Hierarchical Map

Hideyuki Sekiguchi

hideyuki_sekiguchi@hotmail.com

***Abstract**. Traditional maps and lists support only one-dimensional information and lack support for the representation of hierarchically organized data. On the other hand, there are many cases where hierarchically organized data makes the information clearer and easier to handle. This pattern aims to define a data structure interface that can easily handle hierarchically organized information. Thus, nodes containing information are accessible by a simple path-like composite key.*

1. Context

There are many situations, during software development, where it is necessary to deal with a set of information. This set of information can be divided into subsets with hierarchical relationships among them. Each subset has a label to facilitate identification.

This pattern is for software developers who need to deal with a set of hierarchically organized information.

2. Problem

How should the information be organized to represent which pieces are subsets and which are supersets? How can a particular subset be manipulated without traversing through all the subsets composing the information.

3. Forces

- *Hierarchically organized information is easy for humans to understand. Deeply nested hierarchies are hard to manipulate.*

  It is difficult for a human to process more than six pieces of information. Organizing data in small pieces makes them friendlier for users and simplifies their understanding. For the same reason, it is difficult for a programmer to deal with more than a couple of nested iterators.

- *Information accessed through a label is user friendly. Information accessed through an index is more efficient for a machine.*

  It is easier for a human being to refer to information using a label because we can choose a more meaningful name. On the other hand, machines spend less effort retrieving information using a positional index.

- *The same information should be able to be reorganized in a different hierarchy. The same information should have a single label as its identity.*

Sometimes, it is necessary to reorganize information in a different hierarchy depending on the user. Although, understanding of the information becomes easier when the same information has the same label.

4. Solution

Provide a simple data structure in which you can query the objects as in a hierarchical file system. Define a single interface, called a HierarchicalMap for this data structure. This data structure should map the object to a key and organize these keys hierarchically as in the directory structure of a file system. Each directory, or node, corresponds to an instance of a HierarchicalMap and the file, or leaf, represents the other objects. Here the key, which corresponds to the name of directory or file, can be repeated, i.e. the key would not be unique.

The interface should have the following methods:

- Querying operations: get(key) and getAll(key)
- Structure handling operations: put(key, value), add(key, value), addAll(key, collection), remove(key) and removeAll(key)
- Collection operations: values(), keySet() and entrySet()

This data structure can be implemented using the LinkedHashMultiMap, which is based on HashMap (Figure 1).

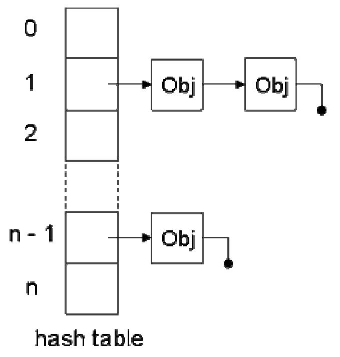

Figure 1. Data Structure for implementation of HashMap

LinkedHashMultiMap has extra pointers (Figure 2) to permit insertion-order iteration through "before" and "after" pointers. Other pointers, "nextSibling" and "previousSibling," are used to keep more than one object with same key.

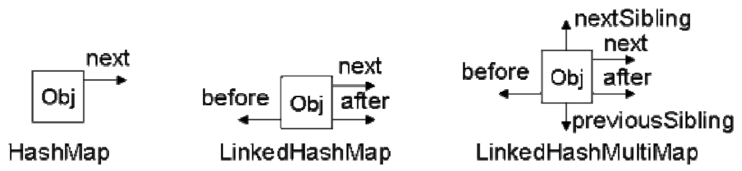

Figure 2. Several pointers are used to implement different types of HashMap

The sequence diagram in Figure 3 illustrates the interaction between HierarchicalMap and LinkedHashMultiMap to implement get(key) operation. Note that there is a Tokenizer class to parse the key parameter into tokens.

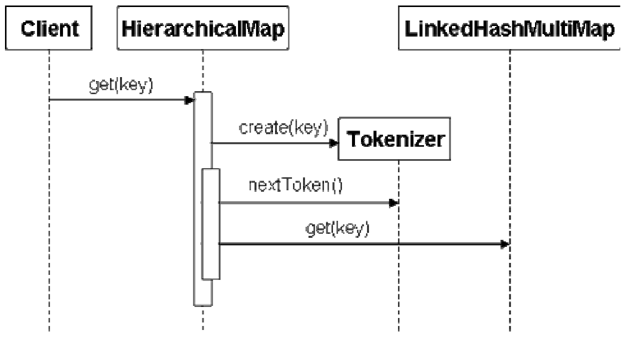

Figure 3. Sequence Diagram for get(key) operation

The nextToken() and get(key) operation over LinkedHashMultiMap should be repeated while there is a valid token returned by Tokenizer class.

5. Resulting Context

Since each node of the HierarchicalMap may contain other HierarchicalMaps, a lot of data can be manipulated in a single operation. It is simpler to transfer a lot of data instead of creating a loop and then iterating over each data item, however, it also increases the chance of accidentally moving data that is not necessary.

HierarchicalMap has the same problem of any map in which there is no indication of allowed keys or required keys. A helper class should be used to avoid this situation. This class could check the keys and associated data types against XSD (XML Schema Definition) for instance, and also the IDE could check it during coding, and the compiler could insert checking code at runtime.

```
//@schema customer customer.xsd
HierarchicalMap customer = new BasicHierarchicalMap();
```

The presence of recursive reference must be checked to avoid infinite-loop during serialization of HierarchicalMap.

Table 1 summarizes the positive and negative consequences:

Positive	**Negative**
Huge amounts of data can be moved in a single operation	A lot of data can accidentally be moved in a single operation
The information can be split to form a smaller set	It is harder to figure out the big picture of the entire information structure
The same node can be shared among several HierarchicalMaps	The same HierarchicalMap can be recursively referred to causing infinite loop during serialization

Table 1. Positive and negative consequences of HierarchicalMap solution

6. Rationale

The HierarchicalMap could be implemented using the Composite pattern (Figure 4) to hold the hierarchy of information and the Visitor pattern for iterating over it. This solution is best for GUI applications, for instance, where graphic components are grouped using the Composite pattern and the Visitor pattern can be used to change the back ground color.

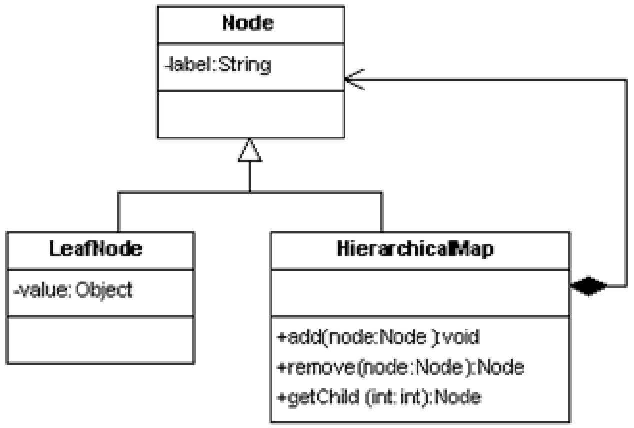

Figure 4. Class diagram for the data structure applying the Composite pattern

Sometimes the relationships among data composing the information can be very weak or even unrelated. In other circumstances, the data is temporary without any common operations. This is the case of configuration information used to setup applications behavior like “TCP/IP socket port number” or “maximum log file size. For this type of information, a simple data structure like HashMap or LinkedList is desirable.

The class in Figure 4 can hold all the desired data, but it is difficult to present the notion of hierarchy. Indeed, there is no way to obtain the node that is not the direct child of the current node. To reach a node that is several nodes deeper than the current node, we have to navigate through the child lists.

A similar difficulty occurs when we have to add a node lower in the hierarchy (Figure 5). We have to descend through the hierarchy while the intermediate nodes exist and append newly created nodes to reflect the desired hierarchy. After that we can finally add the target node.

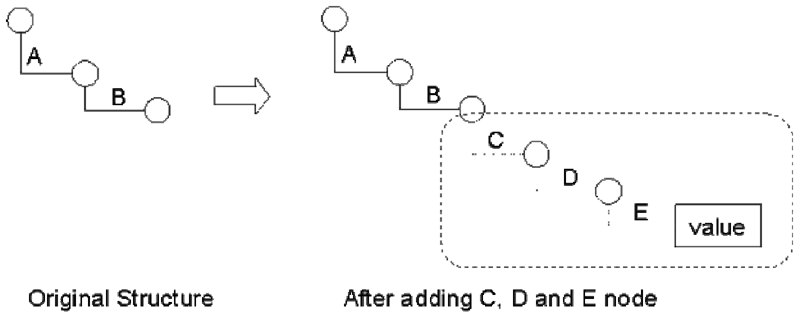

Figure 5. Appending a node lower in the hierarchy

7. Example

Many applications rely on configuration information stored apart from the application, which can be altered to produce the desired behavior without modifying the application itself. The increasing complexity of this configuration information, associated with the variety of alternatives to store it, results in considerable effort to provide a highly customizable application.

This configuration information can be stored in several formats such as the Windows Registry, Environment Variables, Configuration File (e.g. .ini or xml files), Relational Data Base, or even in a Directory Service.

Suppose an application provides a service with a network interface. The application needs to be installed on many different sites. All sites are expected to run Win32 servers. Each site administrator can choose the port number in which the application will run and the available protocols (TCP/UDP, HTTP/SOAP). There are other fine-tuning settings such as: maximum buffer size, maximum number of threads, and thread scheduling policy.

One way to deliver this application is to compile a version for each site needed to run the service. Another way is to configure the application at runtime (when the system starts). Applications under Win32 are usually configured by accessing the Registry, Environment Variables or Config files (.ini or .xml). In many cases the information is better represented in a hierarchy, where the ancestors of a node provide context to that node (Figure 6).

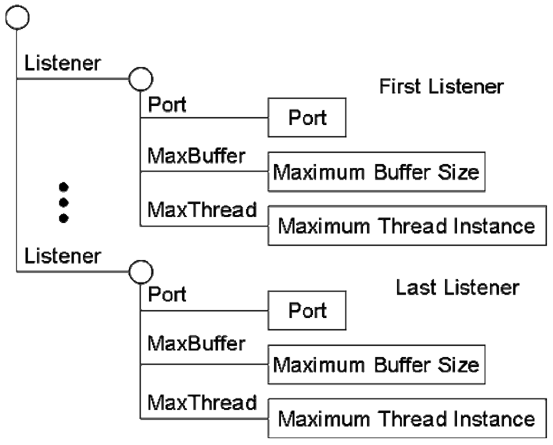

Figure 6. information required by this sample application

Something like the following code can be written to retrieve the information from the Unix like Environment Variable:

```
void getConfig(char **szPortNumber, char **szMaxBuff, char **szMaxThread)
{
	*szPortNumber = getenv("PORT");
	*szMaxBuff    = getenv("MAXBUFF");
	*szMaxThread  = getenv("MAXTHREAD");
}
```

For Win32 Registry (in C), it would be:

```
void getConfig(char *szPortNumber, char *szMaxBuff, char *szMaxThread)
{
	HKEY hKey;
	DWORD dwSize;
	RegOpenKeyEx(HKEY_LOCAL_MACHINE, "SOFTWARE\\ACME\\MY_APP",
	0, KEY_QUERY_VALUE, &hKey);
	RegQueryValueEx(hKey, "PORT", NULL, NULL,
	     (unsigned char*)szPortNumber, &dwSize);
	RegQueryValueEx(hKey, "MAXBUFF", NULL, NULL,
	     (unsigned char*) szMaxBuff, &dwSize);
	RegQueryValueEx(hKey, "MAXTHREAD", NULL, NULL,
	     (unsigned char*) szMaxThread, &dwSize);
}
```

And for Directory Server supporting LDAP (in Java):

```
public void getConfig(DirContext ctx, String port, String maxBuff, String
maxThread) {
	try {
	Attributes attrs = ctx.getAttributes("cn=MY_APP, ou=ACME");
	port = attrs.get("Port").get().toString();
	maxBuff = attrs.get("MaxBuffer").get().toString();
	maxThread = attrs.get("MaxThread").get().toString();
	} catch (NamingException e) {
	System.err.println(e);
	}
}
```

Note that, although the information needed is exactly the same (Port Number, Maximum Buffer Size and Maximum Concurrent Threads), the way to retrieve this information can vary substantially.

Suppose that this configuration will be kept in a Directory Service to allow centralized management of applications. But start storing the data in an XML file while the “Corporate LDAP Server Project” has not yet finished inside the organization.

This means that writing an application bound to the particular configuration repository is not a good idea. The application should not have to be concerned with the particular retrieval method of the repository. Instead, this information should be converted into a repository-neutral data structure so that the rest of the application only has to deal with this single data structure. The HierarchicalMap could be used for this purpose.

8. Sample Code

The following data (Figure 7) will be used during this sample:

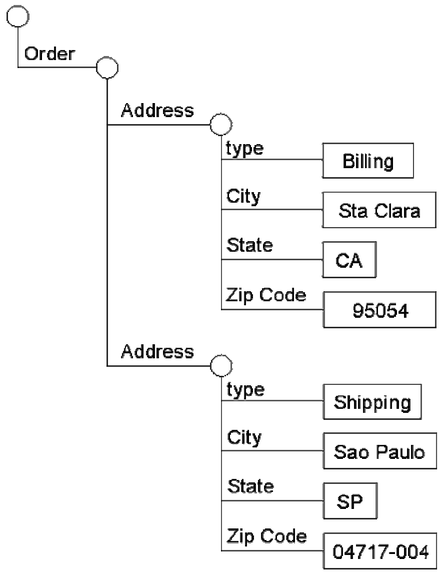

Figure 7. Information structure used during this example

Each circle or node represents an instance of HierarchicalMap and the rectangle or leaf represents any other object. A node can map subsequent nodes (this is the case of "Order" and "Address"), or can map leaves, e.g. *java.lang.String*, (this is the case of “type”, "City", "State" and "Zip Code").

8.1 Creating the structure

The following code will create a data structure representing the data shown above:

```
//create a HierachicalMap containing first address
HierarchicalMap address1 = new BasicHierarchicalMap();
address1.put("type", "Billing");
address1.put("City", "Sta Clara");
address1.put("State", "CA");
address1.put("ZIP Code", "95054");

//add the address to newly created hmap
HierarchicalMap hmap = new BasicHierarchicalMap();
hmap.add("Order/Address", address1);
```

Continue to next page …

```
//create another HierachicalMap containing second address
HierarchicalMap address2 = new BasicHierarchicalMap();
address2.put("type", "Shipping");
address2.put("City", "São Paulo");
address2.put("State", "SP");
address2.put("ZIP Code", "04717-004");

//add the second address to hmap
hmap.add("Order/Address", address2);
```

The method "add" could also have another signature in which it receives only the "key" and return an empty map appended under that "key". Then, the second address can be added using the following code:

```
//create another HierachicalMap containing second address
HierarchicalMap address2 = hmap.add("Order/Address");
address2.put("type", "Shipping");
address2.put("City", "São Paulo");
address2.put("State", "SP");
address2.put("ZIP Code", "04717-004");
```

8.2 Recovering the data

Once the structure is filled, you can query the data inside:

```
System.out.println(hmap.get("Order/Address/City"));
System.out.println(address2.get("State"));
```

The lines above will produce the following output:

```
Sta Clara
SP
```

Note that the get operation will recover a single data item, which we stipulated to be the first data inserted. Even though there are two *cities* stored in the structure, the first one (*Sta Clara*) was retrieved in the example above. On the other hand, the return for the *state* was *SP* because we retrieved the first state from the *address2* node. If you need to obtain entire set of objects with same key you should use getAll instead:

```
//example using getAll wich returns a Collection of objects matching
//the key
java.util.Collection val = hmap.getAll("Order/Address/City");

//get the iterator to printout all the objects returned
java.util.Iterator itr = val.iterator();
while(itr.hasNext()) {
      System.out.println(itr.next());
}
```

The code above will produce the following output:

```
Sta Clara
São Paulo
```

8.3 Interacting with Collections

Iterating with the structure will give the most interesting result:

```
//first, get all the addresses
java.util.Collection addresses = hmap.getAll("Order/Address");

//for each address in addresses
java.util.Iterator adritr = addresses.iterator();
while(adritr.hasNext()) {
       System.out.println("Address");
       //get the first address wich is a HierarchicalMap
       HierarchicalMap address = (HierarchicalMap)adritr.next();

       //get the entrySet to iterate with its children, i.e. City, State
       //and ZIP code
       java.util.Set set = address.entrySet();
       java.util.Iterator itr = set.iterator();
       while(itr.hasNext()) {
       //printout the pair key value for each entry
       java.util.Map.Entry entry =
       (java.util.Map.Entry)itr.next();
       System.out.println(entry.getKey() + " : " +
       entry.getValue());
       }
       System.out.println();
}
```

This will produce the following output:

```
Address
type : Billing
City : Sta Clara
State : CA
ZIP code : 95054

Address
type : Shipping
City : São Paulo
State : SP
ZIP code : 04717-004
```

8.4 Restructuring the Map

Although the structure in Figure 7 conforms to XSD recommendations, it is not possible to take advantage of HierarchicalMap. Information organized in this way does not allow direct access to inner data (e.g. to get shipping State). This direct access can be achieved with a little restructuring of the HierarchicalMap. We can implement th following method for example:

```
public static void denormalize(HierarchicalMap map,
                    String field, String keyField) {
    //first, remove all fields and substitute with empty map
    Collection col = map.removeAll(field);
    HierarchicalMap newMap = map.add(field);

    //iterate through removed fields
    Iterator itr = col.iterator();
    while(itr.hasNext()) {
        //for each object removed check it if it is a node
        //(i.e. instance of HierarchicalMap)
        Object o = itr.next();
        if(o instanceof HierarchicalMap) {
            //if so, try to get the value of key field in which
            //the node, will be added
            Object key = ((HierarchicalMap)o).get(keyField);
            if(key != null) {
                newMap.add(key.toString(), o);
            }
        }
    }
}
```

With the above method, we can transform the original map to the following structure (Figure 8):

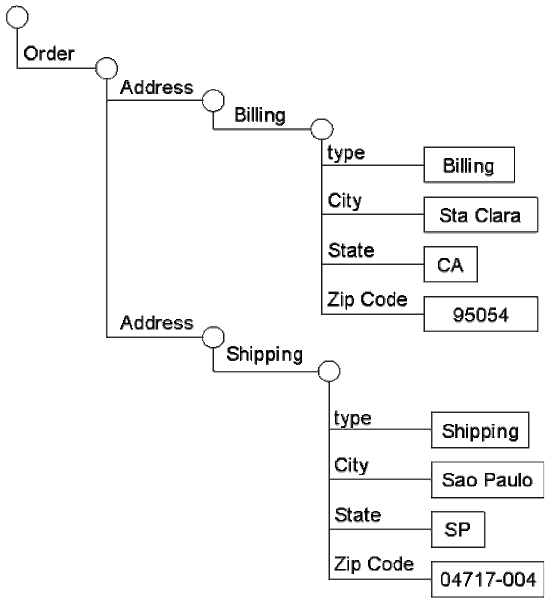

Figure 8. Reorganized information structure to take advantage of HierarchicalMap

Now, we can query the map as follows:

```
//call denormalize method defined earlier to reorganize the structure
//from Figure 7 to Figure 8
denormalize(map,"Order/Address","type");

//now fetch the desiered information directly
System.out.print("Shipping State: ");
System.out.println(map.get("Order/Address/Shipping/State"));
System.out.print("Billing State: ");
System.out.println(map.get("Order/Address/Billing/State"));
```

The result will be:

```
Shipping State: SP
Billing State: CA
```

Now, we can write the following method to restructure the map back to the original format:

```
public static void normalize(HierarchicalMap map,
                    String field, String keyField) {
    //first, remove all fields
    Collection col = map.removeAll(field);

    Iterator itr = col.iterator();
    while(itr.hasNext()) {
        //for each object removed check it if it is a node
        //(i.e. instance of HierarchicalMap)
        Object o = itr.next();
        if(o instanceof HierarchicalMap) {
            //Then, add each sub-node under field, but setting
            //a new field containing the key in which the sub-node
            //was referred
            Iterator itr2 = ((HierarchicalMap)o).entrySet().iterator();
            while(itr2.hasNext()) {
                Entry e = (Entry)itr2.next();
                if(e.getValue() instanceof HierarchicalMap) {
                    HierarchicalMap newMap =
       (HierarchicalMap)e.getValue();
                    //note that it is using put instead of add
                    //to ensure uniqueness of the key field
                    newMap.put(keyField, e.getKey());
                    map.add(field, newMap);
                }
            }
        }
    }
}
```

9. References

Tarr, Bob (Spring 2005) “CMSC 446: Introduction To Design Patterns”, http://www.research.umbc.edu/~tarr/dp/spr05/cs446.html, Computer Science and Electrical Engineering Department, University of Maryland Baltimore County.

Gamma, E., Helm, R., Johnson,R.,Vlissides, J. (1995) "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley.

10.Acknowledgement

Thanks to Douglas Atique (douglas.atique@gmail.com), the author of the first implementation of HierarchicalMap, this was the inspiration for this pattern. Also thanks to Marcelo Gulfier (mgulfier@hotmail.com) and Paulo Silveira (silveira@ime.usp.br) who contributed during the discussion regarding further enhancements.

Special thanks to Federico Balaguer (balaguer@uiuc.edu), my shepherd during SugarLoafPlop 2005, for all the encouragement and guidance he had given me to write this pattern.

And great thanks for Linda Rising (linda@lindarising.org / www.lindarising.org) who gave me the kind “gift” during the SugarLoafPlop 2005. She offered me to edit my paper!

An Open Source implementation of HierarchicalMap can be found at:

http://sourceforge.net/projects/hierarchicalmap