Task Executor Library Project

Summer 19 Due: Wednesday 7/9 @ 12 PM

Installing Java and Eclipse IDE

This document and the following video assumes that you have both Java runtime and the Eclipse IDE installed on your PC. If this is not the case the following link is to a video that does a nice job of explaining the process. There are many other videos and other resources to be found on the web. Let me know if you need help.

The Java / Eclipse Video: https://youtu.be/70dN5jqumAs

Project YouTube Video

A YouTube video has been provided that demonstrates the installation of the development project and testing project in an Eclipse workspace. The video also demonstrates the execution of the testing project and how to export the development project as **a jar file** to be submitted for grading.

The video URL is: https://youtu.be/DEvkzMk4BNA

Project Description

This project will produce library JAR file containing a TaskExecutor service. See the video and the section "Packaging the Library JAR File" for more details.

TaskExecutor is a service class that maintains a pool (collection) of N threads (See Thread Pool Design Pattern) that are used to execute instances of Tasks provided by the TaskExecutor's clients. Interfaces for both the TaskExecutor and Task have been provided and must be used to implement the service (including the package structure).

You have also been provided a driver application and Task implementation (TaskExecutorTest.java and SimpleTestTask.java) that you will need to use to design, debug, and test your submissions.

Project Goals

The **primary goal** of this project is to have teams implement the multithreaded synchronization needed to implement the TaskExecutor service.

The **secondary goal** of the project (but the most difficult) is to implement a Finite Bounded FIFO Buffer described in the textbook, and in class, using only Java's Object as monitors and **without the use of synchronized methods**. Note Synchronized blocks are acceptable, even needed to implement the bounded buffer.

Goal1: TaskExecutor Service

The TaskExecutor is a service that accepts instances of Tasks (classes implementing the Task interface) and executes each task in one of the multiple threads maintained by the thread pool. That is, the service maintains a pool of **pre-spawned threads** that are used to execute Tasks.

Figure 1 provides an overview of the structure and design of the TaskExecutor service. Clients provide implementations of the Task interface which performs some application-specific operation. Clients utilize the TaskExecutor.addTask() method to add these tasks to the Blocking FIFO queue. Pooled threads remove the tasks from the queue and call the Task's execute() method. This Task executes for some amount of time before completing by returning from the execute() method. At this point the task-running threads attempts to obtain a new Task from the queue. If the blocking queue is empty (no tasks to execute) the task running thread's execution must be blocked until a new task is added to the task queue. If the blocking queue is full (no space to add a new task) the client's thread's execution must be blocked until a task is been removed from the queue.

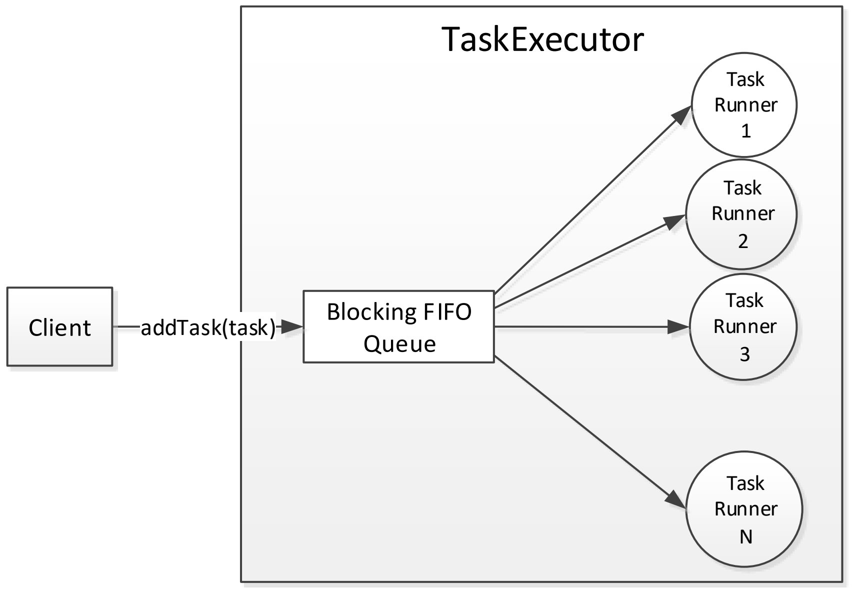


Figure 1: Task Executor Overview

Goal 2: Implementing the Blocking Queue

Teams are to provide an implementation of a Blocking Queue. This is a FIFO queue that is both thread-safe and blocking. Because the queue is internal to the TaskExecutor, the project does not specify the queue's interface. However, for the sake of discussion let us use the following interface as an example:

```
public interface BlockingFIFO
{
   void put(Task item) throws Exception;
   Task take() throws Exception;
```

}

By ‘Blocking’ we mean that…

1. The put(task) method places the given task into the queue. If the queue is full, the put() method must blocking the client’s thread until space is made available i.e. when a Task is removed from the queue through the take() method.
2. The take() method removes and returns a task from the queue. If the queue is empty, the pooled thread calling take() will block until a Task has been placed in the queue though the put() method.

As described in the Grading Section, teams have two options when implementing the Blocking Queue.

1. Teams can initially use the class `java.util.concurrent.ArrayBlockingQueue` which is provided by the Java Development Kit (JDK) runtime library. ArrayBlockingQueue implements the needed blocking behavior as described above. But the use of ArrayBlockingQueue provides less than full credit for the project.
2. Teams implement their own BlockingFIFO queue. The three restrictions are 1) teams must implement their BlockingFIFO using the boundedbuffer design as provided in the book (and later in this document) 2) the BlockingFIFO must implement using Java Objects as monitors (Not Semaphores) as described in this document and, 3) implementations must be based on using an array of Task as its container. That is, the size of the queue must be fixed when the container is created. The FIFO implementation size (array length) must be no more than 100 elements. Implementations cannot use any of Java’s built-in container classes (e.g. ArrayList) which has its built-in synchronization.

Project Requirements

Be sure to read and understand these project requirements. Your submissions will be held accountable for them.

1. **The BlockingFIFO must implement the design given for boundedbuffer as described in the text book, and later in this document (See the section BlockingFIFO Implementation Notes). Any other FIFO design will not receive credit for the implementation.**
2. **The BlockingFIFO must implement its synchronization using Java Objects as monitors. You cannot use Semaphores, Locks, or other synchronization mechanisms offered by Java.**
3. **The classes implementing the TaskExecutor, BlockingFIFO, Thread Pool cannot contain synchronized methods. All of the synchronization must be implemented using primitive Object monitors. Also synchronized block will be needed.**
4. The project will provide an implementation of the provided TaskExecutor interface.
5. The TaskExecutor will accept and execute implementations of the provided Task interface.

6. The TaskExecutor and Task interface must implement the interface definitions given in the student development project source files including package and exact method signatures i.e. no changes to the interfaces are allowed.
7. You have been provided with the source for the test program TaskExecutorTest and the task SimpleTestTask both of which you can use to exercise and verify your TaskExecutor implementations. However, you must not modify either of these test classes. ***Grading will be evaluated using unmodified versions of these test classes***.
8. Each thread will be assigned a unique name when created. See Thread.setName(String).
9. Threads will be maintained in a pool and reused to execute multiple Tasks. Threads are not to be created and destroyed for each task's execution.
10. Exceptions thrown during Task execution cannot cause the failure of the executing thread. It is suggested that Task execution be wrapped in a try / catch block that logs and ignores the caught exceptions.
11. Tasks will execute concurrently on **N threads** where N is **the thread pool size** and is provided as a service initialization parameter.
12. The BlockingFIFO implementation must use an Array to store tasks. You cannot use synchronized data structures such as ArrayList to implement your blocking queue.
13. *The* BlockingFIFO *implementation must use an array of length no more than 100 elements.* The reason for this requirement is that an array larger that the number of tasks injected during testing will not exercise the blocking nature of FIFO put() operations.
14. When the number of inserted tasks exceeds the number of threads, unexecuted tasks will remain on the BlockingFIFO until removed by a task running thread.
15. Every pooled thread's execution must block when the BlockingFIFO is empty i.e. Pool threads should not spin or busy-wait when attempting to obtain a task from the service's empty FIFO.
16. When the BlockingFIFO is full, client threads attempting to add a new task to the queue must block until a Task is removed. Again, no polling, or busy waiting is allowed.
17. **The project will be delivered as a library jar file which will be linked with the test applications used to initialize and test the TaskExecutor's correct implementation.**
18. The TaskExecutor should catch, report (log), and eat any exceptions thrown during an application-specific Task's execution i.e. an **Exception thrown by a Task should not cause a pool'ed thread to exit**.
19. **Your implementation cannot print any messages to stdout. The number of lines printed to the console will be used to determine the correctness of your implementation and any additional lines of text will throw off the count and will cause your project to fail.**
20. It is entirely correct for the test application to not exit once the N tasks have been processed. I will leave it to students to figure out the reason why ☺.

Packaging the Library JAR File

Your implementation of the TaskExecutor and the interfaces will be packaged and delivered in a library JAR file. The implantation will be evaluated by executing a prewritten test application using the provided library jar. The TaskExecutor and Task interfaces must be delivered in the package specified by the given source files. Instructions for exporting your project into a library JAR file has been provided at the end of this document.

Student Testing

Teams have been provided a sample application and Task implementation (SimpleTestTask.java and TaskExecutorTest.java) that they can use to test their TaskExecutor implementation. This code can be used to test your implementation. Remember that eventually your team needs to execute the test application / task against the library jar file that will be submitted for grading. In Eclipse, this means generating the jar in a development project and executing the test application using the imported JAR on its classpath.

Instructor Testing

Team will submit for grading a library JAR file containing their implementation of TaskExecutor interface. The library jar will be installed in a project containing a test application SimpleTestTask.java and TaskExecutorTest.java. It is expected that 1) the test program compile without any modification 2) that the program TaskExecutorTest.java produces the correct result from the execution of Task implementations that are part of the testing procedure.

NOTE: As explained in class, one success criteria will be counting the number of output lines generated by TaskExecutorTest.java. Your implementation can not modify either SimpleTestTask.java or TaskExecutorTest.java. Your implementation submited for grading cannot produce any console output. You can (and should) use console output for debugging, but be sure that the additional output is commented out or removed from your code before compiling and packaging your library JAR file for submission.

Application Output

The output generated by the TestExecutorTest application is useful for diagnosing the correct operation of the TaskExecutor's implementation. The project materials provided include a file "sampleOutput.txt" which illustrates the following points:

1. Initially there will be N+M task injection messages printed to the output where N is the size of the FIFO and M is the number of threads.
2. The output will then vary from messages produced by executing threads and messages produced by the injection of new tasks into the queue.
3. Tasks names (e.g. SimpleTask234) will be printed in non-sequential order indicating that the tasks are being schedule out of insertion order (as should be expected).
4. Thread names (e.g. TaskThread8) should also be listed in a random order.
5. The numberOfActivations counter should be equal the number of tasks at the end of the run.
6. The number of the lines in the file should equal 2 times the number of tasks.

Provided Materials

Teams have been provided the following materials on eLearning.

1. Task Executor Project Description.docx: This document.
2. sampleOutput.txt: This is a sample of the correct output generated from the testing application and task provided in the project taskExecutorStudentTestingProject.
3. Task Executor Evaluation – Team XX.docx: This document is to be completed by teams and turned in with the other project deliverables.
4. taskExecutorStudentDevProject.zip: This is an Eclipse project that can be imported into an Eclipse workspace. It serves as the foundation / starting point for the team's development efforts. See the section "Importing a Project from a Zip File" for instructions on importing projects into your Eclipse workspace.
5. taskExecutorStudentTestingProject.zip: This is an Eclipse project that can be imported into an Eclipse workspace. It contains the testing code that will be executed against team's implementations. See the section "Importing a Project from a Zip File" for instructions on importing projects into your Eclipse workspace.

Materials to be Submitted using a UTD Box Folder

The following items must be placed in a UTD Box Folder used to submit your projects. See https://utdallas.account.box.com/login .

After you create the Box folder, you must share the folder with mgc013000@utdallas.edu (me!) and make my account an Editor so I can both read from, and place files into your project folder.

Once the folder is created, place these materials into it for grading…

1. The completed evaluation document "Task Executor Evaluation - Team XX" with the XX replaced by the team's number and the names and NetIDs of team members.
2. All project source code. The simplest approach is to zip and submit your Eclipse project taskExecutorStudentTestingProject.
3. A library JAR containing the Task and TaskExecutor interfaces, their TaskExecutor implementation, and any additional classes needed to support their implementation. The library JAR must compile and execute against an unmodified SimpleTestTask.java and TaskExecutorTest.java as provided including maintaining the package structure.

Grading Criteria

The following is the criteria and percentages used to assign the project's grade.

1. The test program produces the correct results but uses java.util.concurrent.BlockingQueue to implement the BlockingFifoQueue: **60 pts**.
2. The test program produces the correct results and the team implements their own BlockingFifoQueue as described in this document. : **40 pts**.

Class Interfaces

The following are the interfaces provided in the source files TaskExecutor.java and Task.java. These interfaces must be implemented and the code must compile and execute against the test code provided under the eLearning folder TestingSource (SimpleTestTask.java and TaskExecutorTest.java).

```java
public interface Task
{
    void execute();
    String getName();
}

public interface TaskExecutor
{
    void addTask(Task task);
}
```

BlockingFIFO Implementation Notes

Our text book provides a description of Bounded Buffer on page 229 and shown below as Java pseudo code which must serve as a start of your team's BlockingFIFO's design. Note that this design utilizes monitors to implement thread blocking.

```
// Producer Consumer Example from Bounded Buffer on page 229
// translated to Java-like pseudo code.

Task buffer[N];
int nextin, nextout;
int count;
Object notfull, notempty; // Monitors used for synchronization

void put(Task task)
{
  if(count == N) notfull.wait();   // Buffer is full, wait for take

  synchronized(this) {
    buffer[nextin] = X;
    nextin = (nextin + 1) % N;
    count++;
    notempty.notify(); // Signal waiting take threads
  }
}

Task take()
{
  if(count == 0) notempty.wait(); // Buffer is empty, wait for put
  synchronized(this) {
    Task result = buffer[nextout];
    nextout = (nextout + 1) % N;
    count--;
    notfull.notify(); // Signal waiting put threads
    return result;
  }
}
```

Important Note

Be aware that this design contains a race condition that needs to be addressed in your BlockingFIFO implementation. The race condition exists between threads that are unblocked by the notEmpty / notFull monitors and the other concurrently executing threads.

This is an example of the race condition / problem that exist with the design shown above:

1. Thread 1 enters append(), finds the queue full (count == N), and waits on the monitor notFull.
2. Thread 2 enters take(), removes an item from the queue, and signals (notify) monitor notFull.
3. At this point Thread 1 is unblocked and eligible for scheduling. However, this does not mean that Thread 1 immediately begins execution. At this point the unblocked thread is "ready" to be scheduled for execution.
4. Before Thread 1 is scheduled for execution, **Thread 3** executes append(), finds count < N, and adds an item to the queue. Now count = N.
5. **Notice that when Thread 1 resumes execution, it does not re-check if count is still < N (which it won't be because of Thread 3's action).**

It is up to your team to create a solution, or an alternative design, that avoids this race condition while still retaining the BoundedBuffer design outlined in the text. As described in class, a synchronized block on 'this' is needed to allow the put / take methods to manipulate the array data structure without a race condition.

TaskRunner Design

A suggested design for the threads responsible for executing Tasks is to create a Runnable that 1) obtains a Task from the FIFO 2) executes the TASK by calling the execute() method (See Figure 4). The following provides an example of the TaskRunner's implication of its run() method.

```
public void run() {
    while(true) {
        // take() blocks if queue is empty
        Task newTask = blockingFifoQueue.take();
        try {
            newTask.execute();
        }
        catch(Throwable th) {
           // Log (e.g. print exception's message to console)
           // and drop any exceptions thrown by the task's
           // execution.
        }
    }
}
```

Note that the execution of the task is wrapped in an exception handler that will consume any Throwable generated during the execution of the Task's execute method. This is needed to keep the Throwable (exception) from killing the TaskRunner's thread.

The following is a sample design of this service. The interfaces provided by a Java library or by the project are marked in blue. The Task implementation is also marked in blue. The remaining classes compose a suggested design implementing the project's requirements.

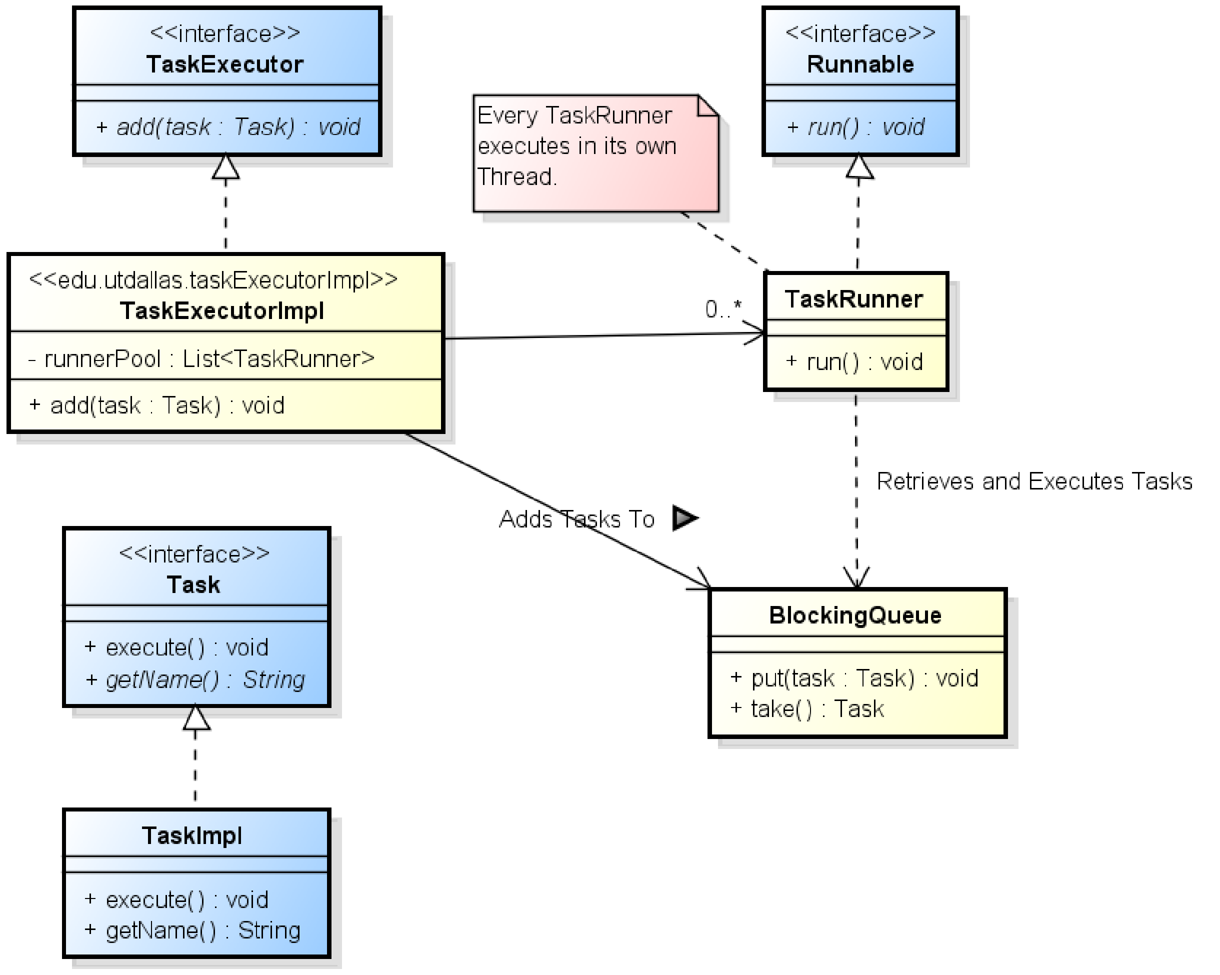


Figure 2: Suggested Design

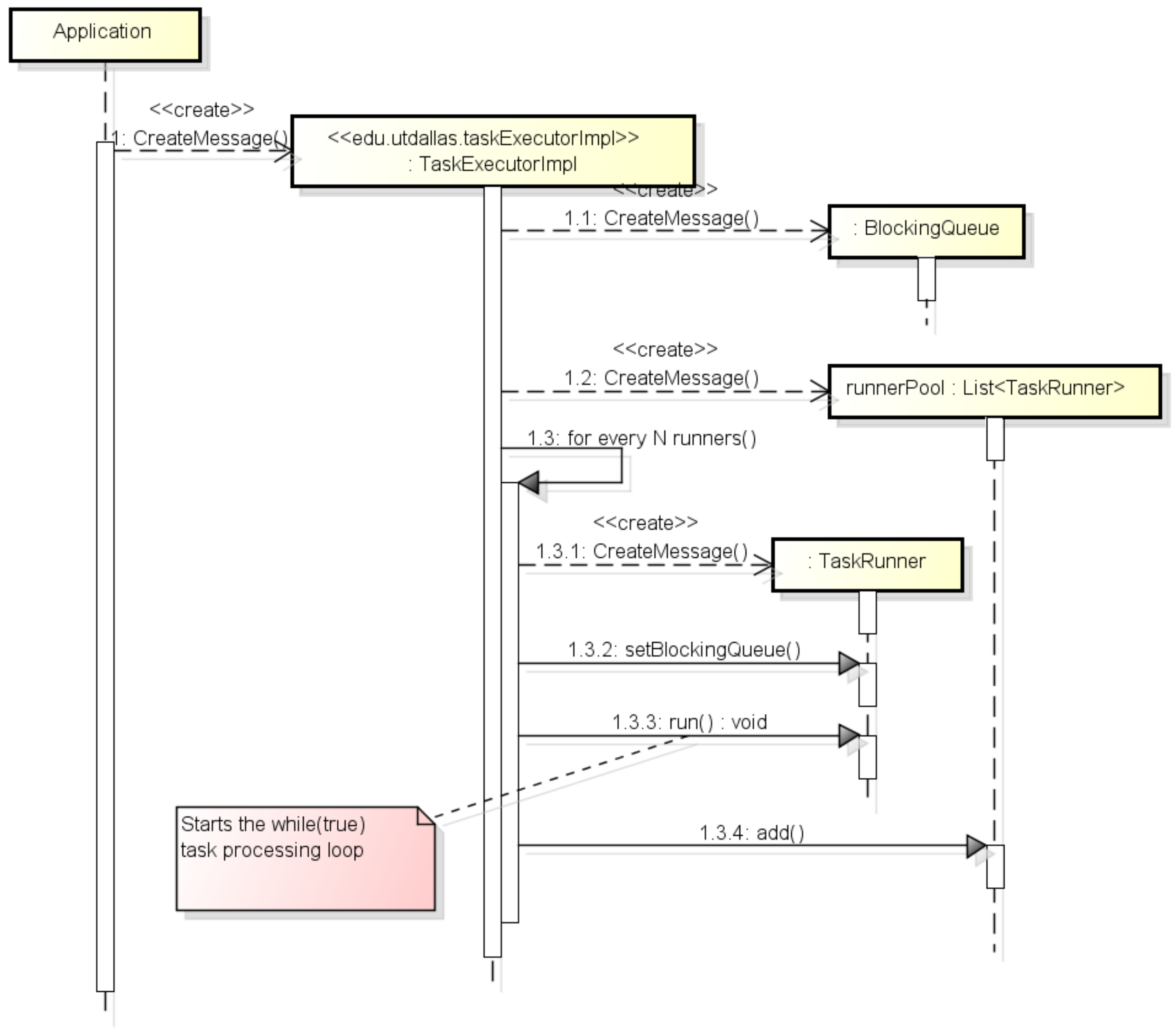


Figure 3: Suggested TaskExecutor Initialization

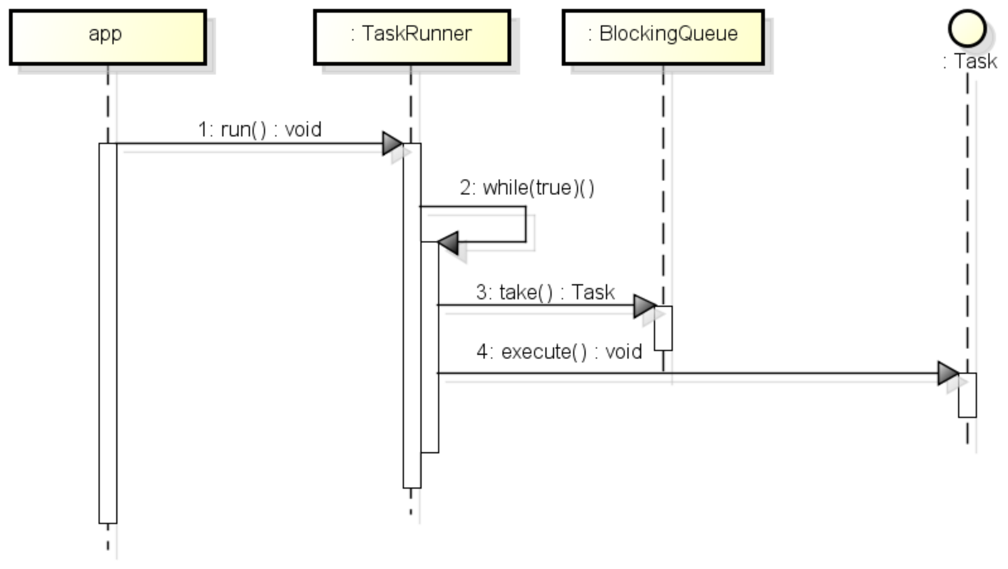


Figure 4: Suggested Task Runner Design

Working With Eclipse

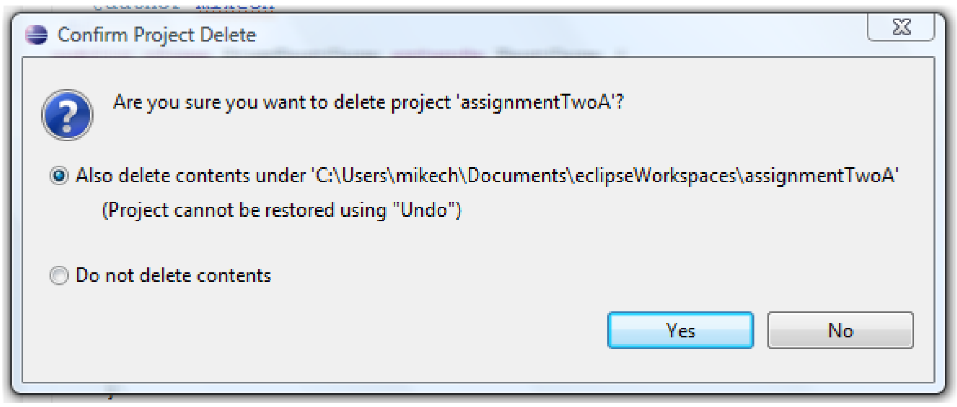
Importing Projects from a Zip File

Note that the material presented in this section is better described in the project video mentioned at the beginning of this document.

Many programming assignments will provide an exported project that you can import into you Eclipse workspace. These projects will be provided zip file archives that will be one of the files that can be downloaded from eLearning. The zip archive may contain sample code or a project template that can be used as a starting point for your efforts. You will be importing the project zip archive into your workspace.

Optional: Removing existing projects with the same name from the workspace

You cannot import a project with the same name into the workspace. This means that if you import and try to re-import the project template you must first delete the old project from the workspace. This is accomplished by selecting the existing project from the package explorer and selecting the "Edit > Delete" menu item. This will bring up the dialog shown in the following graphic. Notice that the option "Also Delete Contents Under C:\..." is selected**. It is very important that this option is selected** so that the project files are removed from you workspace


Confirm Project Delete

Are you sure you want to delete project 'assignmentTwoA'?

● Also delete contents under 'C:\Users\mikech\Documents\eclipseWorkspaces\assignmentTwoA'
(Project cannot be restored using "Undo")

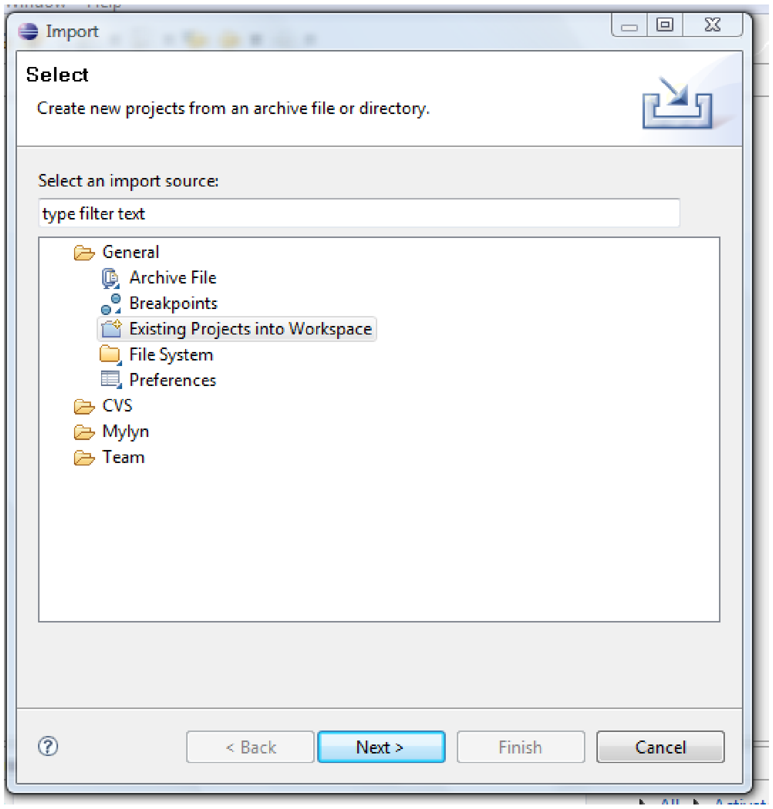
○ Do not delete contents

A this point the Old project will be have been removed from your workspace and you may begin importing the project template

Importing the Project

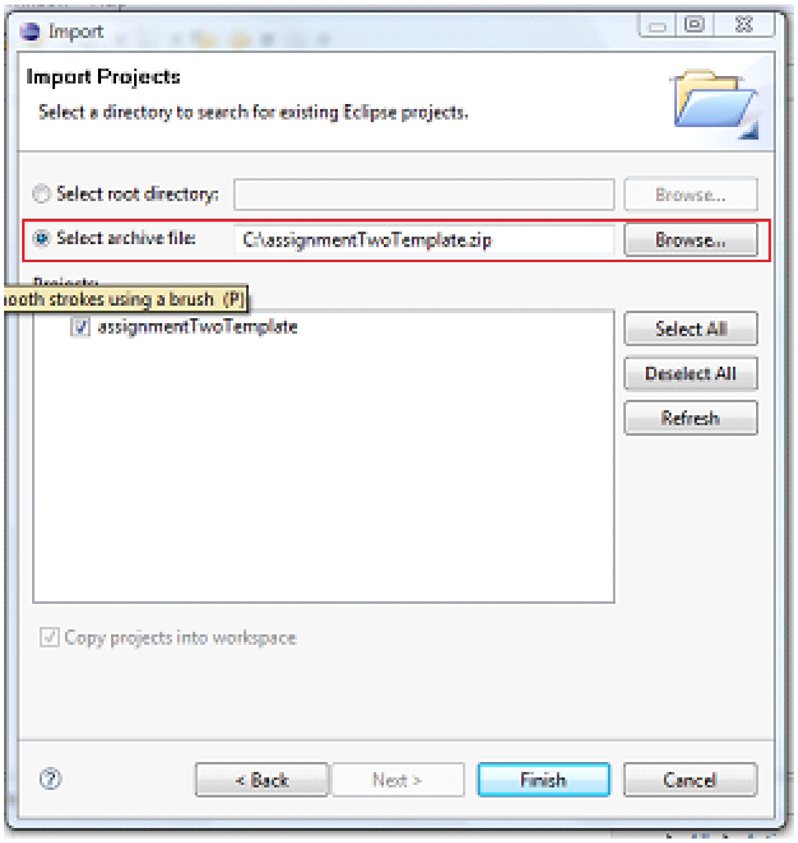
The process for importing the template project is a follows.

Open the import wizard using the "File > Import" menu item. This brings up the import dialog shown in the following graphic. Make sure to select the "Existing Projects into Workspace" option (under General) and press Next.


Import
Select
Create new projects from an archive file or directory.
Select an import source:
type filter text
General
Archive File
Breakpoints
Existing Projects into Workspace
File System
Preferences
CVS
Mylyn
Team
< Back
Next >
Finish
Cancel

This brings up the following import dialog. There are a few import things to note:

1. You need to select the “Select archive file” option and then press browse to select the project template archive (zip) file.
2. When the file opens, you need to select the project.
3. Press Finish and the project will be imported into your workspace.



Select archive file: C:\assignmentTwoTemplate.zip

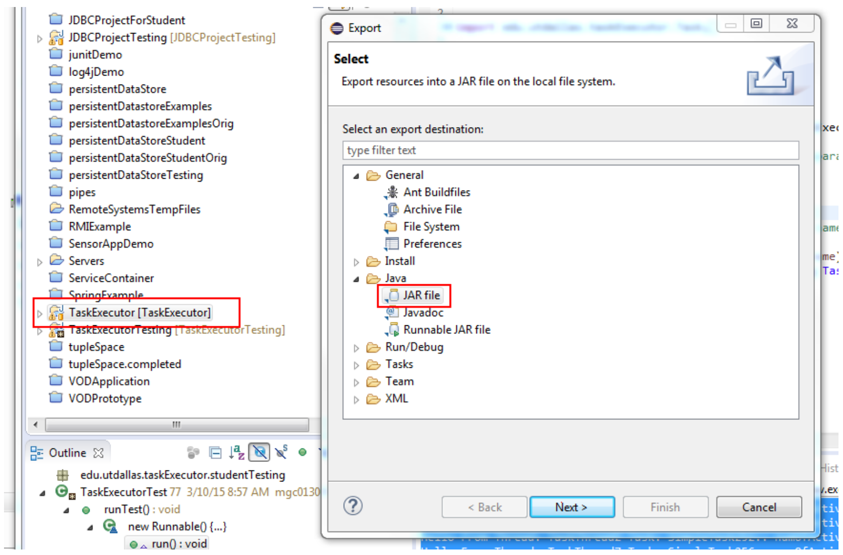
☑ assignmentTwoTemplate

Exporting an Eclipse Project as a Library JAR File

Note that the material presented in this section is better described in the project video mentioned at the beginning of this document.

This section provides a procedure describing of how to export the project containing your team’s TaskExecutor implementation as a library .jar file for submission.

1. Select the project that you wish to export.
2. Use the right mouse button, or the file menu, to select the Export feature.
3. Select Java >>JAR File as shown below, and then Next.



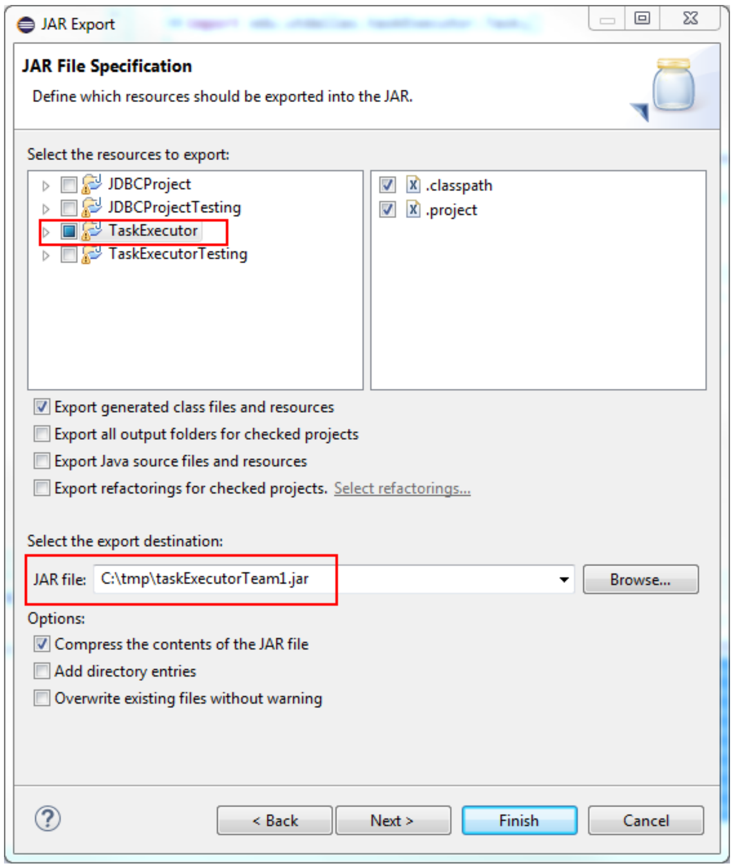
Select

Export resources into a JAR file on the local file system.

Select an export destination:

type filter text

4. On the JAR Export panel, make sure that the desired project is selected and enter the path and file name for the exported library jar file.
5. Select Finish and the export operation will be completed.



JAR File Specification

Define which resources should be exported into the JAR.

Select the resources to export:

[✓] .classpath
[✓] .project

[✓] Export generated class files and resources
[ ] Export all output folders for checked projects
[ ] Export Java source files and resources
[ ] Export refactorings for checked projects. Select refactorings...

Select the export destination:

JAR file: C:\tmp\taskExecutorTeam1.jar

Options:

[✓] Compress the contents of the JAR file
[ ] Add directory entries
[ ] Overwrite existing files without warning