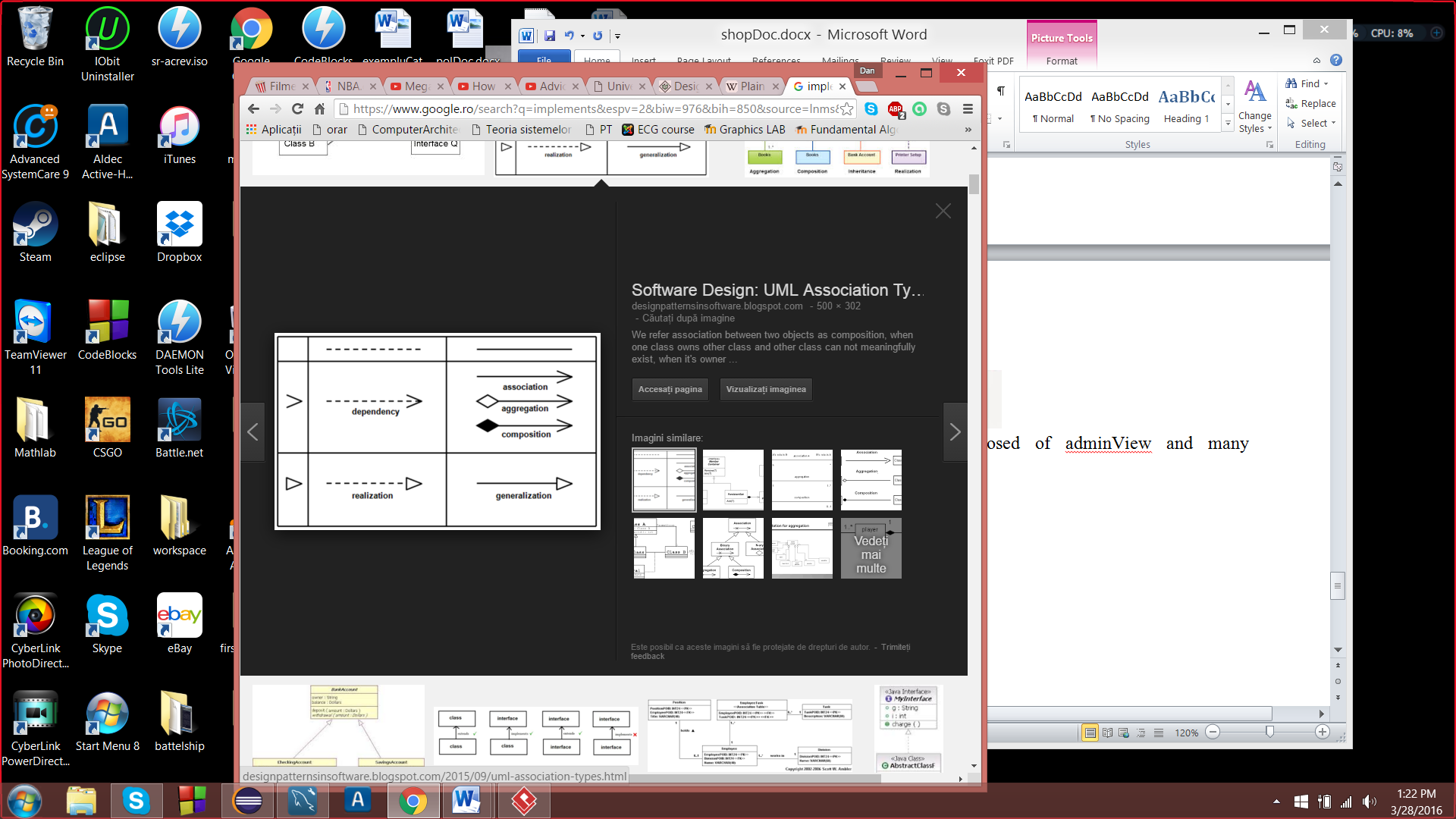


The relationships that are shown in the diagrams are:

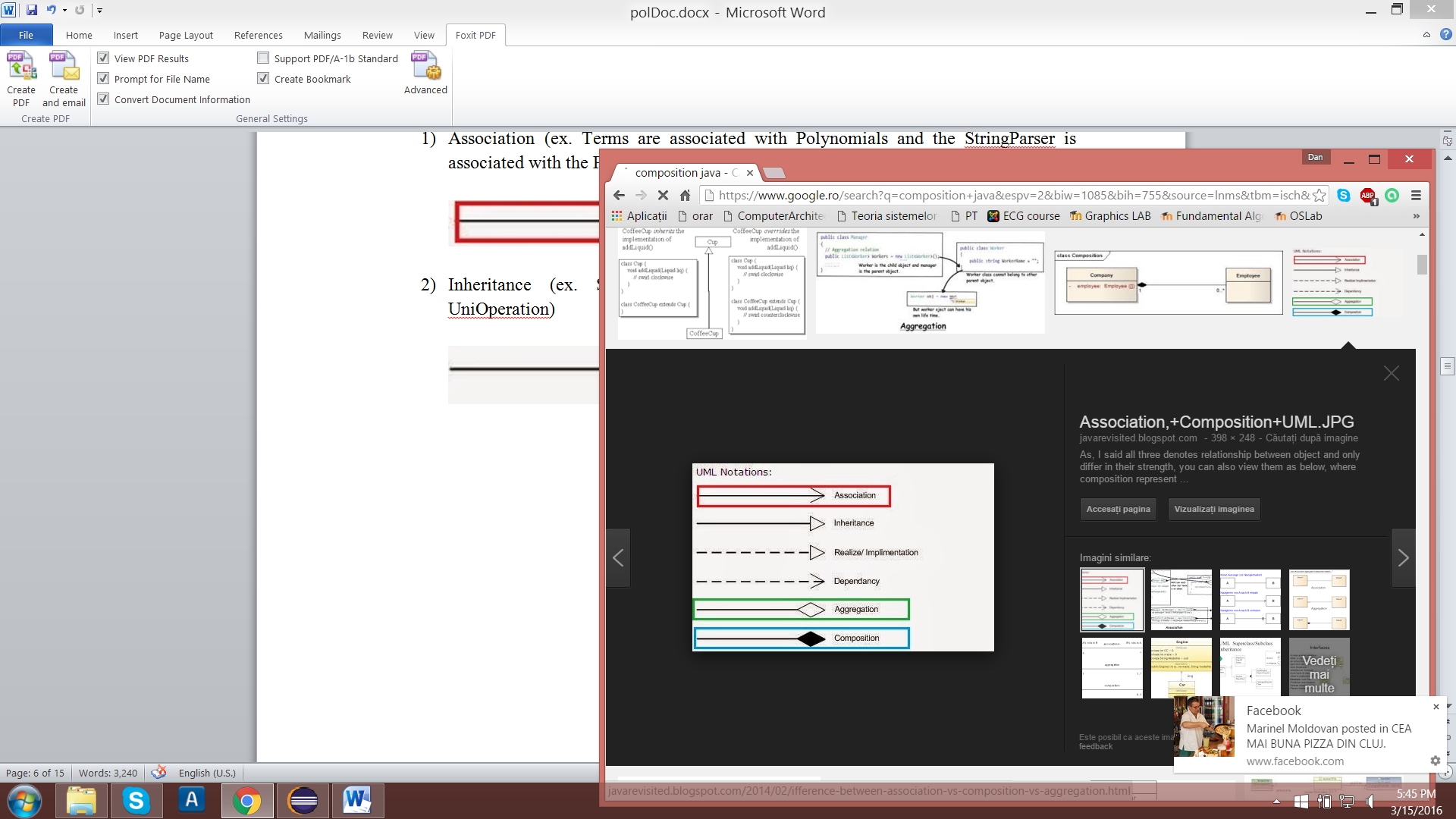
1. Association (ex. Task is associated with the Server).



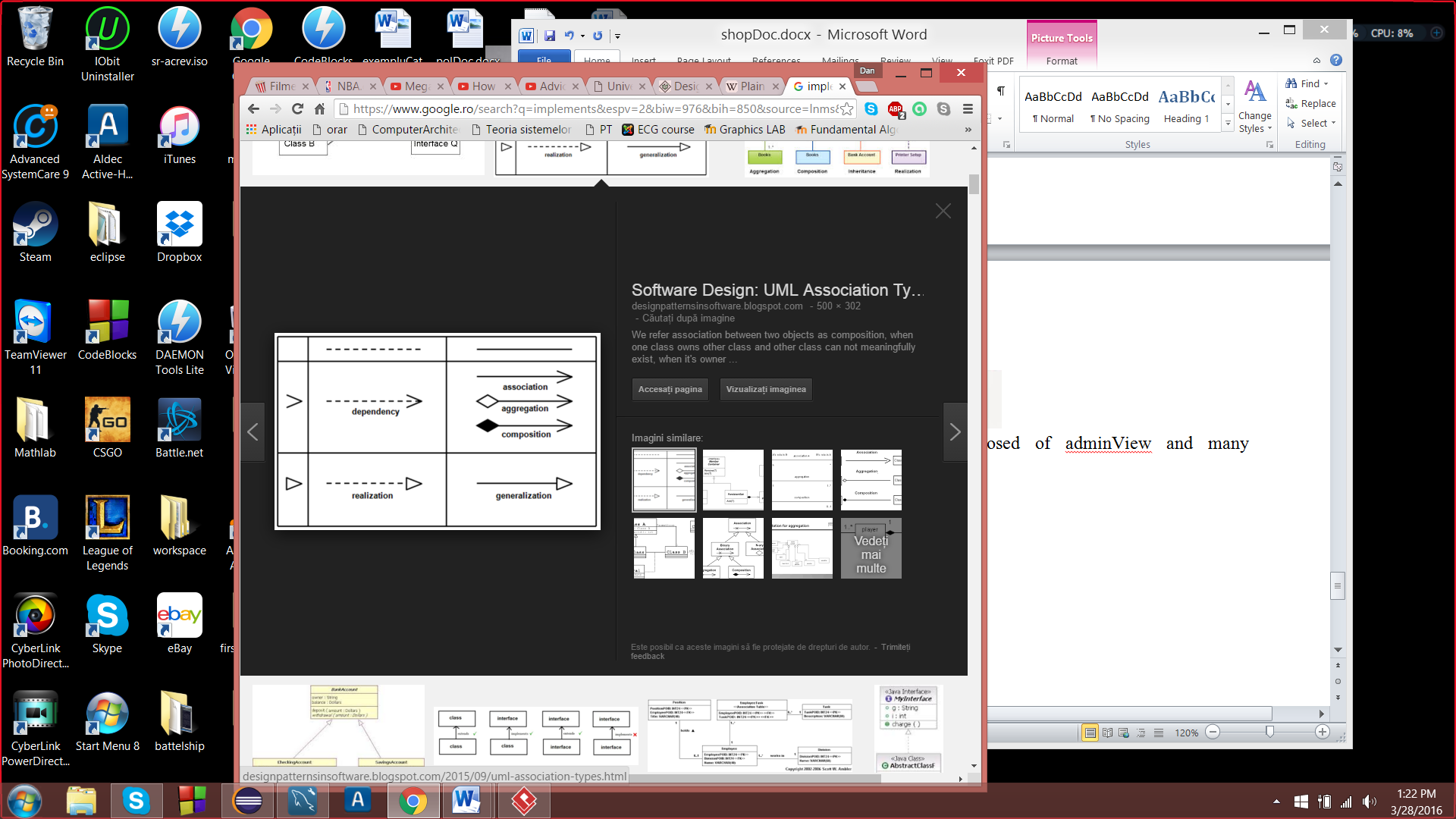
1. Implements (ex. Server and Simulator implements Runnable)



1. Composition (ex. AppControler is composed of Simulator and views).



1. Dependency (ex. ServerPanel classes depends on the ServerModel)



# Class Structure

The classes are split into numerous packages:

1. ro.utcluj.pt.Assigment3.model :

-Server

-Task

-Simulator

-ServerModel

-Scheduler

1. ro.utcluj.pt.Shop.view:

-Server Panel

-Server Frame

-SimultorFrame

-UI Frame

5) ro.utcluj.pt.Shop.controller

-App Controller

-Main

# Implementation

Firstly, let’s start by explaining what the packages main function is and how this packages work together to form this application.

## Model

### Task Class

Has only processTime,creationTime and isGettingProccessedTime. Both properties that used for diplaying info and which are compute by using System.currentTimeMillis() .

### Simulator Class

Simple class, just generates random Task and sends them to the Scheduler.

### Scheduler Class

Has a BlockingQueue where Simulator puts the newly created task and afterwards the Scheduler tries to send it to one of the servers or opens a new server if all servers have reached threshold.

### Server Class

Same problem has to have a BlockingQueue from where it takes its task to be proccessed. The waiting time is proccessed when a task is added to the queue and when it is taken from the queue. Being a Runnable object the main thread of this object fetches data from the queue and computes the avearages.

#### BlockingQueue

The Java BlockingQueue interface in the java.util.concurrent package represents a queue which is thread safe to put into, and take instances from.

A BlockingQueue is typically used to have on thread produce objects, which another thread consumes. Here is a diagram that illustrates this principle:



The producing thread will keep producing new objects and insert them into the queue, until the queue reaches some upper bound on what it can contain. It's limit, in other words. If the blocking queue reaches its upper limit, the producing thread is blocked while trying to insert the new object. It remains blocked until a consuming thread takes an object out of the queue.

The consuming thread keeps taking objects out of the blocking queue, and processes them. If the consuming thread tries to take an object out of an empty queue, the consuming thread is blocked until a producing thread puts an object into the queue.

A BlockingQueue has 4 different sets of methods for inserting, removing and examining the elements in the queue. Each set of methods behaves differently in case the requested operation cannot be carried out immediately. Here is a table of the methods:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Throws Exception** | **Special Value** | **Blocks** | **Times Out** |
| **Insert** | add(o) | offer(o) | put(o) | offer(o, timeout, timeunit) |
| **Remove** | remove(o) | poll() | take() | poll(timeout, timeunit) |
| **Examine** | element() | peek() |  |  |

[**put**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html#put(E))([**E**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html) e)Inserts the specified element into this queue, waiting if necessary for space to become available.

[**take**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html#take())()Retrieves and removes the head of this queue, waiting if necessary until an element becomes available.

However using a synchronized data structure will be absolete if we do not understand the concept behing it, which is multithreading and synchronization of threads.

#### Threads

In computer science, a *thread* of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system. The implementation of threads and processes differs between operating systems, but in most cases a thread is a component of a process. Multiple threads can exist within one process, executing concurrently (one starting before others finish) and share resources such as memory, while different processes do not share these resources. In particular, the threads of a process share its instructions (executable code) and its context (the values of its variables at any given time).

On one processor, multithreading is generally implemented by time slicing (as in multitasking), and the central processing unit (CPU) switches between different *software threads*. This context switching generally happens often enough that users perceive the threads or tasks as running at the same time (in parallel). On a multiprocessor or multi-core system, multiple threads can be executed in parallel (at the same instant), with every processor or core executing a separate thread simultaneously; on a processor or core with *hardware threads*, separate software threads can also be executed concurrently by separate hardware threads.

Threads made an early appearance in OS/360 Multiprogramming with a Variable Number of Tasks (MVT) in 1967, in which they were called "tasks". The term "thread" has been attributed to Victor A. Vyssotsky. Process schedulers of many modern operating systems directly support both time sliced and multiprocessor threading, and the operating system kernel allows programmers to manipulate threads by exposing required functionality through the system call interface. Some threading implementations are called *kernel threads*, whereas *light-weight processes* (LWP) are a specific type of kernel thread that shares the same state and information. Furthermore, programs can have *user-space threads* when threading with timers, signals, or other methods to interrupt their own execution, performing a sort of ad hoc time slicing.

#### Processes vs Threads

A process is an executing instance of an application. What does that mean? Well, for example, when you double-click the Microsoft Word icon, you start a process that runs Word. A thread is a path of execution *within* a process. Also, a process can contain multiple threads. When you start Word, the operating system creates a process and begins executing the primary thread of that process.

It’s important to note that a thread can do anything a process can do. But since a process can consist of multiple threads, a thread could be considered a ‘lightweight’ process. Thus, the essential difference between a thread and a process is the work that each one is used to accomplish. Threads are used for small tasks, whereas processes are used for more ‘heavyweight’ tasks – basically the execution of applications.

Another difference between a thread and a process is that threads within the same process share the same address space, whereas different processes do not. This allows threads to read from and write to the same data structures and variables, and also facilitates communication between threads. Communication between processes – also known as IPC, or inter-process communication – is quite difficult and resource-intensive.

#### Multithreading

Is mainly found in multitasking operating systems. Multithreading is a widespread programming and execution model that allows multiple threads to exist within the context of one process. These threads share the process's resources, but are able to execute independently. The threaded programming model provides developers with a useful abstraction of concurrent execution. Multithreading can also be applied to one process to enable parallel execution on a multiprocessing system.

Multithreaded applications have the following advantages:

* *Responsiveness*: multithreading can allow an application to remain responsive to input. In a one-thread program, if the main execution thread blocks on a long-running task, the entire application can appear to freeze. By moving such long-running tasks to a *worker thread* that runs concurrently with the main execution thread, it is possible for the application to remain responsive to user input while executing tasks in the background. On the other hand, in most cases multithreading is not the only way to keep a program responsive, with non-blocking I/O and/or Unix signals being available for gaining similar results.[[6]](https://en.wikipedia.org/wiki/Thread_(computing)#cite_note-6)
* *Faster execution*: this advantage of a multithreaded program allows it to operate faster on computer systems that have multiple central processing units(CPUs) or one or more multi-core processors, or across a cluster of machines, because the threads of the program naturally lend themselves to parallel execution, assuming sufficient independence (that they do not need to wait for each other).
* *Lower resource consumption*: using threads, an application can serve multiple clients concurrently using fewer resources than it would need when using multiple process copies of itself. For example, the Apache HTTP server uses thread pools: a pool of listener threads for listening to incoming requests, and a pool of server threads for processing those requests.
* *Better system utilization*: as an example, a file system using multiple threads can achieve higher throughput and lower latency since data in a faster medium (such as cache memory) can be retrieved by one thread while another thread retrieves data from a slower medium (such as external storage) with neither thread waiting for the other to finish.
* *Simplified sharing and communication*: unlike processes, which require a message passing or shared memory mechanism to perform inter-process communication (IPC), threads can communicate through data, code and files they already share.
* *Parallelization*: applications looking to use multicore or multi-CPU systems can use multithreading to split data and tasks into parallel subtasks and let the underlying architecture manage how the threads run, either concurrently on one core or in parallel on multiple cores. GPU computing environments like CUDA and OpenCL use the multithreading model where dozens to hundreds of threads run in parallel on a large number of cores.

Multithreading has the following drawbacks:

* *Synchronization*: since threads share the same address space, the programmer must be careful to avoid race conditions and other non-intuitive behaviors. In order for data to be correctly manipulated, threads will often need to rendezvous in time in order to process the data in the correct order. Threads may also require mutually exclusive operations (often implemented using semaphores) in order to prevent common data from being simultaneously modified or read while in the process of being modified. Careless use of such primitives can lead to deadlocks.
* *Thread crashes a process*: an illegal operation performed by a thread crashes the entire process; therefore, one misbehaving thread can disrupt the processing of all the other threads in the application.

#### Models of Multithreading

### 1:1 (kernel-level threading)

Threads created by the user in a 1:1 correspondence with schedulable entities in the kerne are the simplest possible threading implementation. OS/2 and Win32 used this approach from the start, while on Linux the usual C library implements this approach (via the NPTL or older LinuxThreads). This approach is also used by Solaris, NetBSD, FreeBSD, OS X and iOS.

### N:1 (user-level threading)

An N:1 model implies that all application-level threads map to one kernel-level scheduled entity;the kernel has no knowledge of the application threads. With this approach, context switching can be done very quickly and, in addition, it can be implemented even on simple kernels which do not support threading. One of the major drawbacks however is that it cannot benefit from the hardware acceleration on multithreaded processors or multi-processor computers: there is never more than one thread being scheduled at the same time.For example: If one of the threads needs to execute an I/O request, the whole process is blocked and the threading advantage cannot be used. The GNU Portable Threads uses User-level threading, as does State Threads.

### M:N (hybrid threading)

M:N maps some M number of application threads onto some N number of kernel entities,or "virtual processors." This is a compromise between kernel-level ("1:1") and user-level ("N:1") threading. In general, "M:N" threading systems are more complex to implement than either kernel or user threads, because changes to both kernel and user-space code are required. In the M: N implementation, the threading library is responsible for scheduling user threads on the available schedulable entities; this makes context switching of threads very fast, as it avoids system calls. However, this increases complexity and the likelihood of priority inversion, as well as suboptimal scheduling without extensive (and expensive) coordination between the userland scheduler and the kernel scheduler.

## Views

### Server Panel

Must display in real time what tasks are to be processed by a queue, they are not in process stage. The AppController will fetch info about the queue form each server and it is going to be update on the panel.

### Server Frame

Only a container for the Server Panels.

# Conclusion and further developments

Arriving to the last point of this presentation, I personally considered this project a good exercise for many things, such as multithreading, use of Runnable objects, working with time processing and computing a lot of info and data. Thought the development of this project I’ve experienced many problems with views and threads, but it has been all for the best. Now I have the skill to display info in real time and synchronize more than 2 threads.

The only problem I have not tried to actually fix, because I was not required to and I had not much time at hand was to actually close servers.

Some improvements and future development plans would be:

1. Check the formats of the inputted data if they have the right format, using regex statements or simple methods and add some more inputs
2. Creating a more dynamic server management system, like I said to close some when they reach a minimum threshold
3. Have a multi-thread improvement to display more data in the Simulation Frame and switch from the simple design to a more complex one
4. Design a better user interface with much more functionalities and prettier views that have animations and more info
5. More notifications for the user, based upon his or her actions and include more stats to be displayed at the end of the simulation like best waiting time per server, most processed tasks and intervals when each server was at its maximum capacity.

# References

<https://en.wikipedia.org/wiki/Thread_(computing)#Fiber_implementation_examples>

http://www.programmerinterview.com/index.php/operating-systems/thread-vs-process/

<https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html>

https://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html