Project: Database – Part 1

Deliverables

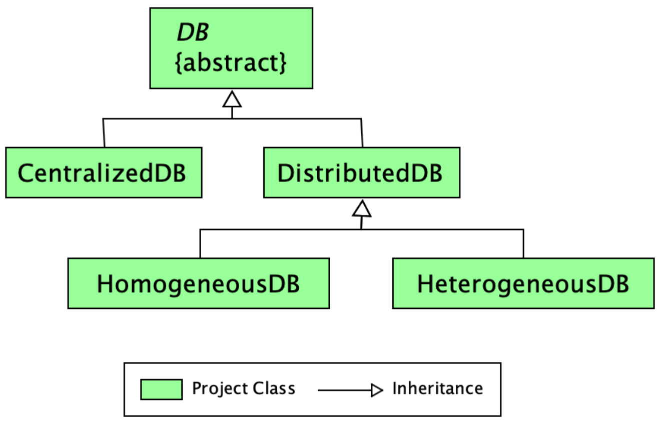
Your project files should be submitted to Web-CAT by the due date and time specified. Note that there is also an optional Skeleton Code assignment which will indicate level of coverage your tests have achieved (there is no late penalty since the skeleton code assignment is ungraded for this project). The files you submit to skeleton code assignment may be incomplete in the sense that method bodies have at least a return statement if applicable or they may be essentially completed files. In order to avoid a late penalty for the project, you must submit your completed code files to Web-CAT no later than 11:59 PM on the due date for the completed code assignment. If you are unable to submit via Web-CAT, you should e-mail your project Java files in a zip file to your TA before the deadline. The grades for the Completed Code submission will be determined by the tests that you pass or fail in your test files and by the level of coverage attained in your source files as well as usual correctness tests in Web-CAT.

Files to submit to Web-CAT:

- DB.java
- CentralizedDB.java, CentralizedDBTest.java
- DistributedDB.java, DistributedDBTest.java
- HomogeneousDB.java, HomogeneousDBTest.java
- HeterogeneousDB.java HeterogeneousDBTest.java
- (Optional) DBPart1.java, DBPart1Test.java

Specifications

Overview: This project is the first of three that will involve the monthly cost and reporting for database systems. You will develop Java classes that represent categories of a database including centralized database and distributed database (both homogeneous and heterogeneous). You may also want to develop an optional driver class with a main method. As you develop each class, you should create the associated JUnit test file with the required test methods to ensure the classes and methods meet the specifications. You should create a jGRASP project upfront and then add the source and test files as they are created. All of your files should be in a single folder. Below is the UML class diagram for the required classes which shows the inheritance relationships.



A **centralized database** is basically a type of database that is stored, located as well as maintained at a single location only. This type of database is modified and managed from that location itself. This location is thus mainly any database system or a centralized computer system. A **distributed database** is basically a type of database which consists of multiple databases that are connected with each other and are spread across different physical locations. The data that is stored on various physical locations can thus be managed independently of other physical locations. The communication between databases at different physical locations is thus done by a computer network. In a **homogeneous database**, all different sites store database identically. The operating system, database management system and the data structures used – all are same at all sites. Hence, they're easy to manage. In a **heterogeneous database**, different sites can use different schema and software that can lead to problems in query processing and transactions. Also, a particular site might be completely unaware of the other sites. Different computers may use a different operating system, different database application. [https://www.geeksforgeeks.org/difference-between-centralized-database-and-distributed-database/]

You should read through the remainder of this assignment before you start coding.

- **DB.java**

  Requirements: Create an *abstract* DB class that describes a DB and provides methods to access its descriptive data.

  Design: The DB class has fields, a constructor, and methods as outlined below.

  (1) **Fields**:
  instance variables (*protected*) for: (1) name of type String, (2) base cost of type double, and (3) database storage in terabytes (TB) of type double.
  class variable (*protected static*) of type int for the count of DB objects that have been created; set to zero when declared and then incremented in the constructor.
  These are the only fields that this class should have.

  (2) **Constructor**: The DB class must contain a constructor that accepts three parameters representing the instance variables (name, base cost, database storage) and then assigns them as appropriate. Since this class is abstract, the constructor will be called from the subclasses of DB using *super* and the parameter list. The count field should be incremented in the constructor.

  (3) **Methods**: Usually a class provides methods to access (or read) and modify each of its instance variables (known as get and set methods) along with any other required methods. At minimum you will need the following methods.
  - `getName`: Accepts no parameters and returns a String representing the name.
  - `setName`: Accepts a String representing the name, sets the field, and returns nothing.
  - `getBaseCost`: Accepts no parameters and returns a double representing base storage cost.
  - `setBaseCost`: Accepts a double representing the base storage cost, sets the field, and returns nothing.
  - `getDbStorage`: Accepts no parameters and returns a double representing database storage in terabytes.

- `setDbStorage`: Accepts a double representing the database storage in terabytes, sets the field, and returns nothing.
- `getCount`: Accepts no parameters and returns an int representing the count. Since count is *static*, this method should be *static* as well.
- `resetCount`: Accepts no parameters, resets count to zero, and returns nothing. Since count is *static*, this method should be *static* as well.
- `toString`: Returns a String describing the DB object. This method will be inherited by the subclasses. For an example of the toString result, see the CentralizedDB class and DistributedDB class below. Note that you can get the class name for an instance c by calling c.getClass() [or if inside the class, this.getClass()].
- `monthlyCost`: An *abstract* method that accepts no parameters and returns a double representing the monthly cost of the database.

Code and Test: Since the DB class is abstract you cannot create instances of DB upon which to call the methods. However, these methods will be inherited by the subclasses of DB. You should consider first writing skeleton code for the methods in order to compile DB so that you can create the first subclass described below. At this point you can begin completing the methods in DB and writing the JUnit test methods for your subclass that tests the methods in DB.

- **CentralizedDB.java**

Requirements: Derive the class CentralizedDB from DB.

Design: The CentralizedDB class has a field, a constructor, and methods as outlined below.

(1) **Field**: *instance* variable for license of type double. This variable should be declared with the *private* access modifier. This is the only field that should be declared in this class.

(2) **Constructor**: The CentralizedDB class must contain a constructor that accepts four parameters representing the three instance fields in the DB class (name, base cost, database storage) and the one instance field for license declared in CentralizedDB. Since this class is a subclass of DB, the super constructor should be called with field values for DB. The instance variable for license should be set with the last parameter. Below is an example of how the constructor could be used to create a CentralizedDB object:

```
CentralizedDB db1 = new CentralizedDB("Database One", 1200.0,
                                      5.00, 1500.0);
```

(3) **Methods**: Usually a class provides methods to access (or read) and modify each of its instance variables (known as get and set methods) along with any other required methods. At minimum you will need the following methods.

- `getLicense`: Accepts no parameters and returns a double representing license.
- `setLicense`: Accepts a double representing the license, sets the field, and returns nothing.
- `monthlyCost`: Accepts no parameters and returns a double representing the monthly cost for the CentralizedDB calculated as the sum of base cost and license.

- `toString`: Returns a String describing the CentralizedDB object by calling parent's toString method, super.toString(), which returns the first three lines, and then appending the line for license. Below is an example of the toString result for CentralizedDB db1 as it is declared above. *Note that database storage should use the DecimalFormat pattern* `"0.000"`, *whereas monthly cost, base cost, and license should use* `"$#,##0.00"`.

```
Database One (class CentralizedDB) Monthly Cost: $2,700.00
Storage: 5.000 TB
Base Cost: $1,200.00
License: $1,500.00
```

Code and Test: As you implement the CentralizedDB class, you should compile and test it as methods are created. Although you could use interactions, it should be more efficient to test by creating appropriate JUnit test methods. You can now continue developing the methods in DB (parent class of CentralizedDB). The test methods in CentralizedDBTest should be used to test the methods in both DB and CentralizedDB. Remember, a CentralizedDB object *is-a* DB object which means CentralizedDB inherited the instance methods and fields defined in DB. Therefore, you can create instances of CentralizedDB in order to test methods of the DB class. You may also consider developing DBPart1 (page 8) in parallel with this class to aid in testing.

- **DistributedDB.java**

Requirements: Derive the class DistributedDB from DB.

Design: The DistributedDB class has a field, a constructor, and methods as outlined below.

(1) **Fields:**
instance variables (*protected*): (1) number of users of type int and (2) cost per user of type double. These variables should be declared with the *protected* access modifier.
constant (*public static final*) COST_FACTOR of type double set to 1.1, which can be referenced as DistributedDB.COST_FACTOR. Note that this constant will require a Javadoc comment since it is *pubic*.
These are the only fields that should be declared in this class.

(2) **Constructor**: The DistributedDB class must contain a constructor that accepts five parameters representing the three instance fields in the DB class (name, base cost, database storage) and the two instance fields (number of users and cost per user) declared in DistributedDB. Since this class is a subclass of DB, the super constructor should be called with field values for DB. The instance variables for number of users and cost per user should be set with the last two parameters. Below is an example of how the constructor could be used to create a DistributedDB object:

```
DistributedDB db2 = new DistributedDB("Database Two", 2000.0,
                                       7.5, 100, 12.0);
```

(3) **Methods**: Usually a class provides methods to access (or read) and modify each of its instance variables (known as get and set methods) along with any other required methods. At minimum you will need the following methods.

- `getNumberOfUsers`: Accepts no parameters and returns an int representing the number of users.
- `setNumberOfUsers`: Accepts an int representing the number of users, sets the field, and returns nothing.
- `getCostPerUser`: Accepts no parameters and returns a double representing the cost per user.
- `setCostPerUser`: Accepts a double representing the cost per user, sets the field, and returns nothing.
- `userCost`: Accepts no parameters and returns a double representing the user cost, calculated as (number of users * cost per user).
- `getCostFactor`: Accepts no parameters and returns a double representing the cost factor. Although this method is returning a constant, it should not be *static* to ensure the cost factor for this class is returned of when called on an object of this class.
- `monthlyCost`: Accepts no parameters and returns a double representing the monthly cost for the DistributedDB object as follows:
  base cost + userCost() * DistributedDB.COST_FACTOR
- `toString`: Returns a String describing the DistributedDB object by calling parent's toString method, super.toString() and then appending the lines for number of users, cost per user, user cost, and cost factor. Be sure to call the getCostFactor method for the cost factor value. Below is an example of the toString results for DistributedDB db2 as it is declared above. *Note that database storage should use the DecimalFormat pattern* `"0.000"`, *whereas monthly cost, base cost, cost per user, and user cost should use* `"$#,##0.00"`.

```
Database Two (class DistributedDB) Monthly Cost: $3,320.00
Storage: 7.500 TB
Base Cost: $2,000.00
Number of Users: 100
Cost per User: $12.00
User Cost: $1,200.00
Cost Factor: 1.1
```

Code and Test: As you implement the DistributedDB class, you should compile and test it as methods are created. For example, as soon you have implemented and successfully compiled the constructor, you should create an instance of DistributedDB in a JUnit test method in the DistributedDBTest class and then run the test file. If you want to view your objects in the Canvas, set a breakpoint in your test method the run *Debug* on the test file. When it stops at the breakpoint, step until the object is created. Then open a canvas window using the canvas button at the top of the Debug tab. After you drag the instance onto the canvas, you can examine it for correctness. If you change the viewer to "toString" view, you can see the formatted toString value. You can also enter the object variable name in interactions and press ENTER to see the toString value. *Hint: If you use the same variable names for objects in the test methods, you can use the menu button on the viewer in the canvas to set "Scope Test" to "None". This will allow you to use the same canvas with multiple test methods.* You may also consider developing DBPart1 (page 8) in parallel with this class to aid in testing.

- **HomogeneousDB.java**

  Requirements: Derive the class HomogeneousDB from DistributedDB.

  Design: The HomogeneousDB class has a field, a constructor, and methods as outlined below.

  (1) **Field**: **constant (*public static final*)** COST_FACTOR of type double set to 1.2, which can be referenced as HomogeneousDB.COST_FACTOR outside of the class. Note that this constant will require a Javadoc comment since it is *pubic*.
  These are the only fields that should be declared in this class.

  (2) **Constructor**: The HomogeneousDB class must contain a constructor that accepts five parameters representing the three instance fields in the DB class (name, base cost, database storage) and the two instance fields (number of users and cost per user) declared in DistributedDB. Since this class is a subclass of DistributedDB, the super constructor should be called with field values for DB and DistributedDB; i.e., all five parameters. Below is an example of how the constructor could be used to create a HomogeneousDB object:

  ```
  HomogeneousDB db3 = new HomogeneousDB("Database Three", 2000,
                                        7.5, 100, 14.0);
  ```

  (3) **Methods**: Usually a class provides methods to access (or read) and modify each of its instance variables (known as get and set methods) along with any other required methods. At minimum you will need the following methods.
  - `getCostFactor`: Accepts no parameters and returns a double representing the cost factor. Although this method is returning a constant, it should not be *static* to ensure the cost factor for this class is returned of when called on an object of this class.
  - `monthlyCost`: Accepts no parameters and returns a double representing the monthly cost for the DistributedDB object as follows.
    base cost + userCost() * HomogeneousDB.COST_FACTOR
  - **There is no `toString` method in this class**. When toString is invoked on an instance of HomogeneousDB, the toString method inherited from DistributedDB is called. Below is an example of the toString result for HomogeneousDB db3 as it is declared above.

  ```
  Database Three (class HomogeneousDB) Monthly Cost: $3,680.00
  Storage: 7.500 TB
  Base Cost: $2,000.00
  Number of Users: 100
  Cost per User: $14.00
  User Cost: $1,400.00
  Cost Factor: 1.2
  ```

  Code and Test: As you implement the HomogeneousDB class, you should compile and test it as methods are created. For details, see **Code and Test** above for the CentralizedDB and DistributedDB classes. You may also consider developing DBPart1 (page 8) in parallel with this class to aid in testing.

- **HeterogeneousDB.java**

Requirements: Derive the class HeterogeneousDB from class DistributedDB.

Design: The HeterogeneousDB class has a field, a constructor, and methods as outlined below.

(1) **Field**: **constant (*public static final*)** COST_FACTOR of type double set to 1.3, which can be referenced as HeterogeneousDB.COST_FACTOR outside the class. Note that this constant will require a Javadoc comment since it is *pubic*.
This is the only field that should be declared in this class.

(2) **Constructor**: The HeterogeneousDB class must contain a constructor that accepts four parameters rep**r**esenting the two instance fields in the DB class (name and base storage cost) and the two instance fields in the DistributedDB (number of users and cost per user). Below is an example of how the constructor could be used to create a HeterogeneousDB object:

```
HeterogeneousDB db4 = new HeterogeneousDB("Database Four", 2000,
                                           7.5, 100, 14.0);
```

(3) **Methods**: Usually a class provides methods to access (or read) and modify each of its instance variables (known as get and set methods) along with any other required methods. At minimum you will need the following methods.

- o `getCostFactor`: Accepts no parameters and returns a double representing the cost factor. Although this method is returning a constant, it should not be *static* to ensure the cost factor for this class is returned of when called on an object of this class.
- o `monthlyCost`: Accepts no parameters and returns a double representing the monthly cost for the DistributedDB object as follows.
  base cost + userCost() * HeterogeneousDB.COST_FACTOR
- o **There is no `toString` method in this class**. When toString is invoked on an instance of HeterogeneousDB, the toString method inherited from DistributedDB is called. Below is an example of the toString result for HeterogeneousDB db4 as it is declared above.

```
Database Four (class HeterogeneousDB) Monthly Cost: $3,820.00
Storage: 7.500 TB
Base Cost: $2,000.00
Number of Users: 100
Cost per User: $14.00
User Cost: $1,400.00
Cost Factor: 1.3
```

Code and Test: As you implement the HeterogeneousDB class, you should compile and test it as methods are created. For details, see **Code and Test** above for the CentralizedDB and DistributedDB classes. You may also consider developing DBPart1 (below) in parallel with this class to aid in testing.

- **DBPart1.java (Optional)**

Requirements: Driver class with main method is optional, but you may find it helpful.

Design: The DBPart1 class only has a main method as described below.

The main method should be developed incrementally along with the classes above. For example, when you have compiled DB and CentralizedDB, you can add statements to main that create and print an instance of CentralizedDB. [Since DB is abstract you cannot create an instance of it.] When main is completed, it should contain statements that create and print instances of CentralizedDB, DistributedDB, HomogeneousDB, and HeterogeneousDB. Since printing the objects will not show all of the details of the fields, you should also run DBPart1 in the canvas (or debugger with a breakpoint) to examine the objects. Between steps you can use interactions to invoke methods on the objects in the usual way. For example, if you create db1, db2, db3, and db4 as described in the sections above and your main method is stopped between steps after db4 has been created, you can enter the following in interactions to get the monthly cost and user cost for the HeterogeneousDB object.

```
▶ db4.monthlyCost()
  3820.0
▶ db4.userCost()
  1400.0
```

The output from main assuming you create and print db1, db2, db3, and db4 as described in the sections above is shown as below. Note these were printed using println (rather than print) and a new line was added in main before each object to achieve the spacing below.

```
Database One (class CentralizedDB) Monthly Cost: $2,700.00
Storage: 5.000 TB
Base Cost: $1,200.00
License: $1,500.00

Database Two (class DistributedDB) Monthly Cost: $3,320.00
Storage: 7.500 TB
Base Cost: $2,000.00
Number of Users: 100
Cost per User: $12.00
User Cost: $1,200.00
Cost Factor: 1.1

Database Three (class HomogeneousDB) Monthly Cost: $3,680.00
Storage: 7.500 TB
Base Cost: $2,000.00
Number of Users: 100
Cost per User: $14.00
User Cost: $1,400.00
Cost Factor: 1.2
```

```
Database Four (class HeterogeneousDB) Monthly Cost: $3,820.00
Storage: 7.500 TB
Base Cost: $2,000.00
Number of Users: 100
Cost per User: $14.00
User Cost: $1,400.00
Cost Factor: 1.3
```

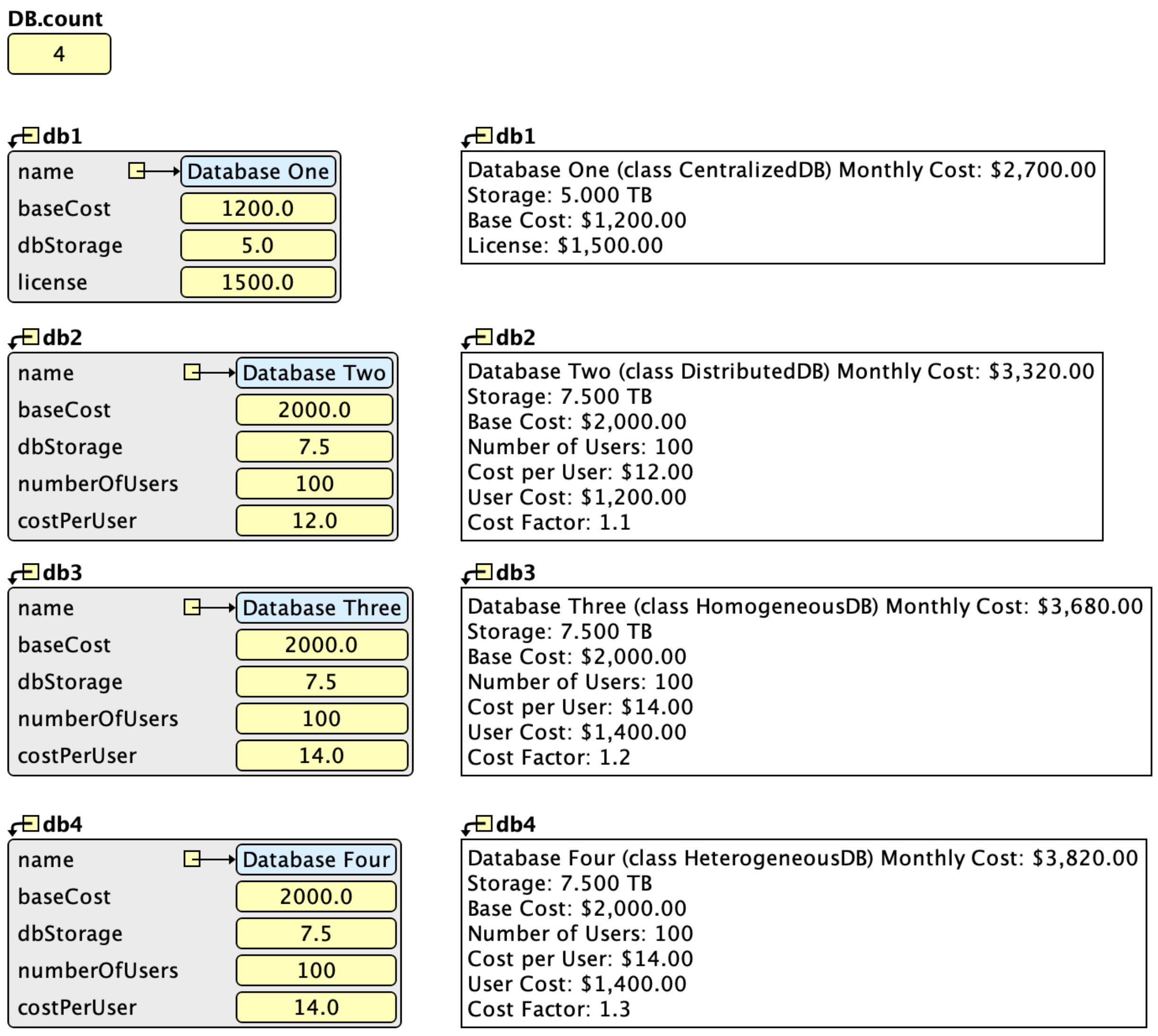
Code and Test: After you have implemented the DBPart1 class, you should create the test file DBPart1Test.java in the usual way. The only test method you need is one that checks the class variable *count* that was declared in DB and inherited by each subclass. In the test method, you should reset *count*, call your main method, then assert that *count* is five (assuming that your main creates five objects from the DB hierarchy). The following statements accomplish the test.

```
      DB.resetCount();
      DBPart1.main(null);
      Assert.assertEquals("DB count should be 5. ",
                          4, DB.getCount());
```

See the Canvas for DBPart1 on the next page.

Canvas for DBPart1

On the is an example of a jGRASP viewer canvas for DBPart1 that contains a viewer for the class variable DB.count and two viewers for each of db1, db2, db3, and db4. The first viewer for each is set to Basic viewer and the second is set to the toString viewer. The canvas was created dragging instances from the debug tab into a new canvas window and setting the appropriate viewer. Note that you will need to unfold one of the instances in the debug tab to find the static variable *count*.



DB.count

4

db1

name	Database One
baseCost	1200.0
dbStorage	5.0
license	1500.0

db1

```
Database One (class CentralizedDB) Monthly Cost: $2,700.00
Storage: 5.000 TB
Base Cost: $1,200.00
License: $1,500.00
```

db2

name	Database Two
baseCost	2000.0
dbStorage	7.5
numberOfUsers	100
costPerUser	12.0

db2

```
Database Two (class DistributedDB) Monthly Cost: $3,320.00
Storage: 7.500 TB
Base Cost: $2,000.00
Number of Users: 100
Cost per User: $12.00
User Cost: $1,200.00
Cost Factor: 1.1
```

db3

name	Database Three
baseCost	2000.0
dbStorage	7.5
numberOfUsers	100
costPerUser	14.0

db3

```
Database Three (class HomogeneousDB) Monthly Cost: $3,680.00
Storage: 7.500 TB
Base Cost: $2,000.00
Number of Users: 100
Cost per User: $14.00
User Cost: $1,400.00
Cost Factor: 1.2
```

db4

name	Database Four
baseCost	2000.0
dbStorage	7.5
numberOfUsers	100
costPerUser	14.0

db4

```
Database Four (class HeterogeneousDB) Monthly Cost: $3,820.00
Storage: 7.500 TB
Base Cost: $2,000.00
Number of Users: 100
Cost per User: $14.00
User Cost: $1,400.00
Cost Factor: 1.3
```