# Programming Assignment 1: Multiple and Repeated Inheritance

Faith-Anne L. Kocadag

CSCI 3330 Comparative Languages, Section 1
Submitted to: Dr. Reed

February 14, 2013

## I. Introduction

Inheritance in object-oriented programming allows for the derivation of new classes from existing classes [1]. The most notable difference between C++ and Java is that C++ allows for multiple and repeated inheritance. That is, a subclass can be defined in C++ that inherits non-private functions and variables from more than one base class. Java, on the other hand, is limited to single inheritance. This project explores the advantages and hazards of multiple inheritance in C++ and endeavors to recreate a similar structure in Java through the use of abstract classes and interfaces.

The source code for this project can be found in Appendix A. The UML Diagrams and program designs for both the C++ and Java programs are located in the following section. Screenshots of test cases appear in Sections IVA and IVB, and a discussion question is examined in Section V.

## IIA. Program Design: C++

C++ allows multiple inheritance, which is achieved through careful class construction involving virtual inheritance. For example, the ContactList class is a subclass of both Dictionary and AssociativeMemory. Figure 1 demonstrates how this relationship is defined in the class declarations. The only functions defined below it are those specific to the ContactList class.

```
class ContactList : public Dictionary, public AssociativeMemory
{
public:
    ContactList();
    void setLock();
    void unsetLock();
    void print();
protected:
    bool lock;
};
```

Figure 1. ContactList.h class declarations

Because of these inheritance relationships (also called parent-child or “is-a” relationships), ContactList inherits the non-private variables and functions that belong to Dictionary, as well as those that belong to AssociativeMemory. In layman’s terms, one says that a ContactList “is-a” Dictionary and a ContactList “is-an” AssociativeMemory. Anything one can do with a Dictionary, one can do with a ContactList. Likewise, anything one can do with an AssociativeMemory, again, one can do it with a ContactList. Since many entities in real life have “is-a” relationships with more than one base entity (e.g. a smartphone “is-a” music player and a smartphone “is-a” phone), this is a straightforward concept.

While these relationships are clearly defined in C++, there is a potential pitfall called the “Deadly Diamond of Derivation” that one must be aware of in multiple inheritance situations. While ContactList “is-a” Dictionary and ContactList “is-an” AssociativeMemory, it is also true that a Dictionary “is-a” Map and an AssociativeMemory “is-a” Map. Both AssociativeMemory and Dictionary are subclasses of a base class called Map, and all three have their own versions of a function called print(). The UML Diagram in Figure 6 reveals this diamond shape.

To explain why this may cause a potential problem, first some background information. C++ and Java both allow one to reference an object through their class or their base class. This is akin to asking for a phone and being brought a smartphone, a flip phone, a rotary phone, or any other type of object of the base class phone. To bring it back to the current program, one could call for an array of Map objects and place anything that “is-a” Map within it, as in Figure 2.

```
Map *array[4];                          // Create array of Base type Map
array[0] = &myMap;
array[1] = &myDictionary;
array[2] = &myMemory;
array[3] = &myContacts;
```

Figure 2. Array of Map objects

Also, it is possible for superclasses to override, or redefine, functions defined in their base classes. In this instance, Map, Dictionary, AssociativeMemory, and ContactList all have their own unique implementations of the print() function. This is called polymorphism—many versions of the same thing. When the Map objects in Figure 2 each call print(), the compiler is confused at the ambiguity. This Deadly Diamond causes an error that can be fixed through the implementation of virtual inheritance.

```
class AssociativeMemory : virtual public Map    // Virtual base class avoids
{                                               // Deadly Diamond
public:
    AssociativeMemory();
    string getKey(string val);
    bool isVal(string val);
    void print();                               // Virtual keyword is not necessary
protected:                                      // for overridden subclass functions
    string valCode;
};
```

Figure 3. AssociativeMemory.h and the virtual  base class Map

```
#include "Map.h"
#include <map>
#include <iostream>
using namespace std;

class Dictionary : virtual public Map   // Virtual base class avoids
{                                       // Deadly Diamond
public:
    Dictionary();
    string getVal(string key);
    void print();                       // Virtual keyword is not necessary
protected:                              // for overridden subclass functions
    string nameCode;
};
```

Figure 4. Dictionary.h and the virtual base class Map

In Figure 3, the AssociativeMemory class extends a virtual Map. The Dictionary.h class uses the virtual keyword in the same way (Figure 4). When creating an object, the program instantiates the base classes, then the derived classes, then lastly the declared class in question. In a Deadly Diamond structure, an instantiation of the lowest subclass (like ContactList) would create two instances of the highest base class (Map) that it inherited from Dictionary and Associative Memory. This causes great confusion to the compiler in cases where there is uncertainty in choosing the exact implementation of a polymorphic function. This situation is clarified by the addition of the virtual keyword in the base class at the polymorphic function, as in Figure 5. Virtual inheritance ensures that only a single copy of a base class exists in the entire derivation lattice" [2] and clarifies which print() function is appropriate to use in an otherwise ambiguous situation.

Not all library classes are compatible with virtual inheritance (they lack virtual keywords as appropriate), and therefore would not allow for multiple inheritance structures such as this.

```
class Map
{
public:
    Map();
    void insert(string key, string val);
    void del(string key);
    bool isMapped(string key) const;
    virtual void print();   // Declared virtual in this base class to avoid
                            // Deadly Diamond. Also, not const because iterator
                            // p is not constant.
protected:                  // Protected variables are available to subclasses
    map<string, string> map1;
    map<string, string>::iterator p;
};
```

Figure 5. Map.h  uses the virtual keyword to define the print function

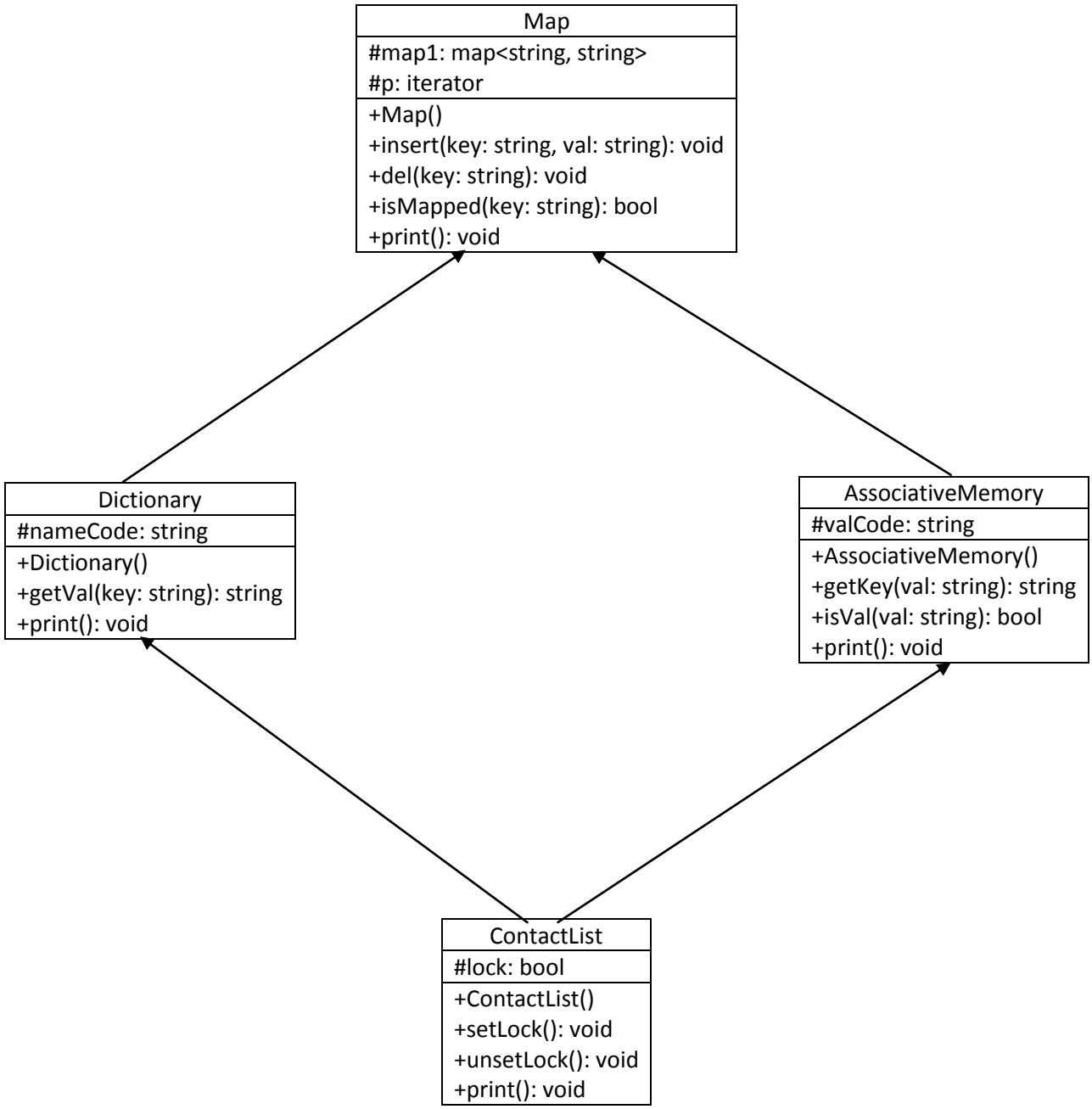


Figure 6. UML Diagram: C++ Class Design

## IIB. Program Design: Java

Because Java only allows for single inheritance relationships, it was a challenge to implement the same class structure as in the C++ program. Concessions had to be made to create a system of classes that would perform the same functions as their C++ counterparts. While many consider using multiple interfaces in Java to simulate multiple inheritance relationships, this requires every class that implements these interfaces to also explicitly define every abstract method of each interface.

Like in the C++ program, the Java program began with the base class Map. A MapInterface was considered, but ultimately discarded for reasons outlined in Test Case IVB.2. Originally two subclasses (Dictionary and AssociativeMemory) were created to directly extend Map, just as in the C++ program. The problem came when trying to define the ContactList. To properly mimic the C++ program, the ContactList class had to employ all of the features of both the AssociativeMemory and Dictionary classes.

In this program, ContactList extended AssociativeMemory. Dictionary would have been a roughly equivalent choice. AssociativeMemory had one variable (valCode) and two methods (getKey(String) and isVal(String)) that ContactList needed to inherit. print() was unimportant, as ContactList needed to override whatever version of print() it inherited in favor of its own implementation. Dictionary also had one variable (nameCode), but only one method (getVal(String)) to pass down to ContactList. Since AssociativeMemory had one more method for ContactList to inherit, it was chosen as ContactList's sole direct superclass.

There was still the problem of how to pass on Dictionary's variable and getVal(String) method to ContactList without explicitly recreating them in the ContactList class. Since Dictionary's variable (nameCode) was static (never expected to change), it could be defined in an interface called DictionaryInterface. A Java interface is a sort of contract that defines the constant variables and method signatures to which all classes that implement said interface must adhere. According to Tucker and Noonan, "an interface is similar to multiple inheritance in the sense that an interface is a type" [3]. The nameCode variable was defined in DictionaryInterface and the Dictionary class and ContactList class both implement it.

```java
/* DictionaryInterface defines the required methods and variables for classes
 * that implement it */
public interface DictionaryInterface {

    // Variable is available to all classes that implement DictionaryInterface
    static String nameCode = "Names";

}
```

Figure 7. DictionaryInterface

```
public class ContactList extends AssociativeMemory implements DictionaryInterface {
```

Figure 8. ContactList defined

The signature of Dictionary's getVal(String) method could have been defined in DictionaryInterface, but not the actual working method itself. Defining getVal(String) in DictionaryInterface would have obligated the classes that implement it (Dictionary and ContactList) to separately include their own implementations of getVal(String). In that case, ContactList's getVal() would not be the same method inherited from Dictionary, even if the contents on both were identical. To prevent having to define a getVal(String) method in ContactList, it was added to AbstractDictionary.

A solution to this problem came in the form of an abstract class called AbstractDictionary. An abstract class is one that is cannot be initialized by a client program [4]. When an abstract class has no abstract methods, it "represents an abstract concept which unifies several concrete classes" [4]. AbstractDictionary defines the getVal(String) method that Dictionary requires. Dictionary is made a child of AbstractDictionary instead of Map and granted access to this method. AbstractDictionary, in turn, would be a direct subclass of Map. Making AbstractDictionary a base class for both Dictionary and AssociativeMemory gives both Dictionary and ContactList access to the getVal(String) method. This also has the unintended consequence of allowing AssociativeMemory access to getVal(String), a situation explored in Test Case IVB.3.

ContactList was not made a child class of AbstractDictionary because, again, it cannot extend both AbstractDictionary and AssociativeMemory. Test Cases IVB4-5 unambiguously illustrate which methods and variables are accessible to objects of each class. The UML Diagram for this Java program is illustrated in Figure 9 (next page).

## IIIA. C++ Program Implementation

(i) The Map Class

The Map class utilizes the STL map and iterator classes. For greater control and better understanding of the effects of Deadly Diamond structures, the STL map class was not used as a base class for this Map class. The Map class took advantage of the STL map class' predefined functions for insertion (insert(string, string)), deletion (erase(string)), and finding elements (find(string)). The efficient STL iterator was useful in this Map's class (as well as in all of its derived classes), especially in the print() functions. The print() function was made virtual to prevent ambiguity in instantiations of the print() functions of its base classes. All functions were made public but the STL map and iterator objects were made protected. The protected modifier made those objects available to its subclasses. Only the

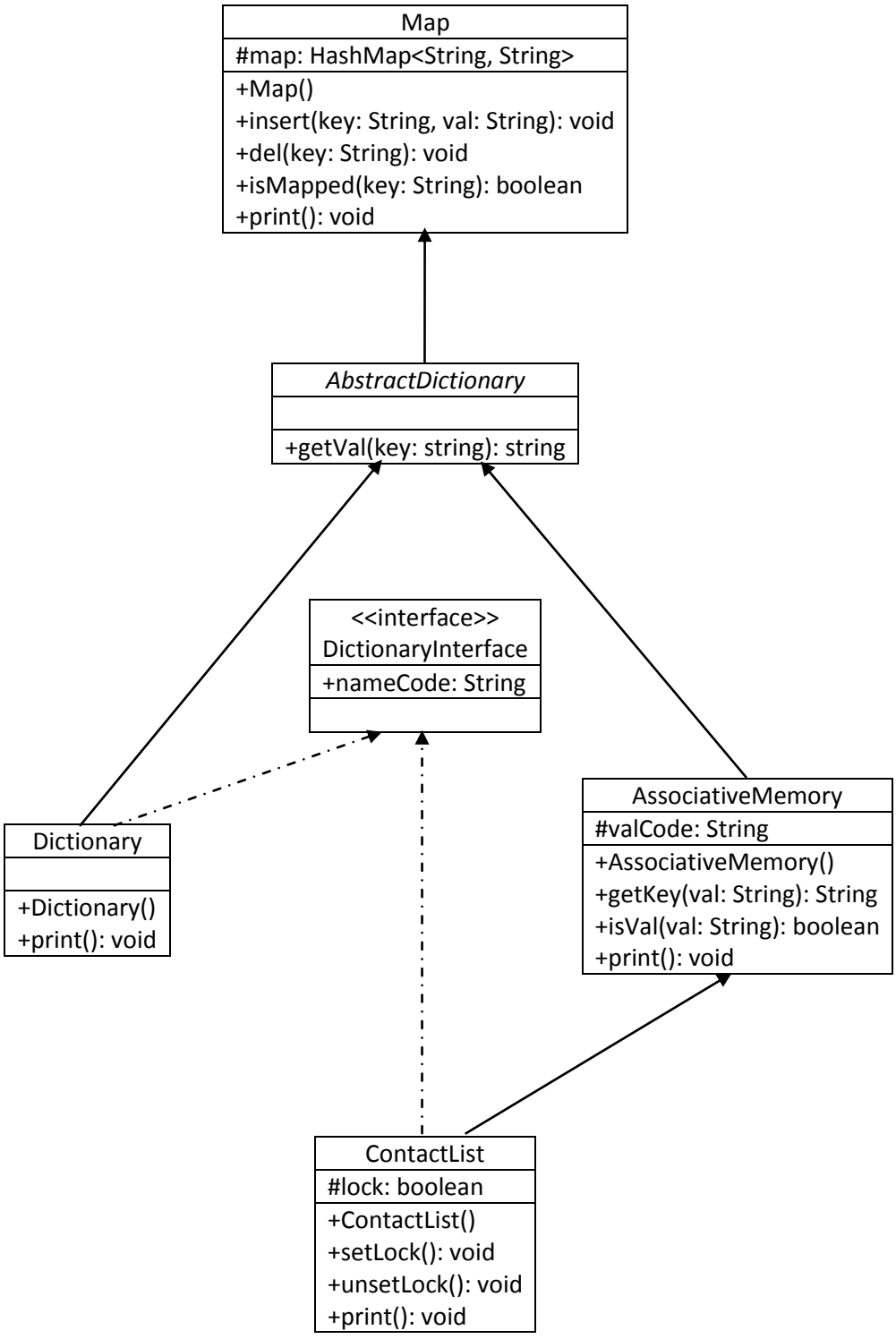


Figure 9. UML Diagram: Java Class Design

isMapped(string) function is constant, as insertions, deletions, and moving pointers in print functions do not preserve all of the variables/objects they use

(ii) The Dictionary Class

The Dictionary class virtually inherits Map. It very simply defines a nameCode string and implements a getVal(string) function and a print() function that overrides the one found in Map. Like in Map, it utilizes the STL iterator in its print() function and the predefined STL map find(String) function. All functions are public, and nameCode is protected. No functions are constant, as the pointers change values in each function.

(iii) The AssociativeMemory Class

The AssociativeMemory class also virtually inherits Map. It features a protected valCode string and public methods for retrieving keys, determining whether or not values exist in the AssociativeMemory, and printing out the contents of the AssociativeMemory. Since no preexisting functions exist in STL map that return a key given a value or return a Boolean given a value, the STL iterator was implemented to traverse each element for inspection in the getKey(String) and isVal(String) functions. All three non-constructor functions implement the pointer, so none of the functions are declared constant.

(iv) The ContactList Class

The ContactList class extends both AssociativeMemory and Dictionary. Both AssociativeMemory and Dictionary virtually extend Map to prevent the ambiguity of repeated inheritance caused by this diamond structure. The boolean variable lock is protected. The only functions defined in ContactList are setLock(), unsetLock(), and print(), none of which are constant.

(v) TestProject1.cpp

The test program, TestProject1 instantiates each of the above classes and fills them with string pairs representing names and numbers. The names are simply letters A-E, and the numbers are randomly generated 10 digit number strings from the generateNumber() function. The 10 digit numbers were generated five digits at a time (as 10 digit numbers caused overflow), converted to stringstreams, concatenated, and converted to strings. Each object (myMap, myDictionary, myMemory, and myContacts) was filled with five records each.

An array of Map objects was created, and each object reference was placed inside. A print() was performed on each, with the test program evidencing the polymorphism of this multi-inherited class structure.

## IIIB. Java Program Implementation

(i) The Map Class

The Map class uses java.util.HashMap as a client. The HashMap<String, String> object, simply called map, is protected for the use by Map's subclasses. This HashMap collects pairs of Strings representing keys and values. The Map class utilizes java.util.Map's predefined methods whenever possible. The insert(String, String) method takes advantage of Map's put(String, String) method, the del(String) uses the remove(String) method, and isMapped(String) uses containsKey(String). The print() method creates a Set of java.util.Map.Entry objects through java.util.Map's entrySet() method, which is then iterated through for printing. HashMaps do not have iterators, which is why the conversion was necessary. Iterators are much more efficient than regular for loops for traversing a data structure. The only drawback of converting to a Set for printing is that the elements become randomly ordered. One could easily cast the entry set to another structure for sorting if it were deemed important.

(ii) The AbstractDictionary Class

The AbstractDictionary class extends Map. Its only method is getVal(String), which, when given a key, returns a value through the use of java.util.Map's get(String) method. This class makes the getVal(String) method inheritable to its subclasses.

(iii) The Dictionary Class

The Dictionary Class extends AbstractDictionary and implements DictionaryInterface. From AbstractDictionary it inherits the getVal(String) method, and from the DictionaryInterface it gets the nameCode variable. The only method defined in Dictionary is print(), which overrides the print() in the Map class. It's print() method is very similar to the Map.print() method in its execution.

(iv) The DictionaryInterface Interface

DictionaryInterface defines the static nameCode variable. The String nameCode could be defined here because it will not change in the implementation of the classes that implement DictionaryInterface. Because both Dictionary and ContactList implement DictionaryInterface, both classes have access to nameCode. AssociativeMemory is rightly blind to this variable, as it does not implement this interface.

(v) The AssociativeMemory Class

The AssociativeMemory class extends AbstractDictionary. In each of its methods (getKey(String), isVal(String), and print()) it creates entry sets that are iterated through to search for specific values or to print the AssociativeMemory's contents. No java.util.Map methods performed these tasks directly, so this iterative approach was adopted.

Even though AssociativeMemory had no direct need of AbstractDictionary's getVal(String) method, it was made a child of AbstractDictionary so that its own subclass ContactList could inherit it.

(vi) The ContactList Class

The ContactList class extends AssociativeMemory and implements DictionaryInterface. It inherits all of AssociativeMemory's non-private variables and methods, as well as the non-private variables and

methods in AssociativeMemory's parent classes (AbstractDictionary and Map). From its implementation of DictionaryInterface it also inherits the nameCode variable, thus mimicking the full multiple-inheritance accessibility found in the C++ version of the ContactList class. This class has one variable, a boolean called lock, and methods to set and unset this lock. Like in the previous classes, the print() method creates an entry set and employs an iterator to print each key/value pair.

(vii) TestProject1.java

The test program, TestProject1, creates objects of each concrete class type: Map, AssociativeMemory, Dictionary, and ContactList. It fills each with five sets of String pairs representing keys/names and values/phone numbers. Keys are letters A-E, and phone numbers are randomly generated through the generateNumber() method. The generateNumber() method creates a 10 digit number of type long and casts it to a String.

Once each object is filled with its key/value pairs, they are placed in an array of Map objects. One by one polymorphic print() calls are made for the myMap, myDictionary, myMemory, and myContacts variables. The differing outputs of each demonstrate the effectiveness of overriding method calls in this single inheritance language.

## IVA. C++ Screen Shots, Test Cases, and Errors

Case 1: The error-free test case output is below.

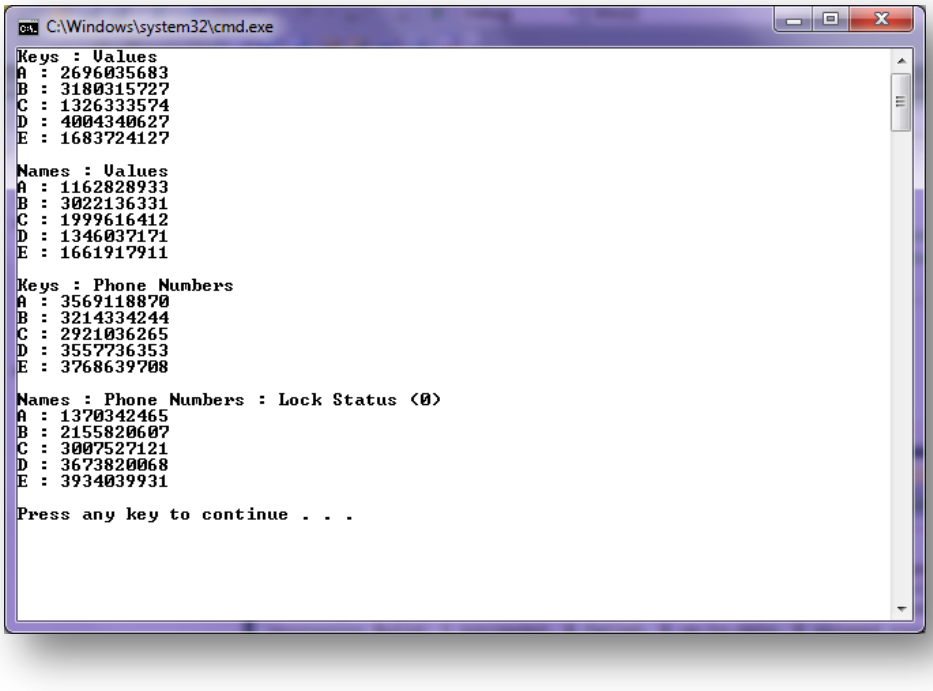


Figure 10. TestProject1.cpp output

Case 2: Removal of the default constructor for Map, although it was empty, caused LINK errors when trying to build the derived classes:

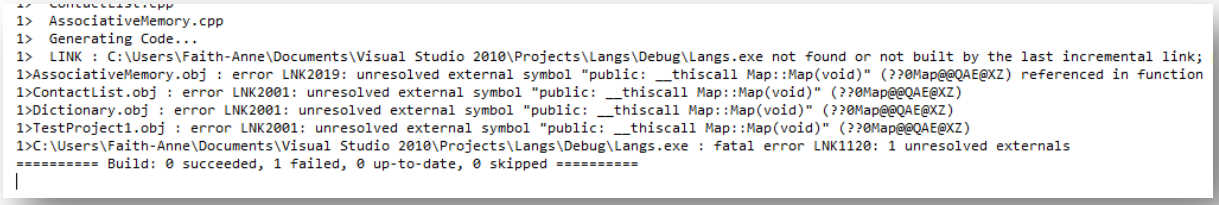


Figure 11. LINK error message

Case 3: Removal of the virtual keyword in the Dictionary header (as below) caused ambiguity when the insert function was called for myContacts, and when the myContacts pointer tried to insert into the Map array.

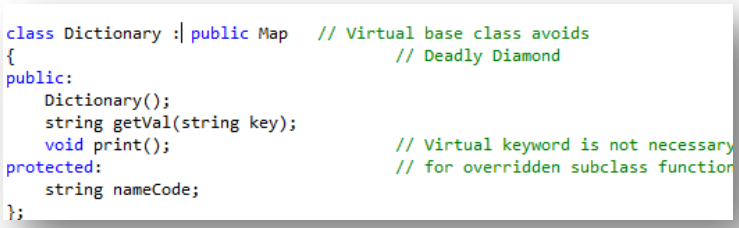


Figure 12. Dictionary.h: Removal of virtual keyword

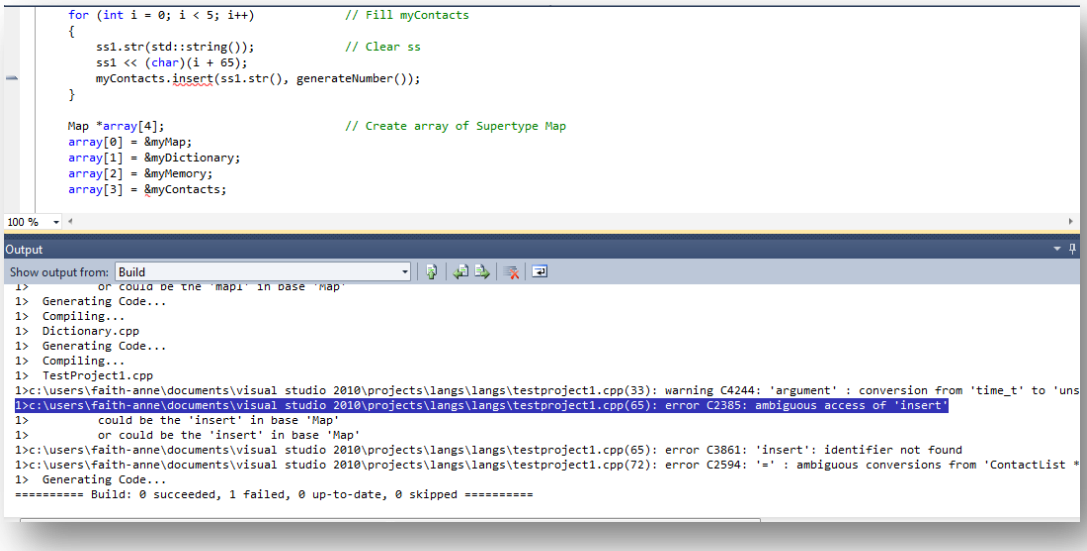


Figure 13. Console error: ambiguous insert call

Case 4: Removal of the virtual keyword from the print() definition in the Map header file did not cause any sematic  or compile errors, however none of the overridden print() functions were implemented. print() was no longer a polymorphic call, as all of the derived classes called on Map's print().

```
public:
    Map();
    void insert(string key, string val);
    void del(string key);
    bool isMapped(string key) const;
    void print();   // Declared virtual in this base
                            // Deadly Diamond. Also,
                            // p is not constant.
protected:                  // Protected variables ar
    map<string, string> map1;
```

Figure 14. Map.h: removal of virtual keyword from print()

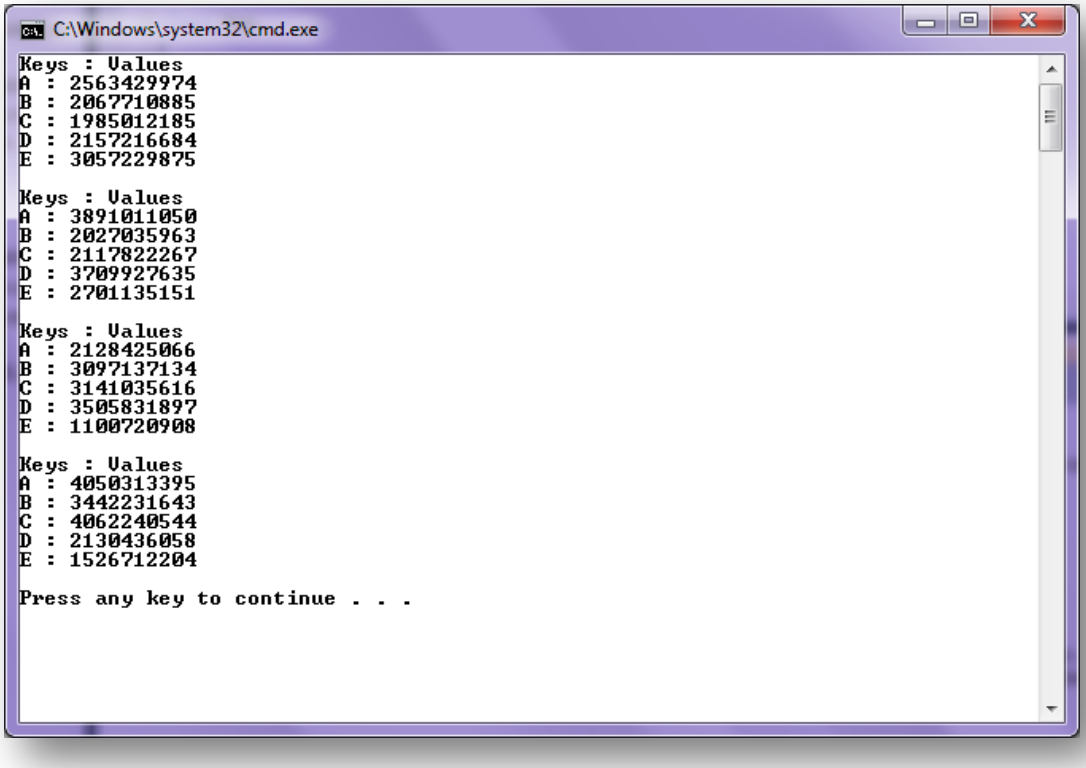


Figure 15. A non-polymorphic print() call

Case 5: These are the compile errors that occur when trying to access each method from objects of all 4 types in a Map array.

```
Map *array[4];
array[0] = &myMap;
array[1] = &myDictionary;
array[2] = &myMemory;
array[3] = &myContacts;

for (int i = 0; i < 4; i++)
{
    array[i] -> insert("String","String");
    array[i] -> del("String");
    array[i] -> isMapped("String");
    array[i] -> getVal("String");
    array[i] -> getKey("String");
    array[i] -> isVal("String");
    array[i] -> setLock();
    array[i] -> unsetLock();
```

100 %

Output

Show output from: Build

```
1>c:\users\faith-anne\documents\visual studio 2010\projects\langs\langs\testproject1.cpp(33): warning C4244: 'argument' : conversion from 'time_t' 
1>c:\users\faith-anne\documents\visual studio 2010\projects\langs\langs\testproject1.cpp(79): error C2039: 'getVal' : is not a member of 'Map'
1>          c:\users\faith-anne\documents\visual studio 2010\projects\langs\langs\map.h(10) : see declaration of 'Map'
1>c:\users\faith-anne\documents\visual studio 2010\projects\langs\langs\testproject1.cpp(80): error C2039: 'getKey' : is not a member of 'Map'
1>          c:\users\faith-anne\documents\visual studio 2010\projects\langs\langs\map.h(10) : see declaration of 'Map'
1>c:\users\faith-anne\documents\visual studio 2010\projects\langs\langs\testproject1.cpp(81): error C2039: 'isVal' : is not a member of 'Map'
1>          c:\users\faith-anne\documents\visual studio 2010\projects\langs\langs\map.h(10) : see declaration of 'Map'
1>c:\users\faith-anne\documents\visual studio 2010\projects\langs\langs\testproject1.cpp(82): error C2039: 'setLock' : is not a member of 'Map'
1>          c:\users\faith-anne\documents\visual studio 2010\projects\langs\langs\map.h(10) : see declaration of 'Map'
1>c:\users\faith-anne\documents\visual studio 2010\projects\langs\langs\testproject1.cpp(83): error C2039: 'unsetLock' : is not a member of 'Map'
1>          c:\users\faith-anne\documents\visual studio 2010\projects\langs\langs\map.h(10) : see declaration of 'Map'
========== Build: 0 succeeded, 1 failed, 0 up-to-date, 0 skipped ==========
```

Figure 16. Compile and runtime errors for Map array

Case 6: These are the compile errors that occur when trying to access each method from objects of all 4 types in a Dictionary array.

(Global Scope) main()

```
Dictionary *array[4];
array[0] = dynamic_cast<Dictionary *> (&myMap);
array[1] = &myDictionary;
array[2] = dynamic_cast<Dictionary *> (&myMemory);
array[3] = &myContacts;

for (int i = 0; i < 4; i++)
{
    array[i] -> insert("String","String");
    array[i] -> del("String");
    array[i] -> isMapped("String");
    array[i] -> getVal("String");
    array[i] -> getKey("String");
    array[i] -> isVal("String");
    array[i] -> setLock();
    array[i] -> unsetLock();
```

100 %

Output

Show output from: Build

```
1>  TestProject1.cpp
1>c:\users\faith-anne\documents\visual studio 2010\projects\langs\langs\testproject1.cpp(33): warning C4244: 'argument' : conversion from 'time_t' to 'uns
1>c:\users\faith-anne\documents\visual studio 2010\projects\langs\langs\testproject1.cpp(80): error C2039: 'getKey' : is not a member of 'Dictionary'
1>          c:\users\faith-anne\documents\visual studio 2010\projects\langs\langs\dictionary.h(11) : see declaration of 'Dictionary'
1>c:\users\faith-anne\documents\visual studio 2010\projects\langs\langs\testproject1.cpp(81): error C2039: 'isVal' : is not a member of 'Dictionary'
1>          c:\users\faith-anne\documents\visual studio 2010\projects\langs\langs\dictionary.h(11) : see declaration of 'Dictionary'
1>c:\users\faith-anne\documents\visual studio 2010\projects\langs\langs\testproject1.cpp(82): error C2039: 'setLock' : is not a member of 'Dictionary'
1>          c:\users\faith-anne\documents\visual studio 2010\projects\langs\langs\dictionary.h(11) : see declaration of 'Dictionary'
1>c:\users\faith-anne\documents\visual studio 2010\projects\langs\langs\testproject1.cpp(83): error C2039: 'unsetLock' : is not a member of 'Dictionary'
1>          c:\users\faith-anne\documents\visual studio 2010\projects\langs\langs\dictionary.h(11) : see declaration of 'Dictionary'
========== Build: 0 succeeded, 1 failed, 0 up-to-date, 0 skipped ==========
```

Figure 17. Compile and runtime errors for a Dictionary array

Case 7: These are the compile errors that occur when trying to access each method from objects of all 4 types in a AssociativeMemory array.

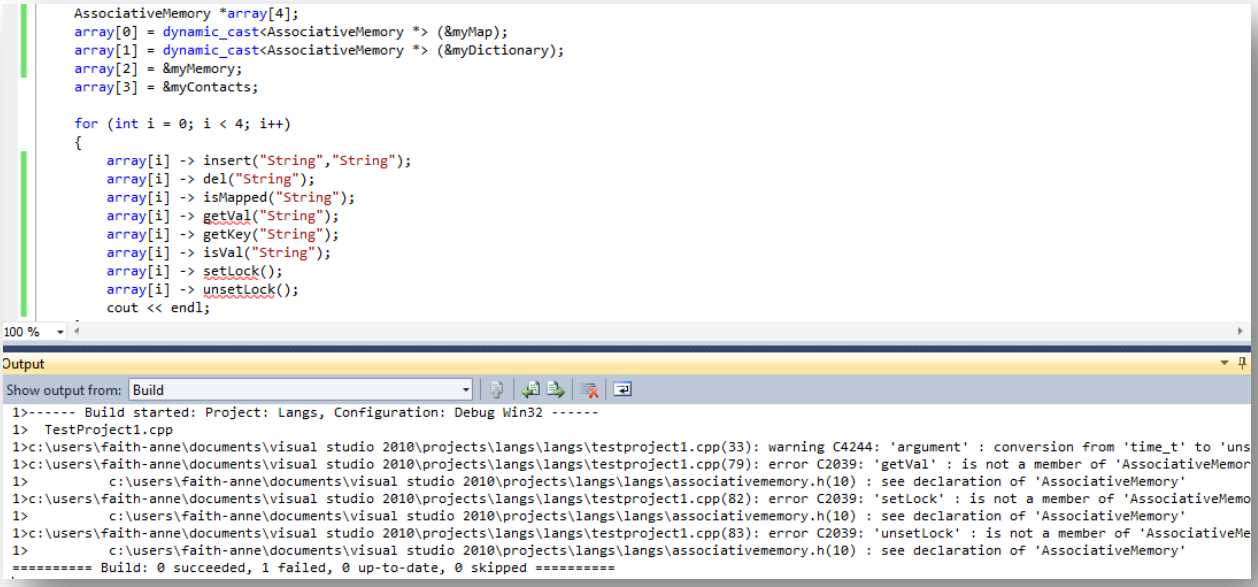


Figure 18. Compile and runtime errors for an AssociativeMemory array

Case 8: Creating an array of type ContactList crashed the IDE.

```
ContactList *array[4];
array[0] = dynamic_cast<ContactList *> (&myMap);
array[1] = dynamic_cast<ContactList *> (&myDictionary);
array[2] = dynamic_cast<ContactList *> (&myMemory);
array[3] = &myContacts;

for (int i = 0; i < 4; i++)
{
    array[i] -> insert("String","String");
    array[i] -> del("String");
    array[i] -> isMapped("String");
    array[i] -> getVal("String");
    array[i] -> getKey("String");
    array[i] -> isVal("String");
    array[i] -> setLock();
    array[i] -> unsetLock();
    cout << endl;
```

Figure 19. ContactList array causes runtime errors

### IVB. Java Screen Shots, Test Cases, and Errors

Case 1: The screen shot for the successfully executed Java program is below.

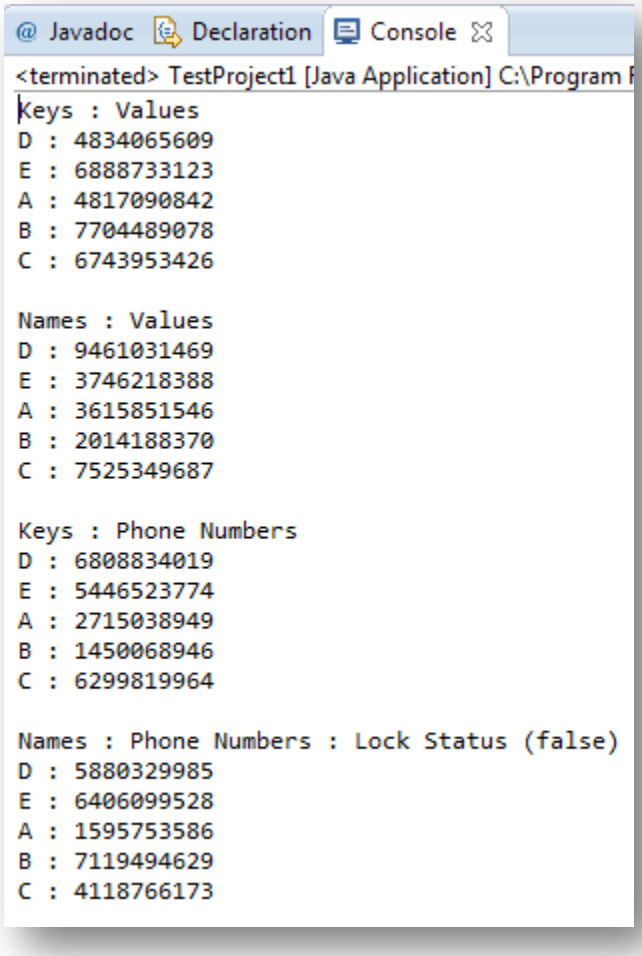


Figure 20. Error-free test program

Case 2: Creating a MapInterface for Map to implement, and instantiating the java.util.HashMap in that interface created a static object. This caused a logical error, but no compile or runtime errors. Running the test program, one sees that all of the entries in each class are identical.

The MapInterface was eventually abandoned, as it would have been only implemented by Map.

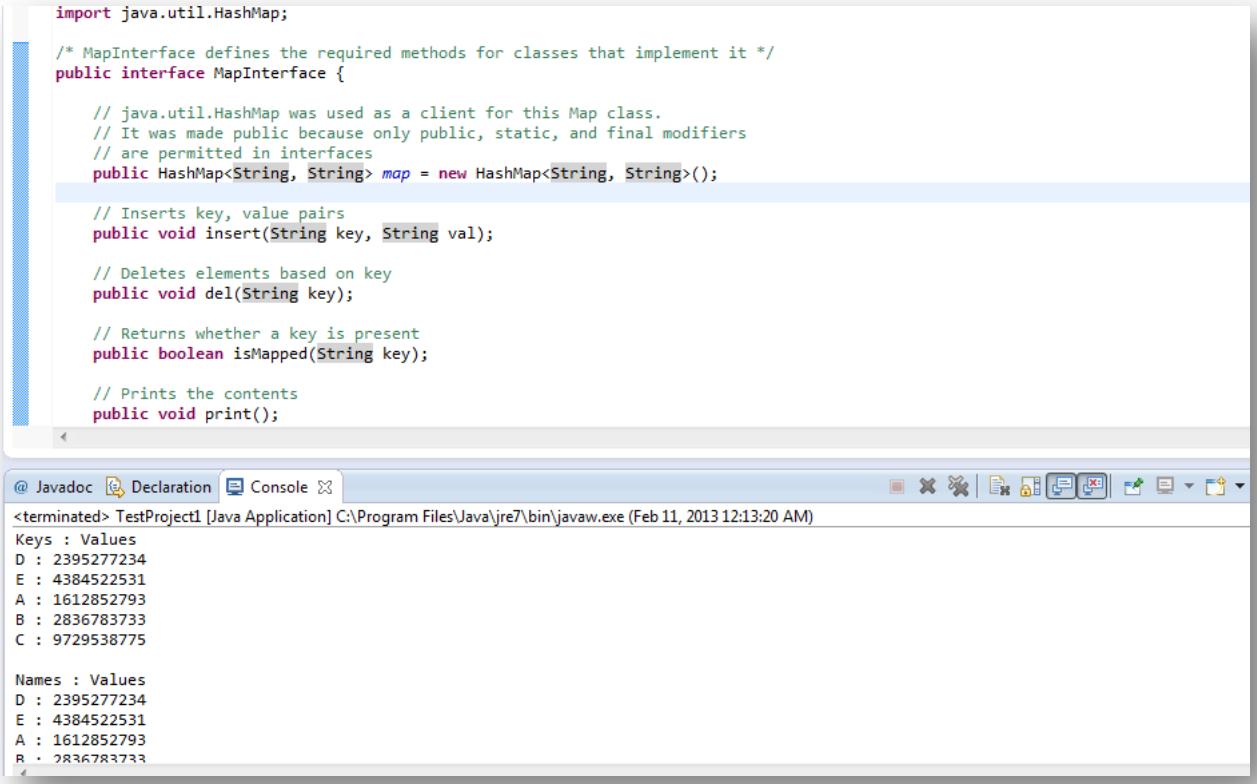


Figure 21. MapInterface test with HashMap declaration

Case 3: Making the AssociativeMemory class a subclass of AbstractDictionary had the unintended consequence of allowing it access to the getVal(String) method. The getVal(String) method was defined in AbstractDictionary because ContactList (a subclass of AssociativeMemory) required inherited access to that method. AssociativeMemory, however, does not have access to the nameCode variable that was defined in DictionaryInterface.

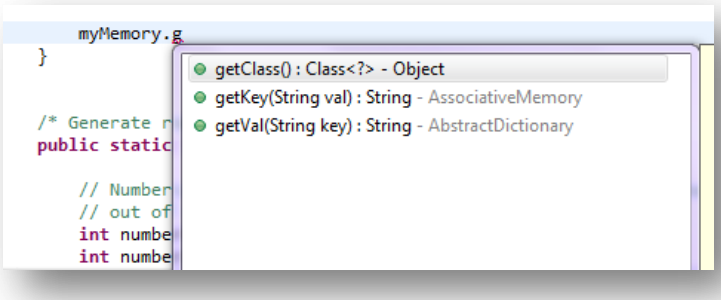


Figure 22: myMemory is given access to getVal(String)

Case 4: The screenshot below demonstrates which variables are available to which classes. myMap has access to none of the subclasses' variables. myDictionary has access to only nameCode. myMemory has access to only valCode, and myContacts has access to all three.

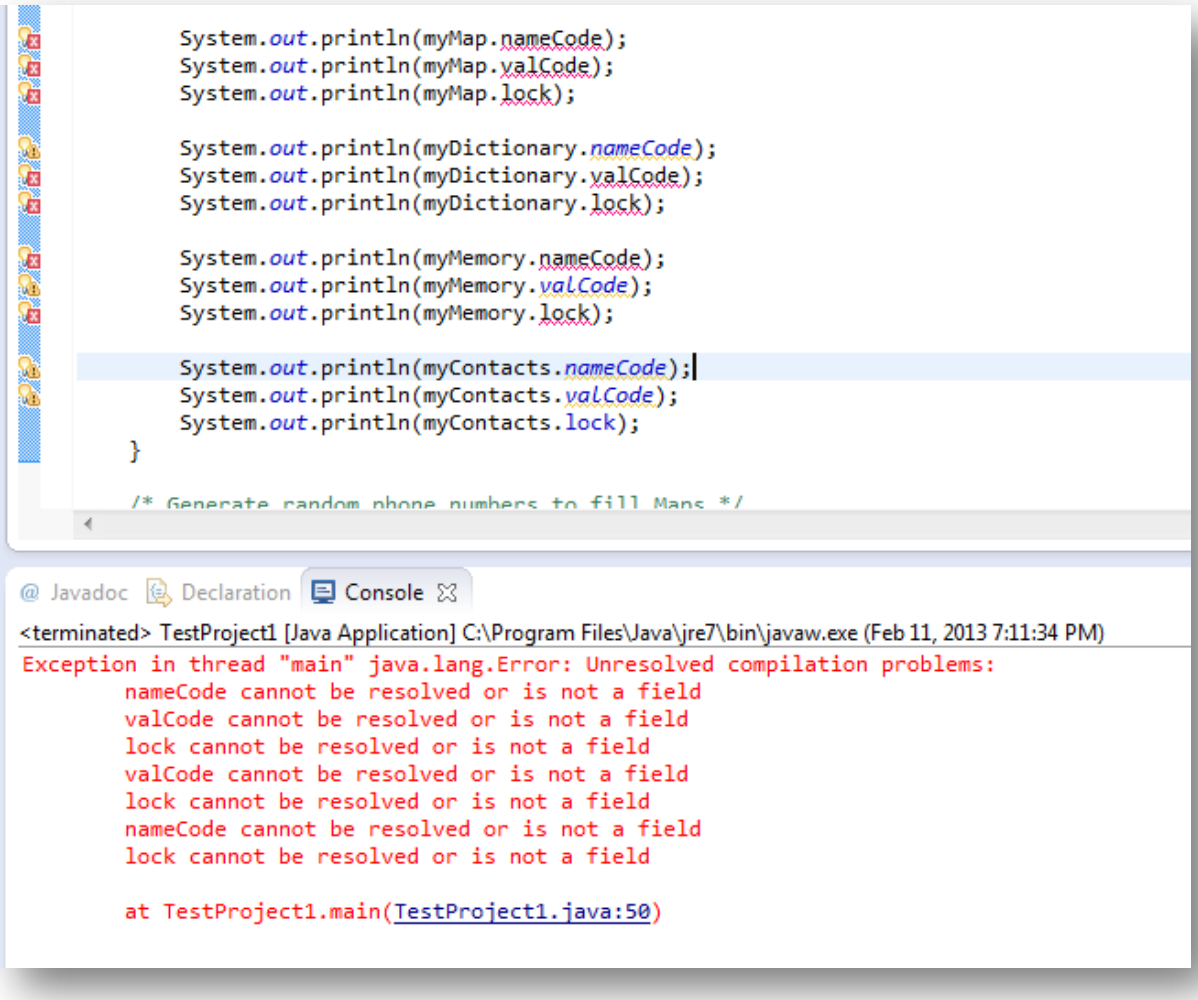


Figure 23. Class/variable availability

Case 5: The following screen shots show which methods are available to each of the classes.

```
myMap.insert("String", "String");
myMap.del("String");
myMap.isMapped("String");
myMap.print();
myMap.getVal("String");
myMap.getKey("String");
myMap.isVal("String");
myMap.setLock();
myMap.unsetLock();

myDictionary.insert("String", "String");
myDictionary.del("String");
myDictionary.isMapped("String");
myDictionary.print();
myDictionary.getVal("String");
myDictionary.getKey("String");
myDictionary.isVal("String");
myDictionary.setLock();
myDictionary.unsetLock();
```

```
myMemory.insert("String", "String");
myMemory.del("String");
myMemory.isMapped("String");
myMemory.print();
myMemory.getVal("String");
myMemory.getKey("String");
myMemory.isVal("String");
myMemory.setLock();
myMemory.unsetLock();

myContacts.insert("String", "String");
myContacts.del("String");
myContacts.isMapped("String");
myContacts.print();
myContacts.getVal("String");
myContacts.getKey("String");
myContacts.isVal("String");
myContacts.setLock();
myContacts.unsetLock();
```

Javadoc | Declaration | Console

```
<terminated> TestProject1 [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Feb 11, 2013 7:20:21 PM)
Exception in thread "main" java.lang.Error: Unresolved compilation problems:
        The method getVal(String) is undefined for the type Map
        The method getKey(String) is undefined for the type Map
        The method isVal(String) is undefined for the type Map
        The method setLock() is undefined for the type Map
        The method unsetLock() is undefined for the type Map
        The method getKey(String) is undefined for the type Dictionary
        The method isVal(String) is undefined for the type Dictionary
        The method setLock() is undefined for the type Dictionary
        The method unsetLock() is undefined for the type Dictionary
        The method setLock() is undefined for the type AssociativeMemory
        The method unsetLock() is undefined for the type AssociativeMemory
```

Figure 24. Class/method availability

Case 6: These are the compile errors that occur when trying to access each method and variable from objects of all 4 types in a Map array.

```
Map[] array = new Map[4];           // Create array of base type Map
array[0] = myMap;
array[1] = myDictionary;
array[2] = myMemory;
array[3] = myContacts;

for (int i = 0; i < 4; i++) {
    array[i].insert("String", "String");              // Polymorphic
    array[i].del("String");
    array[i].isMapped("String");
    System.out.println(array[i].nameCode);
    System.out.println(array[i].getVal("String"));
    System.out.println(array[i].valCode);
    System.out.println(array[i].getKey("String"));
    System.out.println(array[i].isVal("String"));
    array[i].setLock();
    array[i].unsetLock();
}
```

Figure 25. Compile errors for Map array

Case 7: These are the compile errors that occur when trying to access each method and variable from objects of all 4 types in a Dictionary array.

```
Dictionary[] array = new Dictionary[4];          // Create array
array[0] = (Dictionary) myMap;
array[1] = (Dictionary) myDictionary;
array[2] = (Dictionary) myMemory;
array[3] = (Dictionary) myContacts;

for (int i = 0; i < 4; i++) {
    array[i].insert("String", "String");                // Polymo
    array[i].del("String");
    array[i].isMapped("String");
    System.out.println(array[i].nameCode);
    System.out.println(array[i].getVal("String"));
    System.out.println(array[i].valCode);
    System.out.println(array[i].getKey("String"));
    System.out.println(array[i].isVal("String"));
    array[i].setLock();
    array[i].unsetLock();
}

}

/* Generate random phone numbers to fill Maps */
```

```
@ Javadoc  Declaration  Console
<terminated> TestProject1 [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Feb 12, 2013 4
Exception in thread "main" java.lang.Error: Unresolved compilation problems:
        Cannot cast from AssociativeMemory to Dictionary
        Cannot cast from ContactList to Dictionary
        valCode cannot be resolved or is not a field
        The method getKey(String) is undefined for the type Dictionary
        The method isVal(String) is undefined for the type Dictionary
        The method setLock() is undefined for the type Dictionary
        The method unsetLock() is undefined for the type Dictionary
```

Figure 26. Compile errors for a Dictionary array

Case 8: These are the compile and runtime errors that occur when trying to access each method and variable from objects of all 4 types in an AssociativeMemory array.

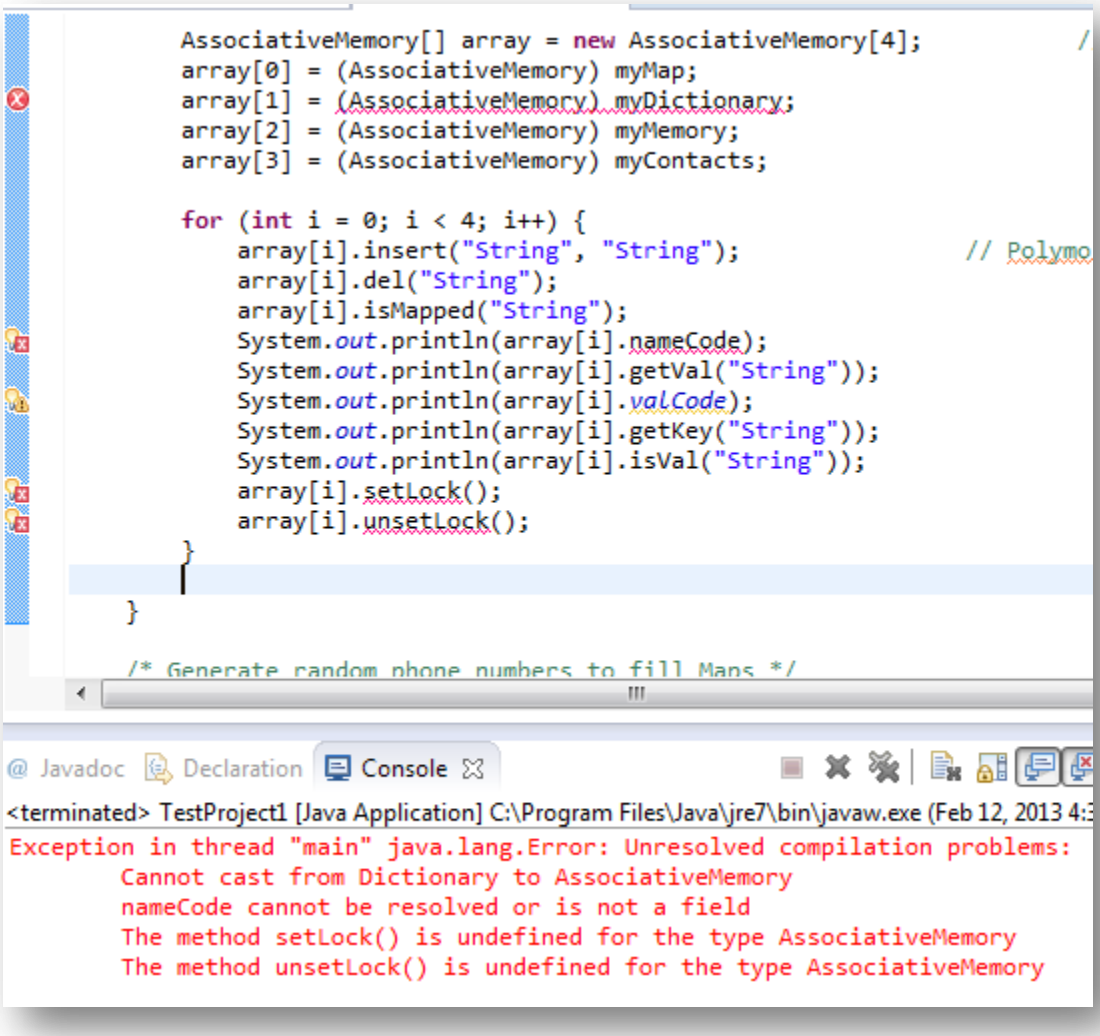


Figure 27. Compile and runtime errors for an AssociativeMemory array

Case 9: These are the compile and runtime errors that occur when trying to access each method and variable from objects of all 4 types in a ContactList array.

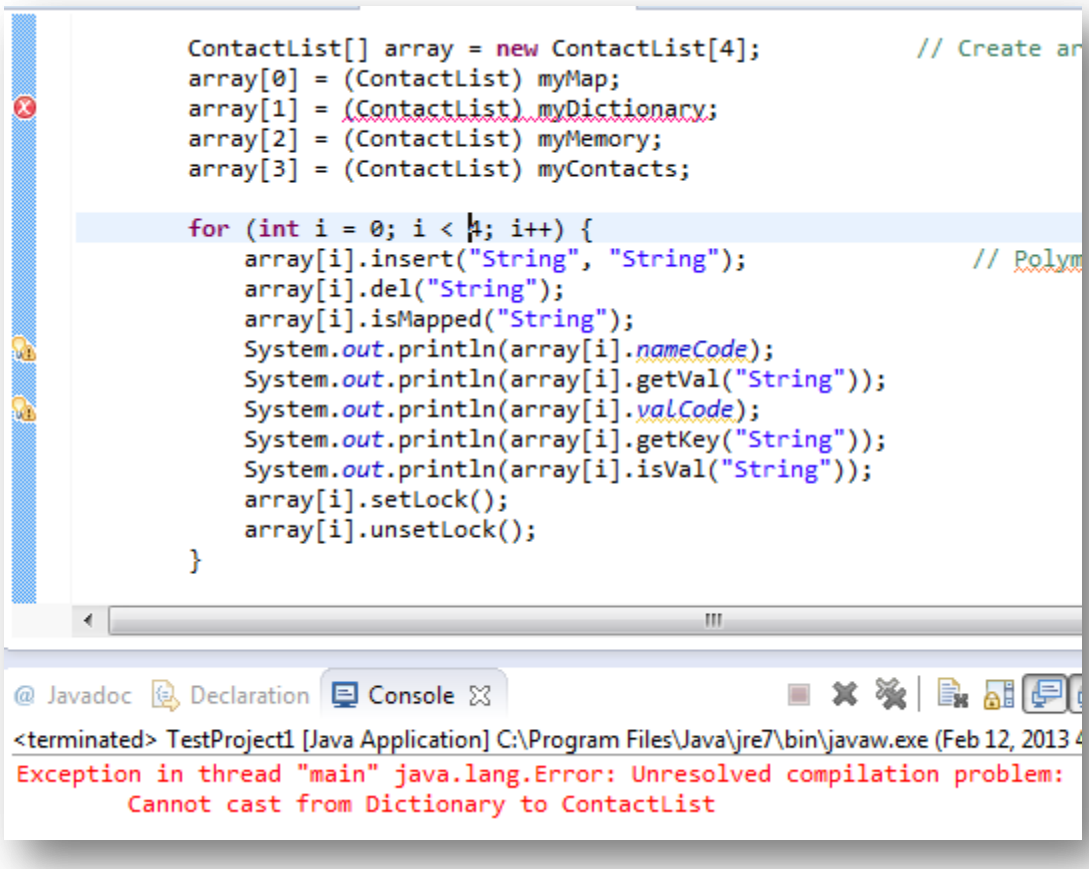


Figure 28. Compile and runtime errors for a ContactList array.

## V. Discussion Question

*What if getVal and getKey functions were named the same, say lookup, in each of Dictionary and AssociativeMemory. Also, assume that these two classes were library functions, and hence, you cannot alter their implementations. What issues are introduced for implementing ContactList using these library classes? Again, contrast C++ and Java.*

In the C++ class design, renaming both getVal in Dictionary and getKey in AssociativeMemory to lookup does cause confusion for the compiler, which provides error messages at those function calls. The way to combat this is to use a binary scope resolution operator to resolve this ambiguity. Such operators can be unwieldy and inelegant, but they are part of the clearly defined rules for allowing multiple inheritance in C++.

```
myContacts.Dictionary::lookup("String");
myContacts.AssociativeMemory::lookup("String");
```

Figure 29. Scope resolution in C++

With this particular Java class design, renaming both getVal in the AbstractDictionary class and getKey in the AssociativeMemory class to lookup does not cause compile or runtime errors. Since AssociativeMemory is a subclass of AbstractDictionary, the lookup method in AssociativeMemory overrides the lookup method in the AbstractDictionary class. Since ContactList directly extends AssociativeMemory, it utilizes the AssociativeMemory version of lookup.

As evidenced in the screenshot below, the lookup method returned a key (“D”) given a value (“9999999999”), as was expected from the AssociativeMemory implementation of lookup.

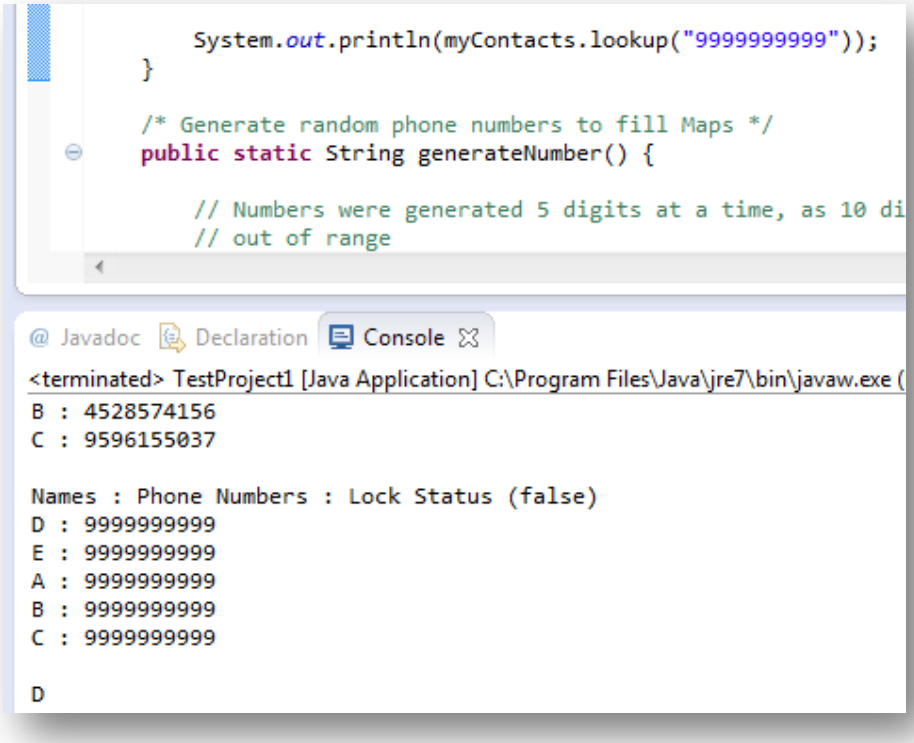


Figure 30. Lookup method output in Java

## V. Conclusions

Multiple inheritance in C++ is a powerful and useful feature. It promotes code re-use and is very analogous to how people categorize objects in the real world. So while Java lacks the elegance of a two parent structure, it also escapes the potential pitfalls. Multiple inheritance structures require strict attention to detail and to prevent compile errors brought on by ambiguity and overlap. That being said, there are a number of cases where the a diamond class structure succeeds, as in the case of iostream, where the class multiple inherits from istream and ostream (subclasses of ios in themselves).

So, while single-inheritance relationships may be simpler to implement in Java, simulating C++-like multi-inherited structures has proven an instructive challenge.

## VI. Compliance to Requirements

This project complies with all assignment requirements, including: C++ class design and implementation per assignment specifications, Java class design and adaptation, appropriate UML Diagrams, test cases and screen shots, program analysis, discussion, and conclusions.

## Appendix A: Source Code

```
/*--------------------------------------------------------------------------
* Author:       Faith-Anne Kocadag
* Written:      2/14/2013
*
* An example of multiple and repeated inheritance in C++ that avoids the
* "Deadly Diamond of Derivation" problem through the use of virtual base
* classes and virtual functions.
*-------------------------------------------------------------------------*/

#include <iostream>
#include <sstream>
#include<time.h>
#include <map>
#include "Map.h"
#include "Dictionary.h"
#include "AssociativeMemory.h"
#include "ContactList.h"
using namespace std;

/* Generate random phone numbers to fill the maps */
string generateNumber()
{
    stringstream ss2;
    int number1 = rand() % 89999 + 10000;   // Generates a random 10 digit
    int number2 = rand() % 89999 + 10000;   // number, 5 digits at a time
    ss2 << number1;
    ss2 << number2;
    return ss2.str();                       // Converts stringstream to string
}

int main()
{
    srand (time(0));                        // Seed random
    stringstream ss1;

    Map myMap;
    for (int i = 0; i < 5; i++)             // Fill myMap
    {
        ss1.str(std::string());             // Clear ss
        ss1 << (char)(i + 65);
        myMap.insert(ss1.str(), generateNumber());
    }

    Dictionary myDictionary;
    for (int i = 0; i < 5; i++)             // Fill myDictionary
    {
        ss1.str(std::string());             // Clear ss
        ss1 << (char)(i + 65);
        myDictionary.insert(ss1.str(), generateNumber());
    }

    AssociativeMemory myMemory;
    for (int i = 0; i < 5; i++)             // Fill myMemory
    {
        ss1.str(std::string());             // Clear ss
        ss1 << (char)(i + 65);
        myMemory.insert(ss1.str(), generateNumber());
    }

    ContactList myContacts;
    for (int i = 0; i < 5; i++)             // Fill myContacts
    {
        ss1.str(std::string());             // Clear ss
        ss1 << (char)(i + 65);
        myContacts.insert(ss1.str(), generateNumber());
```

```
    }

    Map *array[4];                          // Create array of Base type Map
    array[0] = &myMap;
    array[1] = &myDictionary;
    array[2] = &myMemory;
    array[3] = &myContacts;

    for (int i = 0; i < 4; i++)
    {
        array[i] -> print();                // Polymorphic print() calls
        cout << endl;
    }

    system("PAUSE");                        // Pauses the console window, so it
    return 0;                               // doesn't immediately close when
}                                           // run as an exe file
```

```
// Base class: Map header file
#ifndef MAP_H
#define MAP_H

#include <map>
#include <iostream>
using namespace std;

class Map
{
public:
    Map();
    void insert(string key, string val);
    void del(string key);
    bool isMapped(string key) const;
    virtual void print();   // Declared virtual in this base class to avoid
                            // Deadly Diamond. Also, not const because iterator
                            // p is not constant.
protected:                  // Protected variables are available to subclasses
    map<string, string> map1;
    map<string, string>::iterator p;
};

#endif
```

```
#include "Map.h"
#include <map>
#include <iostream>
#include <string>
using namespace std;

// STL map and iterator are used as clients for this Map class
// Map is not a derived class of STL map to explore the effect of multiple
// inheritance in a controlled way. STL map does not allow duplicate keys.
map<string, string> map1;
std::map<string, string>::iterator p;

// Default Constructor
Map::Map()
{
    // Removal of this empty constructor caused LINK errors when trying to
    // build the derived classes.
}

// Insert an element
void Map::insert(string key, string val)
{
    map1.insert(map<string, string>::value_type(key, val));
}

// Delete an element at a given key
void Map::del(string key)
{
    map1.erase(key);
}

// Returns true if the key occurs in the Map
bool Map::isMapped(string key) const
{
    // find(key) returns a pointer, key exists if pointer is not past the end
    return (map1.find(key) != map1.end());
}

// Prints all of the values in the Map
void Map::print()
{
    cout << "Keys : Values" << endl;
    for (p = map1.begin(); p != map1.end(); p++)
        cout << p->first << " : " << p->second << endl;
}
```

```
// Derived class: Dictionary header file
// Subclass of Map
#ifndef DICTIONARY_H
#define DICTIONARY_H

#include "Map.h"
#include <map>
#include <iostream>
using namespace std;

class Dictionary : virtual public Map   // Virtual base class avoids
{                                       // Deadly Diamond
public:
    Dictionary();
    string getVal(string key);
    void print();                       // Virtual keyword is not necessary
protected:                              // for overridden subclass functions
    string nameCode;
};

#endif
```

```
#include "Dictionary.h"
#include <map>
#include <iostream>
#include <string>
using namespace std;

string nameCode;

// Default Constructor
Dictionary::Dictionary() : Map()
{
    nameCode = "Names";
}

// Returns the value corresponding to a key
string Dictionary::getVal(string key)
{
    // find(key) returns the pointer for the element with that key
    p = map1.find(key);
    if (p == map1.end()) return "Name not found";
    else return p->second;
}

// Prints the contents of the Dictionary
void Dictionary::print()
{
    cout << nameCode << " : Values" << endl;
    for (p = map1.begin(); p != map1.end(); p++)
        cout << p->first << " : " << p->second << endl;
}
```

```
// Derived class: AssociativeMemory header file
// Subclass of Map
#ifndef ASSOCIATIVEMEMORY_H
#define ASSOCIATIVEMEMORY_H

#include "Map.h"
#include <iostream>
using namespace std;

class AssociativeMemory : virtual public Map    // Virtual base class avoids
{                                               // Deadly Diamond
public:
    AssociativeMemory();
    string getKey(string val);
    bool isVal(string val);
    void print();                               // Virtual keyword is not necessary
protected:                                      // for overridden subclass functions
    string valCode;
};

#endif
```

```
#include "AssociativeMemory.h"
#include <iostream>
#include <string>
#include <map>
using namespace std;

string valCode;

// Default Constructor
AssociativeMemory::AssociativeMemory() : Map()
{
    valCode = "Phone Numbers";
}

// Returns the key corresponding to a value
string AssociativeMemory::getKey(string val)        // No function exists in
{                                                   // STL Map that returns a
    for (p = map1.begin(); p != map1.end(); p++)    // key given a value, so
        if (p->second == val) return p->first;      // elements must be
    return "Value is not in memory.";               // iterated through.
}

// Returns true if the value does exist in the Associative Memory
bool AssociativeMemory::isVal(string val)
{
    map<string, string>::iterator p;                // No function exists in
    for (p = map1.begin(); p != map1.end(); p++)    // STL Map that checks for
        if (p->second == val) return 1;             // the existence of values,
    return 0;                                       // so elements must be
}                                                   // iterated through.

// Prints the contents of the Associative Memory
void AssociativeMemory::print()
{
    cout << "Keys : " << valCode << endl;
    map<string, string>::iterator p;
    for (p = map1.begin(); p != map1.end(); p++)
        cout << p->first << " : " << p->second << endl;
}
```

```
// Derived class: ContactList header file
// Subclass of both Dictionary and Associative Memory
#ifndef CONTACTLIST_H
#define CONTACTLIST_H

#include "AssociativeMemory.h"
#include "Dictionary.h"
#include <iostream>
using namespace std;

// Base classes do not need to be virtual since Dictionary and
// AssociativeMemory used a virtual Map base class.
class ContactList : public Dictionary, public AssociativeMemory
{
public:
    ContactList();
    void setLock();
    void unsetLock();
    void print();
protected:
    bool lock;
};

#endif
```

```
#include "ContactList.h"
#include <iostream>
#include <string>
using namespace std;

bool lock;

// Default Constructor calls on Dictionary and AssociativeMemory constructors
ContactList::ContactList() : Dictionary(), AssociativeMemory()
{
    lock = 0;   // default is unlocked
}

// Set lock
void ContactList::setLock()
{
    lock = 1;
}

// Unset lock
void ContactList::unsetLock()
{
    lock = 0;
}

// Prints contents of ContactList
void ContactList::print()
{
    cout << nameCode << " : " << valCode << " : Lock Status (" << lock << ")" << endl;
    map<string, string>::iterator p;
    for (p = map1.begin(); p != map1.end(); p++)
        cout << p->first << " : " << p->second << endl;
}
```

```java
/*-------------------------------------------------------------------------
 * Author:      Faith-Anne Kocadag
 * Written:     2/14/2013
 *
 * An example of multiple and repeated inheritance patterns from C++ adapted
 * for Java through the use of Interfaces and Abstract classes.
 -------------------------------------------------------------------------*/

public class TestProject1 {

    public static void main(String[] args) {

        String s;

        Map myMap = new Map();
        for (int i = 0; i < 5; i++) {       // Fill myMap
            s = Character.toString((char) (i + 65));
            myMap.insert(s, generateNumber());
        }

        Dictionary myDictionary = new Dictionary();
        for (int i = 0; i < 5; i++) {       // Fill myDictionary
            s = Character.toString((char) (i + 65));
            myDictionary.insert(s, generateNumber());
        }

        AssociativeMemory myMemory = new AssociativeMemory();
        for (int i = 0; i < 5; i++) {       // Fill myMemory
            s = Character.toString((char) (i + 65));
            myMemory.insert(s, generateNumber());
        }

        ContactList myContacts = new ContactList();
        for (int i = 0; i < 5; i++) {       // Fill myContacts
            s = Character.toString((char) (i + 65));
            myContacts.insert(s, generateNumber());
        }

        Map[] array = new Map[4];           // Create array of base type Map
        array[0] = myMap;
        array[1] = myDictionary;
        array[2] = myMemory;
        array[3] = myContacts;

        for (int i = 0; i < 4; i++) {
            array[i].print();               // Polymorphic print() calls
            System.out.println();
        }
    }

    /* Generate random phone numbers to fill Maps */
    public static String generateNumber() {
        long number1 = (long) (Math.random() * 9000000000L) + 1000000000;
        return Long.toString(number1);
    }
}
```

```java
import java.util.*;
import java.util.Map.Entry;

public class Map {

    // java.util.HashMap was used as a client for this Map class.
    // It is protected so that all subclasses can access it.
    protected HashMap<String, String> map;

    // Instantiate an empty Map
    public Map() {
        map = new HashMap<String, String>();
    }

    // Call on HashMap's put method to insert values
    public void insert(String key, String val) {
        map.put(key, val);
    }

    // Call on HashMap's remove method
    public void del(String key) {
        map.remove(key);
    }

    // Call on HashMap's containsKey method
    public boolean isMapped(String key) {
        return map.containsKey(key);
    }

    // Print the map
    public void print() {

        // Place map entries in a set. Sets are unordered.
        Set<Entry<String, String>> entries = map.entrySet();

        System.out.println("Keys : Values");
        // Use iterator to print the set
        for (Entry<String, String> entry : entries)
            System.out.println(entry.getKey() + " : " + entry.getValue());
    }
}
```

AbstractDictionary.java

```java
// AbstractDictionary extends map.
// It is a base class for Dictionary and AssociativeMemory
public class AbstractDictionary extends Map {

    // Returns a value given a key
    public String getVal(String key) {
        return super.map.get(key);
    }
}
```

```java
import java.util.Set;
import java.util.Map.Entry;

// Dictionary is derived from AbstractDictionary
public class Dictionary extends AbstractDictionary implements DictionaryInterface{

    // Create an empty dictionary
    public Dictionary() {

    }

    // Subclass print() overrides the superclass Map's print()
    @Override
    public void print() {

        System.out.println(nameCode + " : Values");

        // Place map entries in a set. Sets are unordered.
        Set<Entry<String, String>> entries = super.map.entrySet();

        // Use iterator to print the set
        for (Entry<String, String> entry : entries)
            System.out.println(entry.getKey() + " : " + entry.getValue());
    }
}
```

DictionaryInterface.java

```java
/* DictionaryInterface defines the required methods and variables for classes
 * that implement it */
public interface DictionaryInterface {

    // Variable is available to all classes that implement DictionaryInterface
    static String nameCode = "Names";

}
```

```java
import java.util.Set;
import java.util.Map.Entry;

public class AssociativeMemory extends AbstractDictionary {

    static String valCode = "Phone Numbers";

    // Create an empty AssociativeMemory
    public AssociativeMemory() {

    }

    // Returns the key, given a value
    public String getKey(String val) {

        // Iterate through entries
        for (Entry<String, String> entry : super.map.entrySet()) {

            // If value is found, return key
            if (val.equals(entry.getValue()))
                return entry.getKey();
        }

        // If for loop ends without finding value, return error message
        return "Value not found";
    }

    public boolean isVal(String val) {
        // Iterate through entries
        for (Entry<String, String> entry : super.map.entrySet()) {

            // If value is found, return true
            if (val.equals(entry.getValue()))
                return true;
        }

        // If for loop ends without finding value, return false
        return false;
    }

    // Subclass print() overrides the superclass Map's print()
    @Override
    public void print() {

        System.out.println("Keys : " + valCode);

        // Place map entries in a set. Sets are unordered.
        Set<Entry<String, String>> entries = super.map.entrySet();

        // Use iterator to print the set
        for (Entry<String, String> entry : entries)
            System.out.println(entry.getKey() + " : " + entry.getValue());
    }

}
```

```java
import java.util.Map.Entry;
import java.util.Set;

// ContactList extends AssociativeMemory
public class ContactList extends AssociativeMemory implements DictionaryInterface {

    protected boolean lock;

    // Default Constructor
    public ContactList() {
        lock = false;
    }

    // Sets lock to true
    public void setLock() {
        lock = true;
    }

    // Sets lock to false
    public void unsetLock() {
        lock = false;
    }

    // Prints the contents of the ContactList
    @Override
    public void print() {
        System.out.println(nameCode + " : " + valCode + " : Lock Status ("
                + lock + ")");

        // Place map entries in a set. Sets are unordered.
        Set<Entry<String, String>> entries = super.map.entrySet();

        // Use iterator to print the set
        for (Entry<String, String> entry : entries)
            System.out.println(entry.getKey() + " : " + entry.getValue());
    }
}
```

## VII. References

[1] Liang, D. 2011. Introduction to Java Programming: Comprehensive Version, Pearson, Upper Saddle River, NJ, 374.

[2] Kalev, D. "The Virtues of Multiple Inheritance." C++ Reference Guide. 1 Jan 2004. Retrieved from: http://www.informit.com/guides/content.aspx?g=cplusplus&seqNum=167.

[3] Tucker, A. and Noonan, R. 2007. Programming Languages: Principles and Paradigms, McGraw-Hill Higher Education, New York, NY, 328.

[4] Tucker, A. and Noonan, R. 2007. Programming Languages: Principles and Paradigms, McGraw-Hill Higher Education, New York, NY, 326.