Question 4: Interface

1. Instruction

1. Create an Eclipse project.
2. Copy all folders in folder "**Mock04_toStudent**" of the given zip file to your project **src** folder.
3. You are required to implement or complete the following classes (details for each class are provided in Section 3.)
   - i. CalculationTask (package **logic.task**)
   - ii. SortTask (package **logic.task**)
   - iii. RenderTask (package **logic.task**)
   - iv. TaskManager (package **logic.task**)
4. You are required to implement the following interfaces (details for each interface are provided in Section 3.)
   - i. Computable (package **interfaces**)
   - ii. Modifiable (package **interfaces**)
   - iii. Duplicatable (package **interfaces**)
   - iv. Parallelizable (package **interfaces**)
5. You also need to create a UML Diagram containing all classes and interfaces in the **interfaces, logic.task** package. Save it as **umlQ4.png** at the **root** folder of your project.
6. **You must fix all generated incorrect class diagrams manually (using Paint).**
7. JUnit for testing is in the package "grader".
8. Export jar file (**q4.jar**) that includes your source code and your .class files and put it in **root** folder of your project. YOU MUST EXPORT JAR FILE completely with both source code and .class files. And submit it on Mycourseville. IF YOU DO NOT, YOUR CODE WILL NOT BE MARKED.

You should read the whole document before starting code. You may need to use some methods from the classes already provided.

2. Problem Statement: Task Time Estimator

In *Task Time Estimator*, we aim to demonstrate how to estimate time used to process each task. There are 2 main processor units and 5 tasks category to be used in this task.

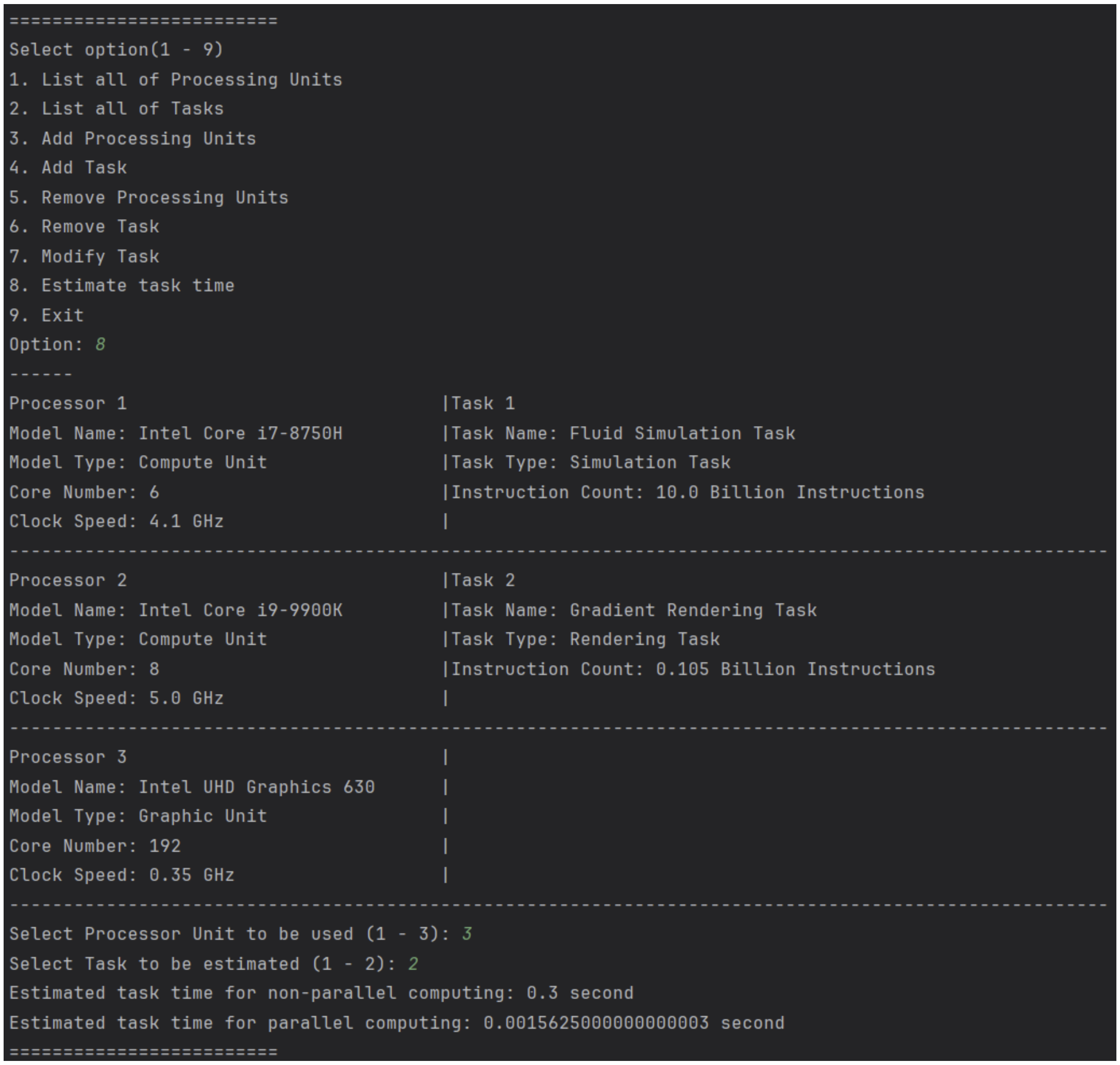
For the processor type, there are

- Compute Unit: Process Unit which can compute anything that can be computed at the cost of low computation power.
- Graphic Unit: Process Unit which can greatly accelerate the static task (task that can be modified) at the cost of task type limitations.

For the task type, there are

- Simple Task: Task that finishes instantly in any process unit.
- Calculation Task: Task mainly used for simple but frequently occur and can be modified in real time.
- Sorting Task: Task mainly used for ordering something to optimize system resources. Sometimes, sorting task can be merged to remove redundancy.
- Rendering Task: Task mainly used for rendering images or 3d shapes. The task can be duplicated to be used together with another processing unit.

Figure 1 shows a working program.



```
=========================
Select option(1 - 9)
1. List all of Processing Units
2. List all of Tasks
3. Add Processing Units
4. Add Task
5. Remove Processing Units
6. Remove Task
7. Modify Task
8. Estimate task time
9. Exit
Option: 8
------
Processor 1                             |Task 1
Model Name: Intel Core i7-8750H         |Task Name: Fluid Simulation Task
Model Type: Compute Unit                |Task Type: Simulation Task
Core Number: 6                          |Instruction Count: 10.0 Billion Instructions
Clock Speed: 4.1 GHz                    |
-----------------------------------------------------------------------------------------------------
Processor 2                             |Task 2
Model Name: Intel Core i9-9900K         |Task Name: Gradient Rendering Task
Model Type: Compute Unit                |Task Type: Rendering Task
Core Number: 8                          |Instruction Count: 0.105 Billion Instructions
Clock Speed: 5.0 GHz                    |
-----------------------------------------------------------------------------------------------------
Processor 3                             |
Model Name: Intel UHD Graphics 630      |
Model Type: Graphic Unit                |
Core Number: 192                        |
Clock Speed: 0.35 GHz                   |
-----------------------------------------------------------------------------------------------------
Select Processor Unit to be used (1 - 3): 3
Select Task to be estimated (1 - 2): 2
Estimated task time for non-parallel computing: 0.3 second
Estimated task time for parallel computing: 0.0015625000000000003 second
=========================
```

Figure 1: Program Demonstration.

3. Implementation Details

To complete this question, you need to understand about **Interface**. The class packages are summarized below. Only the key methods you need to implement or complete for this question are listed. Noted that access modifier notations are listed below

+ (public)

# (protected)

- (private)

Underline (static)

Italic (abstract)

3.1 Package application /*Already Provided*/

3.1.1 Class Main /*Already Provided*/

This class is a main program that controls how the program works. You can run Main.java to see program results after implementing all classes and interfaces.

3.2 Package interfaces */* you must implement from scratch */*

3.2.1 Interface Computable */* you must implement from scratch */*

This defines an object that can compute process time for its task.

Methods

Method	Description
+ double compute(ProcessUnit processUnit)	Method to compute process time for task.

3.2.2 Interface Modifiable *<u>/* you must implement from scratch */</u>*

This defines an object that can modify its task.

Methods

Method	Description
+ void modify(int modifyValue)	Method to modify task.

3.2.3 Interface Duplicatable */* you must implement from scratch */*

This defines an object that can duplicate its task.

Methods

Method	Description
+ void duplicateTask(int taskNumber)	Method to duplicate task.

3.2.4 Interface Parallelizable */* you must implement from scratch */*

This defines an object that can parallel compute its process time for its task.

Methods

Method	Description
+ double parallelCompute(ProcessUnit processUnit)	Method to parallel compute process time for task.

3.3 Package logic /* Partially Provided*/

3.3.1 Package logic.compute /*Already Provided*/

3.3.1.1 Class ProcessUnit /*Already Provided*/

This class represents the Process Unit, it can't be directly instantiated.

Fields

Field	Description
- String modelName	Model name for this unit.
- int coreNumber	Number of cores to be used for parallel computation.
- double clockSpeed	Clock speed in GHz used for all computation.

Methods

Method	Description
+ ProcessUnit(String modelName, int coreNumber, double clockSpeed)	Constructor of process unit class. Set this modelName to modelName. Set this coreNumber to coreNumber. Set this clockSpeed to clockSpeed.
+ *boolean canCompute(Task task)*	Method to check whether task can be computed
+ double compute(Task task)	If can compute task, return task workload/clockSpeed. Else if clockSpeed > 0, return 0. Else, return -1.
+ double parallelCompute (Task task)	If can compute task and can parallelize, return task workload/(clockSpeed*coreNumber). Else, call compute method.

+ String getModelType()	If processor unit is Compute Unit, return "Compute Unit". Else if processor unit is Graphic Unit, return "Graphic Unit". Else, return "General Unit".
+ getter/setter	

3.3.1.2 Class ComputeUnit */*Already Provided*/*

This class represents Compute Unit which has less core number but higher clock speed

Methods

Method	Description
+ ComputeUnit(String modelName, int coreNumber, double clockSpeed)	Constructor of compute unit class. Call super(modelName, coreNumber, clockSpeed).
+ boolean canCompute(Task task)	Return true if clockSpeed > 0 and task can be computed, otherwise return false.

3.3.1.3 Class GraphicUnit */*Already Provided*/*

This class represents Graphic Unit which has much higher core number but lower clock speed

Methods

Method	Description
+ GraphicUnit(String modelName, int coreNumber, double clockSpeed)	Constructor of graphic unit class. Call super(modelName, coreNumber, clockSpeed).
+ boolean canCompute(Task task)	Return true if clockSpeed > 0 and task can be computed and can't be modified, otherwise return false.

3.3.2 Package logic.task */*Partially Provided*/*

3.3.2.1 Class Task */*Already Provided*/*

This class represents a task which can be used to estimate time, it can’t be directly instantiated.

Fields

Field	Description
- String name	Name of task.
- double workload	Number of instructions count for the task (billion instructions).

Methods

Method	Description
+ Task(String name, double workload)	Constructor of Task class.
+ *String fullTaskName()*	Method to call for task category name.
+ String toString()	return name + " " + fullTaskName()
+ void setWorkload(double workload)	Set task workload to workload. If less than 0, set it to 0.
+ getter/setter	

3.3.2.2 Class TaskList */*Already Provided*/*

This class represents a list of all tasks which can be interacted with.

Fields

Field	Description
- ArrayList<Task> tasks	Array List of all tasks

Methods

Method	Description
+ Task getTask(int index)	Return task index in array list.
+ ArrayList<Task> getTasks()	Return all tasks.
+ void setTasks (ArrayList<Task> ts)	Set array list tasks to a given ts.
+ void addTasks(Task task)	Add task to tasks.
+ void removeTasks(int index)	Remove task at position index in tasks array list.

3.3.2.3 Class SimpleTask */*Already Provided*/*

This class represents a simple task that finishes instantly when computed.

Methods

Method	Description
+ SimpleTask(String name, double workload)	Constructor of Simple Task class. Call super(name, workload)
+ String fullTaskName()	Return "Simple Task".

3.3.2.4 Class CalculationTask */*you must implement from scratch*/*

This class represents a task that can be computed and can be modified.

Methods

Method	Description
+ CalculationTask(String name, double workload)	Constructor of Calculation Task class. Call super(name, workload)

+ double compute(ProcessUnit processUnit)	Return the result from using processUnit to compute **this** task.
+ void changeWorkload(int **workload**)	Set task's workload to **workload** + task's workload
+ void modify(int workload)	Call changeWorkload method.
+ String fullTaskName()	return "Calculation Task".

3.3.2.5 Class SortTask */*you must implement from the scratch*/*

This class represents a task that can be computed, can be modified, and can be parallelized.

Methods

Method	Description
+ SortTask(String name, double workload)	Constructor of Sort Task class. Call super(name, workload)
+ double compute(ProcessUnit processUnit)	Return the result from using processUnit to compute this task.
+ double parallelCompute (ProcessUnit processUnit)	Return the result from using processUnit to compute this task by utilizing parallelization.
+ void mergeTask(int index)	Create variable selectTask, then set it to TaskList.getTask(index) If selectTask is SortTask class - Set the workload as the sum of workload of this task and selectTask's workload. - Remove task index from TaskList Else, do nothing.
+ void modify(int workload)	Call mergeTask method.

+ String fullTaskName()	return "Sorting Task".

3.3.2.6 Class RenderTask */*you must implement from scratch*/*

This class represents a task that can be computed, can be parallelized, and can be duplicated.

Methods

Method	Description
+ RenderTask(String name, double workload)	Constructor of Render Task class. Call super(name, workload)
+ double compute(ProcessUnit processUnit)	Return the result from using processUnit to compute this task.
+ double parallelCompute (ProcessUnit processUnit)	Return the result from using processUnit to compute this task by utilizing parallelization.
+ void duplicateTask(int taskNumber)	Add a RenderTask object **taskNumber** number of times to TaskList. Each RenderTask object has name = this task name + "-" + i , where i is from **1** to **taskNumber**. Each RenderTask object has its workload equals to **this**'s workload.
+ String fullTaskName()	return "Rendering Task".

3.3.2.7 Class TaskManager */*Partially Provided*/*

This class represent methods to filter tasks with specific type and manipulate tasks.

Methods

Method	Description
+ boolean instanceOf (Class checkClass, Class interfaceClass)	Method for checking whether interfaceClass is an interface or subclass of checkClass The result is **equivalent to checkClass instanceof interfaceClass** but Class is not predetermined (can be variable type).
+ ArrayList<Task> getTaskByType (ArrayList <Class> types)	// FILL CODE First, get arraylist, name it **tasks**, from **TaskList**. Then, let **returnTasks** = new ArrayList<Task>() For each **task** in **tasks** - If the **task's class** is the **subclass or interface** of any Class in **types,** append **task** to **returnTasks.** Return **returnTasks.** **Note.** Class is the data type for storing details of object which can be instantiated by using these syntaxes - Class test = className.class - Class test = interfaceName.class You can get the class of an object by calling **getClass()** - Class c = data.getClass()
+ void deleteDuplicateTasks()	// FILL CODE Get all **tasks** from taskList into arraylist **tasks**. For each **task** in **tasks** - If **task** can be duplicated and contain "-" in the toString(), remove **task** from **tasks**

4. Scoring Criteria

The scoring criteria is listed below. There are scores for JUnit files as well as the UML Diagram file. The total score of this section is 50, which will be factored to a net score of 10.

1. **CalculationTaskTest** **6 points**
   a. testConstructor 1 point
   b. testChangeWorkload 1 point
   c. testModify 2 points
   d. testCompute 2 points
2. **SortTaskTest** **8 points**
   a. testConstructor 1 point
   b. testMergeTask 1 point
   c. testModify 2 points
   d. testCompute 2 points
   e. testParallelCompute 2 points
3. **RenderTaskTest** **9 points**
   a. testConstructor 1 point
   b. testDuplicateTask 4 points
   c. testCompute 2 points
   d. testParallelCompute 2 points
4. **TaskManagerTest** **13 points**
   a. testGetTaskByType 3 point
   b. testGetTaskByMultipleTypes 5 point
   c. testDeleteDuplicateTasks 5 point
5. **UML Diagram containing all classes and interfaces in the interfaces, logic.task package.** Save the UML Diagram as **umlQ1.png** in the **root** folder of your project. **14 points**
   a. All classes are present with correct variables and methods 4 points

b. All access modifiers are correct 2 points

c. Inheritance/interface structures are correct 8 points

Export jar (**q4.jar**) file that **includes source code and .class files** and put it in **root** folder of your project. YOU MUST EXPORT JAR FILE correctly with all contents inside, and submit the jar file on MyCourseville. IF YOU DO NOT, YOUR CODE WILL NOT BE MARKED.