5. Lab E – Bridge

Purpose

In this lab we will demonstrate how the bridge design pattern may be used.

We will implement a bridge between 2 hierarchies:

1. Data storage hierarchy (like hash map) whose base interface is the DataMap interface. Implementations of this interface know how to utilize a Map to in-memory store data.
2. Data access hierarchy acting upon a given storage (like readOnlyAccess, UniqueKeyAccess, and etc...). At the the top of this hierarchy is the AbstractDBTable class that uses an underlying implementation of the above DataMap to store data.

We <u>separate the data logic from the data implementation</u>. This allows us to expose the same data to different clients with different functionality.

Review of the Bridge

The flexibility gained by this design pattern in this exercise is manifested by the fact that any AbstractDBTable subclass can use any DataMap implementation as it's underlying data storage mechanism. To clear things out, we wish to bridge 2 inheritances hierarchies:

1. The implementation is a storage hierarchy (how we store objects).
2. The abstraction is the access hierarchy (how we access data).

The implementation has 2 classes (one interface and one implementation):

1. DataMap serves as base for data storage implementations
2. HashDataMap implements DataMap and reflects an in memory hash map storage

The abstraction has 4 classes (one abstract and three implementations)

1. AbstractDBTable base abstract class for the abstraction hierarchy
2. NormalDBTable allows users to read, update and select data
3. ReadOnlyDBTable allows users to read data preventing write and updates
4. UniqueKeyDBTable prevents duplicate values

5.1.1. Create a new interface – DataMap

Serves as a base for concrete data storage classes (in memory, file, and etc...).

1. Create the interface DataMap
2. Code the following methods

```
public V get(K key);

public boolean keyExists(K key);

public void put(K key, V value);
```

Note: you can make the entire hierarchy generic by implementing this as a generic interface as follows:

```
public interface DataMap<K, V> {
    public V get(K key);
    public boolean keyExists(K key);
    public void put(K key, V value);
}
```

Implementing classes could then look like:

```
public class HashDataMap<K, V> implements DataMap<K, V>{
    private Map<K, V> dataMap;
    . . .
}
```

5.1.2. Create a new class HashDataMap

This class represents a data storage. Information will be saved in memory to an embedded (composition) hash map.

1. Create the class HashDataMap
2. Extend DataMap
3. Add a Map<K,V> member variable

```java
private Map<K, V> dataMap;
```

4. Implement the methods inherited from DataMap delegating the calls to the embedded hashmap

```java
@Override
public void put(K key, V value) {
    dataMap.put(key, value);
}


@Override
public V get(K key) {
    return dataMap.get(key);
}


@Override
public boolean keyExists(K key) {
    return dataMap.containsKey(key);
}
```

5.1.3. Create a new abstract class AbstractDBTable

This class will serve as Base for the data access hierarchy of classes.

1. Create the class AbstractDBTable
2. Code a member variable of type DataMap

```
private DataMap<PK, T> dataMap;
```

3. Code the constructor to take a DataMap object as param and save it to the global variable

```
public AbstractDBTable(DataMap<PK, T> dataMap) {
   super();
   this.dataMap = dataMap;
}
```

4. Code the following abstract methods

```
public abstract void insert(PK id, T data);

public abstract T select(PK id);

public abstract void update(PK id, T data);
```

5. Code a protected getImpl methods returning the DataMap object

```
protected DataMap<PK, T> getImpl() {
   return this.dataMap;
}
```

This method will be called by classes extending to get access to the DataMap object .

5.1.4. Create a class NormalDBTable

The class reflects a data access object permitting all 3 operations – put, get and keyExists.

1. Create class NormalDBTable and extend AbstractDBTable
2. Implement the methods insert, update and select with delegation to the DataMap object :

```
@Override
public void insert(PK id, T data) {
     getImpl().put(id, data);
}

@Override
public void update(PK id, T data) {
     if (getImpl().keyExists(id))
          insert(id, data);
}

@Override
public T select(PK id) {
     return getImpl().get(id);
}
```

5.1.5. Create a new class ReadOnlyDBTable

The class extends AbstractDBTable. It allows only read access and prevents any write / update by throwing a runtime exception.

1. Create the class ReadOnlyDBTable and extends AbstractDBTable
2. Select will delegate to DataMap, whereas insert and update will be prevented by throwing a run time exception, thus, preventing any modifications to data

```
@Override
public void insert(PK id, T data) {
     throw new UnsupportedOperationException("unsupported");
}

@Override
public void update(PK id, T data) {
    throw new UnsupportedOperationException("unsupported");
}

@Override
public T select(PK id) {
     return getImpl().get(id);
}
```

5.1.6. Create a new class UniqueKeyDBTable

The class extends AbstractDBTable and prevents duplicates. Trying to add an existing value will result in a run time exception.

1. Create the class UniqueKeyDBTable and extends AbstractDBTable
2. Insert will check if the value exists before storing it.
   If it does exist, an exception is thrown.
   Select and update delegate to DataMap as in NormalDBTable.

```
@Override
public void insert(PK id, T data) {
     if (!getImpl().keyExists(id))
         getImpl().put(id, data);
}
```

Note: The solution for this exercise is available in the ‘solutions’ directory.