Efficient Reusable Collections

Davud MOHAMMADPUR\(^{(a)}\), Member and Ali MAHJUR\(^{(b)}\), Nonmember

SUMMARY  Efficiency and flexibility of collections have a significant impact on the overall performance of applications. The current approaches to implement collections have two main drawbacks: (i) they limit the efficiency of collections and (ii) they have not adequate support for collection composition. So, when the efficiency and flexibility of collections is important, the programmer needs to implement them himself, which leads to the loss of reusability. This article presents neoCollection, a novel approach to encapsulate collections. neoCollection has several distinguishing features: (i) it can be applied on data elements efficiently and flexibly (ii) composition of collections can be made efficiently and flexibly, a feature that does not exist in the current approaches. In order to demonstrate its effectiveness, neoCollection is implemented as an extension to Java and C++.

key words: programming language, high-level abstraction, collections efficiency, collections composition, collections reusability

1. Introduction

Programs use a variety of primitive collections or compositions of them. They have a significant effect on the overall performance of applications\(^{(1)}\), \(^{(2)}\). For example, Hadoop, a distributed processing framework for large data sets, uses many Java collections such as List, LinkedList, HashMap, HashSet, TreeSet and LinkedHashSet.

The success of collections depends on their efficiency and flexibility. Without an efficient approach to implement and apply collections on data elements, the performance of programs is reduced. In addition, programmers often need to customize collections flexibly and apply multiple collections simultaneously on a given data set.

As current programming languages do not meet these needs\(^{(3)}\), \(^{(4)}\) programmers have to implement collections from scratch to achieve the desirable efficiency and flexibility (losing reusability).

In current programming languages, collections are typically implemented by referencing to data elements. Mandating programmers to store data references in collections needs that multiple memory blocks are allocated per data element: one to store the data element and another one to store the collection’s node. There were some drawbacks to this traditional method\(^{(5)}\), \(^{(6)}\):

- Decreasing performance: The memory allocation routine should be called several times to allocate the space of a data element and it’s node. Such extra calls reduce the performance of collections.
- Decreasing flexibility: As there is no path from a data to the corresponding node, the collection should be traversed to find the node\(^{(7)}\), \(^{(8)}\). It yields two important shortcomings: (i) inability to implement operations efficiently (ii) inability to compose collections.

To clarify the above issues, consider a doubly linked list. Clearly, removing a data element from it needs O(1)\(^{(9)}\). However examining the implementation of LinkedList in Java as a doubly linked list, reveals that it has just a method to remove an element from the list. As shown in Fig. 1, it traverses the list to find and remove the element from the list. So, it has high time complexity (O(n))\(^{(1)}\).

Now consider the composition of a tree and a list in the following example. The time complexity of add in the Java LinkedList is O(1). Also, removing a data element by its key needs O(n). While is expected that by applying of LinkedList along with a TreeMap removing by key becomes O(log n), which seems a considerable improvement, however, this scenario is not feasible in Java. Since the collections operate independently, after finding the data element in TreeMap, removing it from LinkedList still has the time complexity of O(n). Therefore, applying of LinkedList along with TreeMap only worsens the situation. It increases the insertion time to O(log n), and duplicates data (Storing a data in two collections) without any improvement in other operations.

The solution is to concatenate the data element and the node. We call it concatenating approach\(^{(10)}\). In this ap-
proach, one memory block stores both the data element and the node of the collection. So, each data element of the collection is a node of it and vice versa. This approach has three significant advantages [10]:

1. One allocation: Concatenating approach stores the data elements in the nodes of the collection. Therefore, one memory block is allocated per data element.

2. Data pointer is the node pointer: As the data element is concatenated to the node of the collection, every pointer to the data element is a pointer to its node and vice versa. So, there is no need to traverse to find the node of the data element.

3. Composition: A data element can be concatenated with nodes of multiple collections, so, it provides new facility to apply multiple collections on a set of data elements.

This paper introduces a novel abstraction and its implementation called neoCollection. neoCollection is based on the concatenating approach and improves efficiency and flexibility of collections. neoCollection abstraction encapsulates the collection concern, and provides composition operators to support flexible and efficient composition of collections.

The neoCollection approach does not make any decision on behalf of programmers and does not oblige the programmer to store references or data in the collection. The programmer will have the option of storing references or data itself in the collection.

The neoCollection approach outlines a general, language-independent model. Reusability together with efficiency are advantages of the neoCollection approach.

This paper is organized as follows: Sect. 2 presents the related works. Section 3 discusses the neoCollection abstraction. Section 4 discusses composition of neoCollection. Section 5 presents formalization of neoCollection. Section 6 discusses neoCollection from the performance point of view. Section 7 presents a case study. Section 8 provides an overview of the implementation. Finally, Sect. 9 concludes the paper.

2. Related Works

The current approaches such as generic programming store references to the data elements in the nodes of the collection [11], [12]. This causes the mentioned issues in Sect. 1. The appropriate approach should not limit the programmer, and the programmer will have the option of storing references or data itself in the collection. An appropriate alternative to the current approaches is the inheritance. But due to the limitations of multiple inheritance, it isn’t used for this propose.

In the literature, there are some proposals suggesting special handling of collections at the language level. Smaragdakis et al. [1] present DiSTIL, a software generator for the domain of container data structures. The language of DiSTIL extends the C programming language with domain-specific constructs for specifying complex data structures declaratively. Bergel et al. [13] describe some improvements of the collection library to reduce the amount of waste associated with collection expansions. They have designed two new collection libraries for the Pharo programming language that exhibit better resource management than the standard library.

Techniques such as mixin, traits, inter-type declarations of AspectJ, morphing and other meta-programming facilities try to provide composition of collections. However, they don’t have adequate support of composing collections. Mixin [14] supports linear composition, which imposes some restriction on code reuse. Various alternative reuse mechanisms have been proposed, such as traits [15]. These techniques have limitations with respect to encapsulating and sharing state [15], [16]. Also, morphing [16], [17] is a technique to pattern-based reflection. In this technique, reflective patterns reside in mixin-like structures [17].

The above mentioned techniques do not change the way that an object is formed from its components, and the issue of the right high-level programming interface, especially in collections, is not settled yet [2]. So, as [1], [4] state, complex and efficient data structures must be implemented manually, and this leads to the loss of reusability.

3. neoCollection Abstraction

neoCollection is an abstraction to denote a collection. neoCollection abstraction encapsulates a collection including the implementation of its root, node and operations. It is independent from the data which is stored in it. Like a Java class, it has some attributes and methods to define the behaviors.

neoCollection has a special section called element to define the node. This section is a hypothetical type, element, which can be used as a data type to define variables and parameters inside a neoCollection. However, it is not possible to instantiate an object of element type.

Like Java, a neoCollection operation has a pointer, this, to neoCollection itself. But unlike Java nested classes that have some hidden pointers to provide accessibility to the parent class, neoCollection does not have any additional pointers. Therefore, operations of the element section do not have any access to root section and vice versa.

To clarify, code snippet in Fig. 2 defines neoCollection LinkedList. The element section defines LinkedList node. The remove operation takes an argument of type element and removes it from the list.

Unlike conventional collections, a neoCollection cannot be instantiated directly. Instantiation of a neoCollection is done via applying it on a data set. As an example of applying a neoCollection, assume that class Person is defined as in Fig. 3. The following declaration defines a LinkedList of Person objects.

```java
Person[LinkedList()] people;
```
When a neoCollection is applied on a data set, its data type is mapped to the element of neoCollection. Therefore, the hypothetical type (element) is mapped to the actual data type. In the above example, the element is mapped to class Person. No wi f p is an object of class Person, it can be removed from people as follow:

```java
people.remove(p);
```

If it is essential to apply a neoCollection on a subset of data elements, it can be defined by creating a new class that is inherited from the base class and applying neoCollection on it. As an example, if it is needed to have a collection of a subset of Person, it can be removed from people as follow:

```java
people.remove(p);
```

When a neoCollection is instantiated, its abstract key types should be assigned values. The value of a key is a sequence of expressions composed of the data element attributes and literals. The definition of an expression of a key type is embraced in a <> pair. Some examples of key definitions are followed (Person is the base type):

```java
<id>
<lname, fname>
```

The first expression consists of one attribute and the second one consists of two attributes. For instance, applying neoCollection Tree can be done as follows:

```java
Person[Tree(<id>)] people;
```

### 3.1 Key

An important characteristic of collections is key values. To support key values, neoCollection has a special mechanism: hypothetical key type. If a neoCollection has a key, it should define a key type. Inside a neoCollection, the key type is like a usual data type. It can be used to declare variables and arguments. The only attribute of a key type is that it defines a linear order on the elements of the data set. Therefore, it is possible to compare two data elements by their key.

Often it is required to extract the key of a data element. Assume that e is a data element that the key k is defined on it, e.k extracts it. As an example of key type, Fig. 4 has the definition of the binary search tree neoCollection (Tree). It shows that neoCollection Tree has a key type named k. The lookup operation has an argument of type k and finds an element having that key, i.e. e.k == ka.

A neoCollection can have more than one key. As shown in the following code snippet, k1 and k2 are defined as two key types of X.

```java
neoCollection X (key k1, key k2)
{
    /* rest of the code */
}
```

When a neoCollection is instantiated, its abstract key types should be assigned values. The value of a key is a sequence of expressions composed of the data element attributes and literals. The definition of an expression of a key type is embraced in a <> pair. Some examples of key definitions are followed (Person is the base type):

```java
<id>
<lname, fname>
```

The first expression consists of one attribute and the second one consists of two attributes. For instance, applying neoCollection Tree can be done as follows:

```java
Person[Tree(<id>)] people;
```

### 4. neoCollection Composition

neoCollection composition is used not only to create new neoCollections from the available ones but also to apply multiple neoCollections on a set of data elements. It gives complete control over all aspects of the composition. It provides the ability to add new attributes or operations to each component of the composed neoCollection.

A composed neoCollection has a composition expression. The composition expression defines how the composed neoCollection is made up from other neoCollections. It is made from two elementary operators: horizontal composition (hor) and vertical composition (ver).
The horizontal composition is used when more than one neoCollection is applied on a set of data elements. It is similar to a database table which has more than one index. However, it is not necessary to have a data element in all components. A data element can be placed in one or all of them. In the horizontal composition, the node of composed neoCollection is formed by concatenating the elements of the component neoCollections. neoCollection itself is composed by concatenating the component neoCollections.

The code snippet in Fig. 5 provides QT, the horizontal composition of Tree and LinkedList neoCollections. LinkedList has operations to insert an element and delete an element efficiently. However, it does not have an operation to remove an element by key. To support it, Tree is used to find the element and then it is removed from LinkedList and Tree. Each node of QT is a node of LinkedList and Tree. Figure 5 shows remove operation of the composed neoCollection. It operates in O(log n) time (In the current approaches it operates in O(n)).

The vertical composition happens when the elements of a collection are themselves collections. Figure 6 shows parts of a vertical composition. The composition expression indicates that VCompos is a Tree whose nodes are LinkedList. A name, Node is assigned to the composition in the composition expression to provide the capability of adding a new attribute, keynode to it in the body. The implementation provides one operation: insert. It calls lookup of Tree to get the corresponding LinkedList. Then, insert of LinkedList is called to insert the element.

In the above, VCompos has three level of structures: root, intermediate nodes and nodes. At the first level, root, the root of Tree is placed. At the second level, intermediate nodes, the concatenation of Tree node and LinkedList root are placed. At the third level, nodes, the nodes of LinkedLists are placed.

Using horizontal and vertical operators, the composition expression is capable to make any collection from the primitive ones.

To show the power of composition expression consider the following example. In a data management system, a data record may be inserted in more than one indexes. A sample of this structure is shown in Fig. 7. In this collection, each data record is stored in a tree (or B+-Tree) based on its primary key (the left path in Fig. 7) and on a hash of linked lists and ordered lists (the right path in Fig. 7) based on the secondary key. Each linked list and ordered list has records with the same secondary key. When the records of a secondary key are needed, the corresponding linked list is retrieved. Also, when the list of a secondary key ordered by their primary key is needed, the corresponding ordered list is retrieved. The following code snippet shows the composition expression. Indexes gives capabilities of inserting, removing, and searching data in O(log n) time.

5. Formalization

In this section, the main features of neoCollection are formalized by a syntax, as well as its typing rules and semantic rules.

5.1 neoCollection Syntax

Figure 8 shows abstract syntax of neoCollection. The syntax is formalized based on FGJ[18], with differences that extended the FGJ to support neoCollection. In the syntax C, f, m, mc, fk, id is considered for displaying the terms which
are arbitrary selected by the programmer. \( C \) denotes constant class names; \( ms \) denotes constant collection names; \( fk \) denotes key fields and hypothetical key types; \( id \) denotes non-variable method names; \( m \) denotes non-variable method names; The syntax starts with program term \( (P) \) as the start symbol. Overall description of some new non-terminals is as follows.

**neoCollection term (S):** It presents an overall view of neoCollection including of its name, keys, composition expression and body.

**Class term (L):** It presents an overall view of class consisting of its name, its parent, field declarations, constructor and methods declarations. A field declaration declares a field for class \( C \) whose type \( (EC) \) is a neoCollection or primitive type.

**Composition term (CE):** It is a sequence of composition terms. A composition term can be a neoCollection instance \( (mc) \), horizontal composition or a vertical composition. \( mc \) refers to one of the previously defined neoCollections. It should be considered with proper values for its keys. The horizontal compositions are separated by operation \( \text{hor} \) and vertically compositions by operation \( \text{ver} \). Each composition term is embraced within a \( [ ] \) pair.

**Element Section (Elm):** It defines detail of node consists of fields and methods.

In the syntax, angle brackets \( ⟨...⟩ \) indicates that the item within the brackets is optional, wide hat \( \hat{X} \) for repetition zero or more times and superscript \( X \) for comma list repetition zero or more times. Sequences of field declarations, parameter declarations, and operation declarations are assumed to contain no duplicate names.

All keywords of the syntax have already been explained. As seen in Sect. 3, to achieve the flexibility, in the grammar, a name \( (id) \) can be assigned to each component of horizontal compositions, and root, node and particularly to the intermediate nodes of vertical compositions. Later on, inside the body of the composed neoCollection, variables of that type can be declared, and new details can be added to it.

### 5.2 Typing Rules

The basis of the neoCollection type system is type mapping. Every applying of a neoCollection defines a mapping from the types of the neoCollection into the types of the class. As discussed in previous sections, this mapping means that every parts of a neoCollection is a part of objects of the class that the neoCollection is applied on it.

The main typing rules of neoCollection are presented in Fig. 9. The rules (except \( FT_1 \), \( NT_1 \) and \( MT_1 \)) shows type of a term \( (e) \) in the context of applying a neoCollections \( (mc) \) with hypothetical key types \( k \) on a class \( C \) with \( T \) key fields \( (C[mc(T)]) \). In the Fig. 9, \( FT_1 \) to \( FT_{10} \) spec-
ify the type of a field selection. FT\textsubscript{1} is obvious and shows a field type without neoCollection applying. FT\textsubscript{2} specifies the type of a general fields of an applied neoCollection. FT\textsubscript{3} shows the type of a field of hypothetical element type. FT\textsubscript{4} corresponds to vertical composition of neoCollections and shows the type of a hypothetical element type. FT\textsubscript{5} and FT\textsubscript{6} correspond to horizontal composition and show the type of a hypothetical element type in two cases. FT\textsubscript{7} and FT\textsubscript{8} show the type of field selections in element section of neoCollections. FT\textsubscript{9} to FT\textsubscript{10} correspond to hypothetical key type. NT\textsubscript{1} shows the type of a instantiation and MT\textsubscript{1} to MT\textsubscript{7} specify the type of a method call. MT\textsubscript{1} is obvious and shows the type of a method call without neoCollection applying. MT\textsubscript{2} shows the type of a method call when the arguments and the result are normal types. MT\textsubscript{3} shows the type of a method call when the argument is a hypothetical type (element) and the result is a normal type. MT\textsubscript{4} shows the type of a method call when the argument is a normal type and the result is a hypothetical type (element). Finally, the last three rules (MT\textsubscript{5}, MT\textsubscript{6} and MT\textsubscript{7}) specify the type of a method call in neoCollection composition situations.

5.3 Semantics Rules

Before discussing semantics rules of neoCollection, the representation of an object (e\textsubscript{0}) should be discussed. In neoCollection, an object has a set of parts. The first part of an object (C\textsubscript{1}) represents the set of field’s values corresponding to normal instantiation of the object. Other parts (CE\textsubscript{1}, CE\textsubscript{2},...) define the set of fields values corresponding to neoCollections that applied on it. So, an object is shown as follows:

\[ e_0 = \{C_1, CE_1, CE_2, \ldots\} \]

As discussed in Sect. 2, an applying of a neoCollection defines a value mapping from the values of the object such as key values and references into the hypothetical values of the neoCollection. Therefore, the value mapping function can be extended to map various values of an object to the hypothetical values of the neoCollection as follow:

\[ \omega(e_0, \text{U}_1) = v_{u_1} \]

This function gets an object values (e\textsubscript{0}) and a hypothetical value (U\textsubscript{1}) and returns the corresponding value (v\textsubscript{u_1}) in context of the object. Specially, it can be used to retrieve the keys values which correspond to e\textsubscript{0}.

Figure 10 shows the semantics rules of the neoCollection. FS\textsubscript{1} is obvious. FS\textsubscript{2} and FS\textsubscript{3} select a primitive value. FS\textsubscript{4} selects a value of a hypothetical field. MS\textsubscript{1} specifies that before calling a method, the term specifying the object should be evaluated. MS\textsubscript{2} specifies the evaluation order of a method arguments. After them, there are three rules that show the evaluation of a method call. [this ← v\textsubscript{0}, e\textsubscript{1} ← v\textsubscript{1}, ..., e\textsubscript{n} ← v\textsubscript{e_0}] as used in these rules, shows that before calling a method, the term specifying the object (e\textsubscript{0}) and all arguments of the method (e\textsubscript{1}, ..., e\textsubscript{n}) should be evaluated to primitive values (v\textsubscript{0}, v\textsubscript{1}, ..., v\textsubscript{n}). In these rules, the first two ones (MS\textsubscript{3} and MS\textsubscript{4}) show the evaluation when the result of the evaluation is a primitive value. The last one (MS\textsubscript{5}) shows the evaluation when the result is a hypothetical value.

AS\textsubscript{1} and AS\textsubscript{2} specify the way that an assignment is evaluated. Finally, NS\textsubscript{1} shows the evaluation of an instantiation.

6. neoCollection from the Performance Point of View

This section presents experimental test on the performance of the primitive collections consist of the linked list and the tree. As it was mentioned, the current approaches to apply a collection on a set of data elements is to store a reference to the data element in the collection node. In fact, although the correct approach is concatenation, they use pointers for this purpose [19]. Now, neoCollection is implemented based on the concatenating approach, and it solves the issues of referencing approach. As it was mentioned, applying a neoCollection on a set of data elements is done by injecting the element section to the data element. Therefore, it forms a single memory block. In this manner, in the final collection, it is possible to reach the collection node from the data ele-
Fig. 11  The time to execute the linked lists insert operation

Fig. 12  The time to execute the linked lists remove operation

Fig. 13  The time to execute the trees insert operation

Fig. 14  The time to execute the trees remove operation

ment and vice versa in O(1). So, it solves the performance issues mentioned in the introduction. Also, as they are all in one memory block, it solves the cost and memory footprint issues.

To evaluate neoCollection, in this section it is compared with Java and hand-coded implementations. In Java implementation LinkedList and TreeMap is used from Java SE 10. As the time complexity of Java approach is not satisfactory, the proposed collections is implemented in hand-coded. In hand-coded implementation, collections are implemented from scratch.

We perform testing for a variety of collection sizes from 1000 items to 100M items. We use Java Microbenchmark Harness (JMH) test to conduct the test on a four core machine. The results are presented below subsections.

6.1 Benchmark Results

As mentioned, to evaluate neoCollection, we perform testing for the linked list and the tree. This test measures the performance of creating and populating the linked lists and the trees for a specified number of items in neoCollection, Java, and hand-coded implementations.

The time to execute an insert operation of the linked lists is shown in Fig. 11 and the time to execute a remove operation of the linked lists is shown in Fig. 12 in neoCollection, Java and hand-coded implementations. Also, the time to execute an insert operation of the trees is shown in Fig. 13 and the time to execute a remove operation of the trees is shown in Fig. 14 in neoCollection, Java and hand-coded implementations. We tested from 1000 through 100M items in lists as shown on the X-axis. The X-axis is shown in logarithmic scale to highlight the relevant range of time values. The Y-axis is nanoseconds of execute an operation in the average of 1000 runs for a list with selected size. Therefore each operation was performed 1000 times on each list instance and average time of 1000 runs is considered.

6.2 Performance Discussion

From the results in the previous subsection, a clear conclusion can be drawn. neoCollection presents a major boost in performance. This is mainly due to the fact that it uses the concatenating approach, such as uses in hand-coded implementation. They run in the linked list insert and remove operations in O(1) time. In the tree insert and remove operations, they run in O(log n) time such as Java implementation, but because of lower count of pointers, their overhead is lower and can be run more than 50 percent faster than Java implementation.

Also, according to the obtained results, Java general collections have high time complexity. The obtained results show that as the number of items increases, they becomes slower which leads to lower efficiency compared to others.

neoCollection is similar to hand-coded in time complexity. There is no perceptible difference between neoCollection and hand-coded operations time.

7. Implementation Overview

As presented in the previous sections, a general, language-independent approach is developed for collection encapsulation and composition. To demonstrate the feasibility of the idea, implementations of the neoCollection are provided in Java and C++ programming languages as an extension
to Java and C++ compilers. They get a code in the extended languages and produce output in Java or C++ languages. Our current implementations contain complete lexers and parsers, mostly-complete neoCollection-to-C++ and neoCollection-to-Java code generators, and enough semantic analysis to support code generations.

The neoCollection extension adds a stage to the compiler. In this stage, various neoCollections are gathered to form the layout of objects (classes). For each neoCollection a class is defined and the body of the neoCollection is considered as the body of the class.

However, because of the inability to manage pointers in Java, code generation uses class redefinition to inject the element sections to the classes. Also, for each applying of neoCollections a new class is defined as the root of the structure.

By applying a neoCollection on a set of data elements, the compiler concatenates its element section to the end of the data elements and forms the nodes of the structure. So, the neoCollection approach reduces the number of memory blocks and reference traversing. As an example, by applying neoCollection such as LinkedList on a class such as Person, Person objects generated by the compiler has two parts: the first part is the programmer defined class Person and the second part is the element section of LinkedList. Figure 15a shows the compiler generated Person objects.

Implementation of other primitive collections such as tree and hash table also follow a similar structure with the linked list. In fact, they have the same pattern. Of course, the algorithm for implementing operations varies from one collection to another.

Note that applying a neoCollection can also be done in referencing approach. In this case, the programmer uses ref keyword to apply a neoCollection. This will create a new class containing all modified body of the neoCollection instead of concatenating to existing once. See the case study in next section as a sample.

In case of applying more than one neoCollection on a set of data elements, the element sections of neoCollections are concatenated to the data elements. As an example, the following code snippet shows applying of of LinkedList and Tree neoCollections on Person. Figure 15b shows Person objects after concatenation of the element sections.

Person[LinkedList()] people1;
Person[Tree(<id>)] people2;

As it was mentioned, inside a neoCollection, the element type refers to the corresponding element section of the generated structure. Therefore, in Fig. 15a next pointer of an object points to the element section of the next object not the whole object. Similarly, prev pointer of an object points to the element section of the previous object. Outside a neoCollection, the element type refers to the whole object. In this manner, when an operation of a neoCollection which needs a pointer of the element type is called, the programmer provides it a pointer to the whole object. The compiler adjusts the pointer to the element section of the object and passes it to the operation. Similarly, when an operation returns a pointer to the element type, it is a pointer to the element section. The compiler adjusts it to point to the whole object. To clarify, consider the following example where \( p \) is a Person object.

```
Person p;
...
people.put(p);
```

Now, when \( \text{put} \) is called, the compiler calculates the pointer of the element section of \( p \) object and passes it to \( \text{put} \) operation. By managing pointers to sections of an object, element sections of an object or the whole object can be accessed.

8. Case Study

8.1 Order Book

An order book is a list of orders that a stock broker uses to record the interest of shareholders (buyers and sellers) in the stocks. To make a trade the stock broker matches the buy order with the highest price and the sale order with the lowest price of a stock.

Figure 16 shows the proposed structure for managing an order book. It has two paths to an order. The first path (ctree) partitions the orders based on the stocks (stock_key). For every stock, a node of ctree stores buy and sell orders of the stock separately. The buy orders are stored in a max heap (bheap) which is ordered by their price (price_key) and timestamp (ts_key). Similarly, the sell orders are stored in a min heap (sheap). The second path partitions the orders based on shareholders (itree). The shareholder path has a tree indexed by shareholders (sh_key). For every shareholder, a linked list (olist) stores its orders. The opera-

![Fig. 15](image1.png)  Applying neoCollections on class Person

![Fig. 16](image2.png)  The layout of the order book
tions are *addorder*, *cancelorder* and *maketrade*. *addorder* is called when a shareholder enters a new order. *maketrade* is called when a buy order matches a sell order. *cancelorder* is called when a shareholder deletes an order. *cancelorder* gets an order, finds and deletes it from *bheap*, *sheap* and *olist*.

The order book as a complex composition of collections are implemented in the neoCollection, Java and hand coded approaches. Having *Tree*, *LinkedList*, *Min Heap* and *Max Heap* neoCollections, Fig. 17 shows the neoCollection implementation (*OrderBook*).

Table 1 compares the implementations. All operations have $O(\log n)$ time complexity in neoCollection and hand coded implementations. However, the time complexity of *cancelorder* and *maketrade* are $O(n)$ in Java implementation.

In table 1, lines of code means code needed for implementation of business logic. It does not contain lines of code of Java collection classes and does not contain lines of code for pre-defined neoCollections like *Tree* and *LinkedList*. In table 1, lines of code in hand-coded implementation is nearly three times more than other implementations. Also, the lines of code in neoCollection implementation is nearly 15 percent less than that of Java implementation. Large volume of codes in hand-coded implementation makes it difficult to change and increase the complexity and cost of production.

8.2 Observer Pattern

Observer pattern is a software design pattern in which an object (subject) maintains a list of its dependencies (observers) and notifies them usually by calling one of their methods. The implementation of the pattern is provided in Fig. 18.

The pattern is implemented as neoCollection *ObserverPattern*. It is based on neoCollection *LinkedList*. *ObserverPattern* maintains a list of Observers. It has two methods: *subscribe* and *publish*. *subscribe* is used to add an observer and *publish* is called whenever the state of the *ObserverPattern* changes. Finally, each Observer should have a notify method which is called whenever the state of *ObserverPattern* changes. For this purpose, an empty method *notify* is defined in the element section that has no specific body. However, any data element that this neoCollection applies to can override this method and replace its specific body.

Figure 19 shows an example usage of *ObserverPattern*. In this example, there is a *Sensor* and some *Displays*. It
has an applying of ObserverPattern on Displays (sensorOP) which is used to notify Displays whenever the value of Sensor changes. Note that by using ref keyword in defining of ObserverPattern, pointers to Displays are stored in the linked list. In fact, as it is possible that a Display listens to any number of Sensors, thus it is better to store pointers to them in the linked list.

ref is an arbitrary keyword to provide the option to use the conventional referencing approach in neoCollection. In this case, instead of concatenation, a new class is created for the node individually, and references to data element is stored in the node.

9. Conclusion

This paper introduced neoCollection approach to implement collections. The approach introduces a new abstraction to define a collection and two operators to build composite collections. The neoCollection approach provides a program-independent way to the definition and manipulation of collections. It outlined a general, language-independent model of collections encapsulation and composition, and described a proof-of-concept for Java that demonstrates the feasibility of the idea. The approach can be incorporated into a language in such a way that there is an additional precompilation only when one wishes to make use of the approach. In other cases, default libraries mimic the conventional approach offered by the language.

The approach does not make any decision on behalf of the programmer. It is based on simplicity and flexibility. Moreover, high reusability together with efficiency are advantages of the neoCollection approach. The provided case study and other samples show that the approach effectively increase the efficiency and decreases the volume of codes.

References


Davud Mohammadpur received his B.Sc. degree in Software Engineering from Kharazmi University, Tehran, Iran. He received his M.Sc. degree in Software Engineering from Iran University of Science and Technology, Tehran, Iran. He is currently a Ph.D. student at the Department of Information and Communication Technology (ICT), Malek-Ashtar University of Technology, Tehran, Iran.

Ali Mahjur is an Assistant Professor at the Department of Information and Communication Technology (ICT), Malek-Ashtar University of Technology, Iran. He received her B.Sc., M.Sc. and PH.D. in Software Engineering in the Department of Computer Engineering, Sharif University of Technology, Tehran, Iran.