**CS2420 Capstone Project**

The development phase of your project is due at the end of Module 6, so that you improve your chances of getting your final project done on time and as designed. Since this part of the project will take the most time to complete, be sure to get started no later than the beginning of Module 4.

For your choice from the projects appearing below, **a primary tasks during design is to plan how to convert it from a console application to a GUI application**. Use the Product Maintenance program presented in Chapter 21, which converts the Chapter 11 console version, as a model.

**Project A: Maintain customer data (text or XML file)**

**Console**

Welcome to the Customer Maintenance application  
   
COMMAND MENU  
list - List all customers  
add - Add a customer  
del - Delete a customer  
help - Show this menu  
exit - Exit this application  
   
Enter a command: list  
   
CUSTOMER LIST  
frank46@hotmail.com Frank Jones  
sarah\_smith@yahoo.com Sarah Smith  
   
Enter a command: add  
   
Enter customer email address: test@gmail.com  
Enter first name: text  
Enter last name: test  
   
text test was added to the database.  
   
Enter a command: list  
   
CUSTOMER LIST  
frank46@hotmail.com Frank Jones  
sarah\_smith@yahoo.com Sarah Smith  
test@gmail.com text test  
   
Enter a command: del  
   
Enter customer email to delete: test@gmail.com  
   
text test was deleted from the database.  
   
Enter a command: list  
   
CUSTOMER LIST  
frank46@hotmail.com Frank Jones  
sarah\_smith@yahoo.com Sarah Smith  
   
Enter a command: exit  
   
Bye.

**Operation**

* This application will present a user with five choices: list, add, delete, help, and exit.
  + Menu choices may be used, but buttons are typically used.
* If the user chooses "list", the application displays the customer data that's stored in a text or XML file for persistent storage.
* If the user chooses "add", the application prompts the user to enter data for a customer and saves that data to the persistent storage.
* If the user chooses "delete", the application prompts the user for an email address and deletes the corresponding customer from the persistent storage.
* If the user chooses "help", the application displays the usage information.
* If the user chooses "exit", the application displays a goodbye message and exits.

**Specifications**

* Create a class named Customer that stores data for the user's email address, first name, and last name.
* Create interfaces named CustomerReader and CustomerWriter that define the methods that will be used to read and write customer data to persistent storage In addition, create an interface named CustomerConstants that contains three constants that specify the display size of a customer's email address (30), first name (15), and last name (15). Then, create an interface named CustomerDAO that inherits all three of these interfaces.
* Create at least one persistence class that implements the methods specified by the CustomerDAO interface. Store the customer data. If you use a text or XML file which doesn't exist, this class should create it.
* Create a class named DAOFactory that contains a method named getCustomerDAO. This method should return an instance of the persistence class.
* Create a CustomerMaintApp class that controls execution. This class should use the DAOFactory class to get a CustomerDAO object. Then, it should use the methods of the CustomerDAO object to read customer data from and write customer data to the persistent storage.
* Use the Validator class or a variation of it to validate the user's entries. Non-empty strings are required for the email address, first name, and last name.
* Use spaces to align the customer data in columns on the console. To do that, you can create a utility class named StringUtils with a method that adds the necessary spaces to a string to reach a specified length.
* Add an "update" command that lets the user update an existing customer. This command should prompt the user to enter the email address. Then, it should let the user update the first name or last name for the customer.
* Add a method to the Validator class that uses string parsing techniques to validate the email address. At the least, you can check to make sure that this string contains some text, followed by an @ sign, followed by some more text, followed by a period, followed by some more text. For example, "x@x.x" would be valid while "xxx" or "x@x" would not.

**Project B: Maintain customer data (XML file)**

**Console**

Welcome to the Customer Maintenance application  
   
COMMAND MENU  
list - List all customers  
add - Add a customer  
del - Delete a customer  
help - Show this menu  
exit - Exit this application  
   
Enter a command: list  
   
CUSTOMER LIST  
frank46@hotmail.com Frank Jones  
sarah\_smith@yahoo.com Sarah Smith  
   
Enter a command: add  
   
Enter customer email address: xml\_test@gmail.com  
Enter first name: XML  
Enter last name: Test  
   
XML Test was added to the database.  
   
Enter a command: list  
   
CUSTOMER LIST  
frank44@hotmail.com Frank Jones  
sarah\_smith@yahoo.com Sarah Smith  
xml\_test@gmail.com XML Test  
   
Enter a command: del  
   
Enter customer email to delete: xml\_test@gmail.com  
   
XML Test was deleted from the database.  
   
Enter a command: list  
   
CUSTOMER LIST  
frank44@hotmail.com Frank Jones  
sarah\_smith@yahoo.com Sarah Smith  
   
Enter a command: exit  
   
Bye.

**Operation**

* This application begins by displaying a menu with five choices: list, add, del, help, and exit.
* If the user enters "list", the application displays the customer data that's stored in an XML file.
* If the user enters "add", the application prompts the user to enter data for a customer and saves that data to the XML file.
* If the user enters "del", the application prompts the user for an email address and deletes the corresponding customer from the XML file.
* If the user enters "help", the application displays the menu again.
* If the user enters "exit", the application displays a goodbye message and exits.

**Specifications**

* Create a class named Customer that stores data for the user's email address, first name, and last name.
* Use a text editor to create an XML file named customers.xml in the same directory as the Customer class. This file should contain valid XML tags for at least one customer. For example:

<?xml version="1.0" encoding="UTF-8"?>  
<Customers>  
<Customer Email="frank44@hotmail.com">  
<FirstName>Frank</FirstName>  
<LastName>Jones</LastName>  
</Customer>  
</Customers>

* Create a class named CustomerXMLFile that reads the customers.xml file when it's instantiated. This class should include a public method that return an array list of Customer objects created from the data in the file, a public method that returns a Customer object for a specified email address, and public methods for adding and deleting records. Include any additional private methods that you need to perform these functions.
* Create a CustomerMaintApp class that works as shown in the console output. This class should use the Customer and CustomerXMLFile classes to work with Customer data.
* Use spaces to align the customer data in columns on the console. To do that, you can create a utility class named StringUtils that has a method that adds the necessary spaces to a string to reach a specified length.
* Use the Validator class or a variation of it to validate the user's entries. Non-empty strings are required for the email address, first name, and last name.
* Add an "update" command that lets the user update an existing customer. This command should prompt the user to enter the email address for a customer. Then, it should let the user update the first name and last name for the customer.
* Add a method to the Validator class that uses string parsing techniques to validate the email address. At the least, you can check to make sure that this string contains some text, followed by an @ sign, followed by some more text, followed by a period, followed by some more text. For example, "x@x.x" would be valid while "xxx" or "x@x" would not.
* Use an interface to eliminate any direct calls to the CustomerXMLFile class from the CustomerMaintApp class. To do that, you can use CustomerReader, CustomerWriter, CustomerConstants, and CustomerDAO interfaces as well as a DAOFactory class as described in project 19-3. Then, you can modify the CustomerXMLFile and CustomerMaintApp classes so they use these interfaces and class.

**Project C: Maintain customer data**

**Console**

Welcome to the Customer Maintenance application  
   
COMMAND MENU  
list - List all customers  
add - Add a customer  
del - Delete a customer  
help - Show this menu  
exit - Exit this application  
   
Enter a command: list  
   
CUSTOMER LIST  
frankjones@yahoo.com Frank Jones  
johnsmith@hotmail.com John Smith  
seagreen@levi.com Cynthia Green  
wendyk@warners.com Wendy Kowolski  
   
Enter a command: add  
   
Enter customer email address: jdbc\_test@gmail.com  
Enter first name: JDBC  
Enter last name: Test  
   
JDBC Test was added to the database.  
   
Enter a command: list  
   
CUSTOMER LIST  
frankjones@yahoo.com Frank Jones  
jdbc\_test@gmail.com JDBC Test  
johnsmith@hotmail.com John Smith  
seagreen@levi.com Cynthia Green  
wendyk@warners.com Wendy Kowolski  
   
Enter a command: del  
   
Enter customer email to delete: jdbc\_test@gmail.com  
   
JDBC Test was deleted from the database.  
   
Enter a command: list  
   
CUSTOMER LIST  
frankjones@yahoo.com Frank Jones  
johnsmith@hotmail.com John Smith  
seagreen@levi.com Cynthia Green  
wendyk@warners.com Wendy Kowolski  
   
Enter a command: exit  
   
Bye.

**Operation**

* This application begins by displaying a menu with five choices: list, add, del, help, and exit.
* If the user enters "list", the application displays the customer data that's stored in a database table.
* If the user enters "add", the application prompts the user to enter data for a customer and saves that data to the database table.
* If the user enters "del", the application prompts the user for an email address and deletes the corresponding customer from the database table.
* If the user enters "help", the application displays the menu again.
* If the user enters "exit", the application displays a goodbye message and exits.

**Specifications**

* Create a Derby database named MurachDB that contains the necessary data. To do that, you can Use the SQL script stored in the CreateMurachDB.sql file that's stored in the \murach\java\project\_starts\project\_21-2 directory. (If you did project 21-1, you have already created this database.)
* Create a class named Customer that stores data for the user's email address, first name, and last name.
* Create a class named CustomerDB. This class should include a public method that returns an array list of Customer objects for the customers in the Customer table, a public method that returns a Customer object for the customer with a specified email address, and public methods that add a record to and delete a record from the Customer table.
* Create a CustomerMaintApp class that works as shown in the console output. This class should use the Customer and CustomerDB classes to work with the customer data.
* Use spaces to align the customer data in columns on the console. To do that, you can create a utility class named StringUtils that has a method that adds the necessary spaces to a string to reach a specified length.
* Use the Validator class or a variation of it to validate the user's entries. Non-empty strings are required for the email address, first name, and last name.
* Add an "update" command that lets the user update an existing customer. This command should prompt the user to enter the email address for a customer. Then, it should let the user update the first name and last name for the customer.
* Add a method to the Validator class that uses string parsing techniques to validate the email address. At the least, you can check to make sure that this string contains some text, followed by an @ sign, followed by some more text, followed by a period, followed by some more text. For example, "x@x.x" would be valid while "xxx" or "x@x" would not.

**Project D: Tortoise and the hare race**

**Console**

Get set...Go!  
Tortoise : 10  
Tortoise : 20  
Tortoise : 30  
Tortoise : 40  
Hare : 100  
Tortoise : 50  
Tortoise : 60  
Tortoise : 70  
Tortoise : 80  
Hare : 200  
Tortoise : 90  
Tortoise : 100  
.  
. (output lines omitted)  
.  
Hare : 500  
Tortoise : 900  
Tortoise : 910  
Tortoise : 920  
Tortoise : 930  
Tortoise : 940  
Tortoise : 950  
Tortoise : 960  
Tortoise : 970  
Tortoise : 980  
Tortoise : 990  
Tortoise : 1000  
Tortoise: I finished!  
  
The race is over! The Tortoise is the winner.  
  
Hare: You beat me fair and square.

**Operation**

* This application simulates a race between two or more runners. The runners differ in their speed and how often they need to rest. One of the runners, named "Tortoise," is slow but never rests. The other runner, named "Hare," is ten times as fast but rests 90% of the time. Control these ratios from a text configuration file which is read at the start of each race.
* There is a random element to the runners' performance, so the outcome of the race is different each time the application is run.
* The race is run over a course of 1000 meters. Each time one of the runners moves, the application displays the runner's new position on the course. The first runner to reach 1000 wins the race.
* When one of the runners finishes the race, the application declares that runner to be the winner and the other runner concedes.
* After each race, store the results to a text file or a database table.

**Specifications**

* Each runner should be implemented as a separate thread using a class named ThreadRunner. The ThreadRunner class should include a constructor that accepts three parameters: a string representing the name of the runner, an int value from 1 to 100 indicating the likelihood that on any given move the runner will rest instead of run, and an int value that indicates the runners speed—that is, how many meters the runner travels in each move.
* The run method of the ThreadRunner class should consist of a loop that repeats until the runner has reached 1000 meters. Each time through the loop, the thread should decide whether it should run or rest based on a random number and the percentage passed to the constructor. If this random number indicates that the runner should run, the class should add the speed value passed to the constructor. The run method should sleep for 100 milliseconds on each repetition of the loop.
* If the run method is interrupted, it should display a message that concedes the race and quits.
* The main method of the application's main class should create two runner threads and start them. One of the threads should be named "Tortoise." It runs only 10 meters each move, but plods along without ever resting. The other thread should be named "Hare." It should run 100 meters each move, but should rest 90% of the time.
* This class should also include a method named finished that one of the threads can call when it finishes the race. That method should declare the thread that calls it to be the winner and should interrupt the other thread so it can concede the race.
* The finished method should provide for the possibility that the two threads will finish the race at almost the same time. If that happens, it should ensure that only one of the threads is declared the winner. (There are no ties!)
* Modify the main application class so that it runs the race 100 times and reports how many times each runner wins. (To make the application run faster, you may want to reduce the sleep time in the runner threads.)
* Modify the application so it can support up to 9 runners.
* Add an additional random element to the runner's performance. For example, have a "clumsiness percentage" that indicates how often the runner trips and hurts himself. When the runner trips, he sprains his or her ankle and can run only at half speed for the next five moves.
* Add the ability for runners to interfere with each other. For example, have an "orneriness percentage" that indicates how likely the runner is to trip another runner who is passing him. This will require additional communication among the threads.

**Hints**

* To determine whether a thread should run or rest, calculate a random number between 1 and 100. Then, have the thread rest if the number is less than or equal to the percentage of time that the thread rests. Otherwise, the thread should run.
* The finished method in the main application class will need to know which thread called it.

This application demonstrates the basics of creating independent threads that don't require much synchronization. Even so, you must coordinate those threads to declare one of the threads the winner and have the other thread concede.