Warehouse Laboratory

Introduction

In this exercise you are asked to play the part of a consultant to a warehousing company. In an interest to lower costs and improve efficiency, the company has chosen to automate the warehouse inventory system and utilize a robotic crane to perform the placement and removal of boxes. The company has the infrastructure in place and has provided three basic device driver algorithms that each use a different strategy to add and remove boxes using the crane. It is your job to implement three additional device driver algorithms that will provide more efficient operation and reduce inconsistencies in the inventory.

What you must do

There are three main tasks for you to perform in this laboratory. You are required to follow all style conventions and produce well-written, easily readable code. Your grade for this exercise will be affected by the quality of your code. The tasks for this laboratory are:

- Implement a device driver that is so simple that it is inefficient, similar to the provided device driver named RandomDriver.
- Implement a device driver that is similar to the provided OriginDriver, but with a slight improvement in performance.
- Implement a device driver that is similar to the provided DefragmentDriver, but with a slight improvement in performance.

This document contains instructions that will help you perform these tasks.

Provided code

There are 16 source code files provided in the laboratory archive. HTML documentation for each of the files is provided in the docs folder for the laboratory archive. You are not required to study the given files or their corresponding HTML documentation, but you may find them interesting or helpful. The provided files are an example of well-written Java code.

Six of the files contain classes that are the basis for the warehouse in the simulation. These files are listed below:

- Warehouse.java: Encapsulation of empty space to place parcels.
- Space.java: Space for a single parcel in a warehouse.
- Parcel.java: A single parcel to be placed in a warehouse.
Lot.java Group of related parcels.
WarehouseCrane.java Robotic crane operating on a warehouse.
Statistics.java Encapsulation of simulation statistics.

Five of the files contain classes that build the graphical user interface for the simulation. These files are listed below:

WarehouseView.java View for a Warehouse.
SpaceView.java View for a warehouse Space.
ParcelView.java View for a Parcel.
CraneView.java View for vital warehouse Crane information.
StatisticsView.java View for simulation Statistics.

Four of the files provide the specification and implementation of device driver algorithms for use in the simulation. These files are listed below:

DeviceDriver.java Abstract base class for device driver algorithms.
RandomDriver.java Device driver that performs random behavior.
OriginDriver.java Driver that places parcels in the upper left.
DefragmentDriver.java Driver that tries not to fragment parcel groups.

The last file contains the application class itself. This class contains the public static void main(String[] args) method required for a Java application, and provides the overall user interface that drives the program:

WarehouseApplication.java GUI application and main class.

This laboratory exercise makes use of the JPT. HTML documentation for the tools are available by following the link found in the docs directory of the laboratory archive. The given CodeWarrior™ project file is configured to make use of the JPT in its given location. An attempt to move the JPT archive or modify the project settings may render the tools unusable.

The warehouse

The simulated warehouse is two-dimensional, containing \(m\) rows by \(n\) columns of empty spaces. Each empty space in the warehouse is referred to by its row, column coordinates within the warehouse. The coordinate system works in the same fashion as computer graphics: the origin – (0, 0) – is in the upper left of the warehouse. The row numbering increases towards the bottom of the warehouse on the screen, and the column numbering increases towards the right of the warehouse on the screen. The bottom right space has the coordinates \((m-1, n-1)\) for a warehouse with \(m\) rows and \(n\) columns.

Each space in the warehouse is either empty, or contains exactly one parcel. All parcels used in this simulation are exactly the same size. There is no need to worry about the size and shape of each individual parcel to be placed in the warehouse. This simplification makes the warehouse simulation manageable, though not completely realistic.
Parcels are stored in and retrieved from the warehouse by lot. A lot is a collection of one or more related parcels. Individual parcels are color coded and numbered by their lot for easy reference. In addition, a parcel is displayed with a dashed border if its lot is fragmented: a lot is considered fragmented if one or more of the parcels in the lot is not stored in a space adjacent to another parcel in the lot. In this simulation, adjacent means that a space is closest to a given space in one of the eight main compass directions: N, NE, E, SE, S, SW, W, NW. For instance, the space at (1, 1) is adjacent to the eight spaces at locations (0, 1), (0, 2), (1, 2), (2, 2), (2, 1), (2, 0), (1, 0), and (0, 0), corresponding to the eight given compass directions in order.

There is a single crane in the warehouse that performs the actual operations of storing and retrieving parcels. The crane moves freely around the entire warehouse from one space to an adjacent space in any of the four main directions: N, E, S, and W. A crane has enough empty space to hold all of the parcels in a lot to be added to or removed from its warehouse. There is no need for the crane to move to a location in order to load parcels to be added to the warehouse, or to unload parcels that have been removed from the warehouse. Each lot to be placed in the warehouse appears automatically, and each lot that is removed from the warehouse disappears automatically.

The crane is simply a piece of movable machinery, and requires a software device driver in order to carry out its operations in the warehouse. Device drivers are interchangeable, which means that any driver for use in the crane can be installed at any time except in the middle of a simulation. There are only a few requirements for a device driver, which will be covered in a later section of this document.

In order to compare different device drivers, the crane compiles a number of statistics as it performs its operations. These statistics, and techniques for device driver comparison that use them, will be covered in a later section of this document.
The warehouse application

The following is a screen shot of the warehouse application:

There is a great deal of information available in the user interface for this application. It is separated into the seven sections shown above and described below.

The section of the user interface named “Inventory” shows all of the spaces in the warehouse. Parcels contained in spaces are painted in a color corresponding to their lot number, which is displayed on the top of the parcel. Parcels with a dashed outline are in a lot that was fragmented when it was placed in the warehouse. The black square shows the current location of the warehouse crane. The red square shows the destination for the crane as it moves to perform an operation on that space.

The section named “Crane” shows all of the relevant information for the warehouse crane. The name of the device driver is listed, along with the parcels currently held by the crane, the current and destination location for the crane, and a brief description of the current crane status. The current and destination locations are shown in black and red, respectively, in agreement with the colors of the corresponding squares in the “Inventory” section. The crane
status displays the general operation being performed by the crane, but does not describe specific details about the strategy used by the currently installed device driver.

The section named “Statistics” shows all of the gathered statistics for the current simulation run. For each statistic, its current value and value per operation are displayed. These values are updated in real time, and will change as a simulation progresses. The statistics themselves are discussed in a later section of this document. The “Print” button reports the current simulation statistics in the text console for the application. These reports are useful for comparing the performance of several device drivers or successive simulation runs for the same device driver.

The section named “Device Drivers” shows the device drivers available for use in the simulation. As long as a simulation is not currently running, you can choose the device driver to be installed in the warehouse crane by selecting the desired driver from this list. See the “Testing and Troubleshooting” section of this document to find out how to add a new device driver to this list.

The section named “Simulation” contains buttons to start and stop a simulation. Pushing the “Start” button will lock in the choice of simulation parameters and install the selected device driver in the warehouse crane. These choices will remain until the simulation is stopped or the simulation runs to completion. Pushing the “Stop” button will immediately end a running simulation, enabling you to select a different device driver and change the simulation parameters.

The section named “Parameters” shows the current simulation parameters. As long as a simulation is not currently running, you can change the parameter values by replacing the values in the text fields. The parameters for the simulation will be extracted when a simulation is started, and will remain in effect until the simulation completes or is stopped. The parameters themselves are described in a later section of this document.

The section named “Animation” contains a slider that governs the speed of the animation used in the simulation. The value for this slider can be changed at any time, whether a simulation is currently running or not.

A simulation and its parameters

Before you attempt to write your own device driver algorithms for use in this simulation, you should run at least one simulation for each of the provided device drivers. This will give you an idea of the behavior of each of the provided drivers, and will help you understand the meaning of the different simulation parameters. The four simulation parameters are described below:

Random seed

This value is used to determine the order of the random numbers used by the simulation to choose what will happen next. Each random number seed will produce a different set of random numbers, and will consequently produce a different simulation. Choosing the same random seed for two successive simulations, however, will produce the same set of random numbers and will consequently produce similar or equivalent simulations. The other parameters for the simulation, including the device driver that is chosen, will determine whether or not two successive simulations are exactly the same, but the same random seed must be chosen in order to even have a hope of producing the same simulation twice.
Number of lots

This value represents the number of lots that will be added to the warehouse over the course of the simulation. Depending on other factors, including the random seed and the remove bias, the number of lots that are removed from the warehouse can vary from zero to this value. Increasing this value will produce a longer simulation. Decreasing this value will produce a shorter simulation.

Maximum lot size

This value represents the greatest number of parcels that can be contained in a randomly generated lot. Increasing this value will reduce the average number of lots that can fit in the warehouse. Decreasing this value will increase the average number of lots that can fit in the warehouse.

Remove bias

This value represents the percentage chance that a lot currently contained in the warehouse will be removed. Increasing this value will increase the number of lots that are removed over the course of a simulation, leaving more room for lots to be added. Decreasing this value will decrease the number of lots that are removed over the course of a simulation, leaving less empty space for lots to be added.

You should experiment with different values for each parameter, to gain an understanding of their influence on the simulation.

Device drivers

To put it simply, a device driver is what tells the crane where to go and what to do when a lot of parcels is to be added to or removed from the warehouse. This is an example of the strategy pattern: create classes of objects that represent different strategies for accomplishing a goal. There are three device drivers provided to you for this exercise. You will produce three more device drivers. All in all, there will be six different strategies for using the crane to add parcels to and remove parcels from the warehouse.

There are two basic operations that are required of a device driver: add a lot of parcels and remove a lot of parcels. The abstract class DeviceDriver encapsulates this basic structure in a way that leaves the details of the two operations up to the different strategies themselves. Only code that is common to all device drivers is included in the DeviceDriver class.

The common structure of all device drivers for adding a lot of parcels to the warehouse is provided in the addLot method in the file DeviceDriver.java. This method takes a lot of parcels to be added as its single parameter and is to return true if the lot was added correctly, or false if the lot could not be added correctly. The most important details of this method are as follows:
public boolean addLot(Lot lot) {
    beginAdd(lot);

    Parcel[] parcels = lot.getParcels();

    for (int i = 0; i < parcels.length; i++) {
        Point location = getAddLocation(p);

        // if no location found, quit
        if (location == null)
            return false;

        // move to location
        crane.moveTo(location.x, location.y);

        // place parcel
        replaced = crane.add(parcels[i]);

        // if a parcel was replaced
        if (replaced != null) {
            parcelRemoved(replaced, location);

            parcelAdded(parcels[i], location);

            // quit
            return false;
        }

        // otherwise simply note added parcel
        parcelAdded(parcels[i], location);
    }

    // all parcels placed
    return true;
}

The methods listed in boldface are methods whose behavior must be defined by individual device drivers to implement their own strategy for accomplishing the goal of adding the lot of parcels. These methods will be described below.

The common structure of all device drivers for removing a lot of parcels from the warehouse is provided in the removeLot method in the file DeviceDriver.java. This method takes a lot of parcels to be added as its single parameter and is to return true if the lot was removed correctly, or false if the lot could not be removed correctly. The most important details of this method are as follows:
public boolean removeLot(Lot lot) {
    beginRemove(lot);

    Parcel[] parcels = lot.getParcels();

    // for each parcel in the lot
    for (int i = 0; i < parcels.length; i++) {

        // get location from which to remove parcel
        Point location = getRemoveLocation(parcels[i]);

        // if no location found, quit
        if (location == null)
            return false;

        // move to location
        crane.moveTo(location.x, location.y);

        // place parcel
        removed = crane.remove();

        // note that parcel was added
        parcelRemoved(removed, location);

        // if the wrong parcel was removed, quit
        if (parcels[i] != removed)
            return false;
    }

    // all parcels placed
    return true;
}

The methods listed in boldface are methods whose behavior must be defined by individual device
drivers, to implement their own strategy for accomplishing the goal of removing the lot of
parcels. These methods are described below.

protected Point getAddLocation(Parcel p)

This method should return a Point representing the row and column coordinates where the
given parcel should be added to the warehouse, or null if no such point can be found. The point
to be returned should have its x component set to the appropriate row, and y component set to
the appropriate column in the warehouse. At the very least, a device driver must override this
method in order to result in a working simulation. For example, all three provided device drivers
override this method.

protected Point getRemoveLocation(Parcel p)

This method should return the Point representing the row and column coordinates where the
given parcel should be removed from the warehouse, or null if no such point can be found. The
point to be returned should have its x component set to the appropriate row, and y component set
to the appropriate column in the warehouse. As in the getAddLocation method, a device
driver must override this method in order to result in a working simulation. For example, all
three provided device drivers override this method.

protected void parcelAdded(Parcel p, Point location)

This method is called immediately after the given parcel was added to the warehouse at the given
location. The given Point will have its x component set to the row, and its y component set to
the column, where the given parcel was added. A device driver should override this method if it
wishes to store information regarding the exact locations of parcels that have been added to the
warehouse. For example, the provided OriginDriver and DefragmentDriver both override
this method and store information for each parcel added to the warehouse.

protected void parcelRemoved(Parcel p, Point location)

This method is called immediately after the given parcel was removed from the warehouse at the
given location. The given Point will have its x component set to the row, and its y component
set to the column, where the given parcel was removed. A device driver should override this
method if it wishes to store information regarding the exact locations of parcels that have been
removed from the warehouse. For example, the provided OriginDriver and
DefragmentDriver both override this method and store information for each parcel removed
from the warehouse.

protected void beginAdd(Lot lot)

This method is called at the very beginning of the addLot method. A device driver should
override this method if it wishes to perform any operations that pertain to the entire lot of parcels
before they are individually placed. Remember that the getAddLocation method described
above does not provide information about the entire lot of parcels, but instead just provides the
individual parcel in the lot that is to be added at that time. For example, the provided
DefragmentDriver overrides this method to find a suitable set of locations for an entire lot of
parcels, before any of the parcels are to be added.

protected void beginRemove(Lot lot)

This method is called at the very beginning of the removeLot method. A device driver should
override this method if it wishes to perform any operations that pertain to the entire lot of parcels
before they are individually removed. Remember that the getRemoveLocation method
described above does not provide information about the entire lot of parcels, but instead just
provides the individual parcel in the lot that is to be removed at that time. There is no example
of a device driver overriding this method, as it is not necessary for any of the provided device
drivers.
The provided device drivers

Three example device drivers are provided with this exercise. Each represents a specific kind of strategy that can be used to govern the adding and removing of parcels in this simulation. Each device driver, and the kind of strategy it represents, is described below:

Randomosity v1.0: the RandomDriver class

This device driver class uses a bare minimum of strategy. Whenever a parcel is to be added to or removed from the warehouse, this driver provides a randomly selected Point at which the parcel should be added or removed. Since this driver does not store any information about the current state of the warehouse, it is impossible for this driver to make an educated choice of a Point for either an add or remove operation. It is likely that the Point randomly selected in getAddLocation already contains a parcel. It is even more likely that the Point randomly selected in getRemoveLocation does not contain the parcel to be removed.

This device driver is an example of an algorithm that does not store any information about the state of the data it operates upon. Algorithms like this have extremely bad performance, both in terms of efficiency and correctness. It is not suggested that you use algorithms like this in your own work.

Origins v2.2.1: the OriginDriver class

This device driver class attempts to place each parcel in a lot as close to the origin – row 0, column 0 – as possible. In order to do so, it keeps track of the lot number of the parcel contained at each location in the warehouse. A specially chosen lot number is used to represent an empty location in the warehouse. Since this driver stores this information about the state of the warehouse, it is always able to produce a sensible location to add a parcel, unless the warehouse is completely full, and is always able to produce the location of a parcel to be removed, as long as the parcel is actually contained in the warehouse.

This device driver is an example of an algorithm that stores information about the state of the data it operates upon. Because it stores this useful information, it is able to perform correctly whenever possible. This algorithm is not particularly efficient, however, as it performs unnecessary steps every time it adds or removes a parcel. Often an algorithm such as this represents a good start towards an efficient, correct algorithm, and needs only be optimized to eliminate any unnecessary steps it performs.

Anti-frag v1.1: the DefragmentDriver class

This device driver class attempts to place each of the parcels in a lot next to each other in the same row of the warehouse. In order to do so, it keeps track of the lot number of the parcel contained at each location in the warehouse in the same fashion as the OriginDriver described above. This driver has many of the same qualities as the OriginDriver, except that it uses the stored information to a much greater extent than the OriginDriver. Before a lot of parcels is
to be added, this driver searches through the locations in the warehouse in order to find a good place to put the entire lot of parcels, even before a single parcel is to be placed in the warehouse. Specifically, for this driver, a “good place” to put a lot of parcels is defined as one that reduces the chance that the lot of parcels will be fragmented.

This device driver is an example of an algorithm that not only stores useful information about the state of the data it operates upon, but also preprocesses the data before performing an operation. Because it stores state information, it is able to perform correctly in the same regard as the OriginDriver. This algorithm is more efficient, however, as the preprocessing of data finds appropriate locations for all parcels in a lot ahead of time, rather than as each individual parcel is to be added. Algorithms such as these are usually the most efficient kind of algorithms. The actual level of efficiency of the algorithm depends on the usefulness of the preprocessing performed on the data.

Writing your device drivers

You are required to write three additional device drivers for use in this simulation. Each of these device drivers should be similar to, but not exactly the same as, the three provided device drivers. You will write one device driver that is so simple that it is inefficient, in the same regard as the provided RandomDriver, as it will not store any information about the warehouse. You will write one device driver that stores information about the warehouse in the same fashion as the given OriginDriver, but does not perform obviously unnecessary steps. Finally, you will write a device driver that preprocesses stored information in the same fashion as the provided DefragmentDriver, but performs its operations even more efficiently.

Each of these device drivers will be a public class in its own file. For each, you will need to create a new file, write the code that defines the class, save the file with the same name as the class, and add that file to the CodeWarrior project. You may name the class whatever you like, but its name should make it clear that the class is a device driver. Use the names of the provided device drivers as a guide. For details regarding how to create new files, how to save files, and how to add files to a project, consult the help files on the course web site.

There are two methods in the DeviceDriver class you need to be aware of when creating your device drivers: getName and reset. The getName method should return a short name for the device driver. You will need to override this abstract method for each of the device drivers you create, and return a unique name for the driver. The reset method is called whenever a device driver is chosen for use in the simulation, and should set all of the parameters for the device driver back to their initial values. The first thing that should be done in the reset method for any device driver is to call super.reset(). This will enable the base DeviceDriver class to do the work it needs to do to reset itself, before your device driver does the work it needs to do to reset itself. Consult the reset methods in the provided device drivers as a guide.

Your intentionally inefficient driver

Your first driver will not store information about the warehouse, and as a result, will be inefficient. This class should extend DeviceDriver, and override two methods:
getAddLocation, and getRemoveLocation. (You may want to use the similar class
RandomDriver as a guide when writing this device driver.) Since this driver will not store
information about the warehouse, it will have to choose a location in the warehouse
independently of the current state of the warehouse.

We ask you to write your driver so that it chooses a location at random from the outermost rows
and columns. Specifically, your driver should always choose locations for which one of the
following is true: the location is in row 0, the location is in column 0, the location is in the last
row, the location is in the last column. Like the strategy used in RandomDriver, this is
extremely bad and will result in poor performance.

You will need to know the number of rows and columns in the warehouse in order to choose
locations in this way. Inside any class that extends DeviceDriver, the following method calls
will return the number of rows and columns in the warehouse, respectively:

    warehouse.getRowCount();
    warehouse.getColumnCount();

Write the getAddLocation and getRemoveLocation methods so that they each choose a
location in the outermost rows and columns of the warehouse. The following method call will
produce a random int value that is equal to one of the numbers 0, 1, 2, or 3:

    int i = (int)(WarehouseApplication.nextDouble() * 4);

You can use this random number in the following way to choose the basis for a random location:

    if (i == 0)
        // choose a location in row 0
    else if (i == 1)
        // choose a location in column 0
    else if (i == 2)
        // choose a location in the last row
    else
        // choose a location in the last column

Once you have chosen a row or column for your location, you must choose the other coordinate
at random. For example, if you are to choose a location in row 0, you must select the column at
random. The following method call will produce a random int value to be used as the column
index of the location:

    int cols = warehouse.getColumnCount();
    int c = (int)(WarehouseApplication.nextDouble() * cols);

Given these tools, and the RandomDriver as a guide, you should be able to write your first
device driver in its entirety. See the “Testing and Troubleshooting” section that comes later in
this document for instructions on how to add your device driver to the simulation and test its
performance.
A device driver with memory

Your second device driver should store information about the warehouse, in order to provide a measure of performance efficiency. This class should extend DeviceDriver, and override four methods: getAddLocation, getRemoveLocation, parcelAdded, and parcelRemoved. (You may want to use the similar class OriginDriver as a guide when writing this device driver.) Since this driver will store information about the warehouse, it will be able to choose a location in the warehouse based on the current state of the warehouse. One way to do this is to choose an empty location closest to the current location of the crane to add a parcel, and choose a closest location containing a parcel from the desired lot to remove a parcel.

The provided device driver OriginDriver is inefficient because it always looks for the closest space to the origin that is either empty or contains a parcel from the lot to be removed. In order for this to be a sensible approach, the OriginDriver returns the crane to a location of (0, 0) after each parcel is added or removed. This unnecessary movement takes time and reduces the efficiency of this device driver.

Your device driver should be very similar to OriginDriver, but it should consider a location nearest the center as its home base, rather than the origin. OriginDriver selects the empty location closest to the current crane location in order to add a parcel. Your driver should select the empty location closest to the center. OriginDriver selects the closest location containing a parcel of the desired lot in order to remove a parcel. Your driver should have the same behavior for removing parcels. In order to provide this behavior, your driver will have to store the locations of parcels that are added and removed in the same fashion as OriginDriver.

See the “Testing and Troubleshooting” section that comes later in this document for instructions on how to add your device driver to the simulation and test its performance. When your driver is used in the simulation, it should perform more efficiently on the average than the OriginDriver. If your driver is not more efficient in terms of the average number of crane moves per operation, you should check to make sure that you have written your driver correctly.

An advanced device driver

Your final device driver should not only store information about the warehouse, but also preprocess the stored information in order to find the best location for each lot to be added or the best order in which to remove the parcels in each lot to be removed. This class should extend DeviceDriver, and override at least five of the following six methods: getAddLocation, getRemoveLocation, parcelAdded, parcelRemoved, beginAdd, and beginRemove. (You may want to use the similar class DefragmentDriver as a guide when writing this device driver.) Depending on the nature of the strategy that you implement, you may not need to override the beginRemove method. For example, the provided DefragmentDriver does not override this method.

Since this driver will store information about the warehouse, it will be able to choose a location in the warehouse based on the current state of the warehouse.
the stored information, it will be able to choose a good location for each of the parcels to be
added to or removed from the warehouse, before any individual parcels are actually added or
removed. There are many strategies that can be used to find good locations for each of the
parcels in a lot to be added. The provided DefragmentDriver attempts to store a lot of parcels
next to each other in the same row, but this is only one possible strategy.

It is your job to devise a strategy for choosing good locations for each of the parcels in a lot to be
added. You may want to use the similar class DefragmentDriver as a guide when writing this
device driver, but you may not use the exact same strategy as used in the DefragmentDriver
class. In order to come up with a good strategy, you may want to draw pictures of example
warehouses and perform add and remove operations on paper. It is not suggested that you only
use trial and error to devise a strategy for this device driver, although you may need to
experiment with different strategies before you find one that is efficient. Think of the problem in
terms of the real world: how would you approach this problem in a real warehouse?

In order to make your driver most efficient, you will need to be able to find the current location
of the crane. Inside any class that extends DeviceDriver, the following method call will return
a Point containing the row and column coordinates of the current location of the crane. The x
component will be set to the current row and the y component will be set to the current column:

    crane.getLocation();

In order to test the distance between two warehouse locations, in terms of the number of steps
required for the crane to move from one location to the other, you can call the static distance
method in the Crane class as follows:

    Crane.distance(p, q);

The above example would return the number of steps required to move the crane from the Point
named p to the Point named q. This method does not take into account the current location of
the crane when calculating the distance.

In general, any class that extends device driver has access to the warehouse crane and the
warehouse itself. The data members crane and warehouse in the DeviceDriver class are
protected member data, meaning that they are accessible in classes that extend DeviceDriver.
Depending on your strategy, you may want to call methods on those two objects.

See the “Testing and Troubleshooting” section that comes later in this document for instructions
on how to add your device driver to the simulation and test its performance. When your driver is
used in the simulation, it should perform more efficiently on the average than the
DefragmentDriver. Your driver does not need to be more efficient than the
DefragmentDriver in terms of every simulation statistic, but your driver should be equally
efficient or more efficient in terms of nearly all of the statistics.
Testing and troubleshooting

While writing the code required to complete the tasks for this laboratory, you may receive notice of many different kinds of errors. A listing of compiler errors and possible causes is available on the course website. Contact your instructor if you have further problems.

When you are ready to test a DeviceDriver, you must make that implementation available for use in the application. In WarehouseApplication.java, there is a method named addDeviceDrivers that adds each of the device drivers for use in the application. Using the existing code as a guide, add your driver to the application. Once your driver has been added to the application, it should be available for selection within the user interface.

In order to test your driver, you must select your device driver by its unique name, and run one or more simulations using that driver. Reduce the animation speed to see exactly where your device driver moves the crane in different situations. Increase the animation speed to quickly run a simulation to completion, in order to gather meaningful statistics about the performance of your driver. If you are unable to easily follow the behavior of your device driver, you may want to have the driver give feedback while a simulation is running.

To print out feedback in the console window that opens along with the application window, you may use the following line of code:

```java
console.out.println(something to print);
```

You can print out just about any value in the console window, including Strings and other Java Objects such as points. Place lines of code like the above in your driver code in order to print out information regarding the status of the driver, the locations at which add or remove operations are to occur, and any other information you deem necessary to understand the behavior of the driver.

The following are problems you may encounter when running a warehouse simulation:

- The simulation appears to not run at all.
- The simulation runs for a while, and then stops before the correct number of lots have been added.
- The crane seems to jump around wildly in the warehouse, and parcels seem to appear and disappear.
- The selected driver most likely does not have a call to super.reset() as the first step in its reset method. If the selected driver is one of the drivers you wrote, add a call to super.reset() as the first line in the reset method, and try to run the simulation again. This error should not occur when any of the provided drivers are selected.
- The selected driver failed to add two straight lots of parcels. This is either because the warehouse is too full to add a lot of parcels, or because the selected driver returned null in the getAddLocation or getRemoveLocation methods.
- The simulation is running too quickly for the user interface. This is not really an error, but will make it difficult to see what is happening in the simulation. Reduce the animation speed until the behavior is clear.
You receive a Java Exception: ArrayIndexOutOfBoundsException

The selected driver may have attempted to move the crane out of the row and/or column bounds of the warehouse. Alternately, the selected driver may have attempted to access stored information about a location in the warehouse that is out of the row and/or column bounds of the warehouse. This should not occur when any of the provided drivers are selected.

Grading Scheme and Extra Credit

10 Up to 10 points for a simple, inefficient driver.
15 Up to 15 points for a driver similar to the provided OriginDriver.
20 Up to 20 points for a driver similar to the provided DefragmentDriver.
5 Up to 5 extra credit points for extremely good driver performance.
5 Up to 5 points for good style and techniques.

50 “Perfect” score without any extra credit
55 Maximum total grade

Finishing up your work

Once you have completed the exercise, you must go through all of the files you created or modified, and perform a process called refactoring. Refactoring includes editing your code so that it maximizes readability, follows all necessary coding conventions, and contains no spelling errors in any included comments. It is mandatory for you to refactor the code that you produce for each of your laboratory exercises in this way. You will not only be graded on the output of your application and your understanding of the key concepts, but also on the clarity and overall style used in your code.

There are many general code styles that are acceptable for this course, as in industry. You are not required to adhere to any one specific style, but you must consistently follow an acceptable style throughout your work on each laboratory exercise. For more information about code style, consult the reference “The Elements of Java Style,” or contact your instructor.

Once you have refactored your code and are satisfied with both its output and its style, you must submit your work to the instructor. You must submit the following items for this exercise:

- Soft copy of the entire project folder.
- Hard copy (printout) of each of the three device drivers you created.

Follow the soft copy submission instructions and guidelines as described by your instructor.