Department of Computer Science
COS121 Lecture Notes: L18

Iterator Design Pattern

8 September 2014

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Contents

L18.1 Introduction

To iterate means to repeat. In software it may be implemented using recursion or loop structures such as `for`-loops and `while`-loops. A class that provides the functionality to support iteration is called an *iterator*.

The term *aggregate* is used to refer to a collection of objects. In software a collection may be implemented in an array, a vector, a binary tree, or any other data structure of objects. The iterator pattern is prescriptive about how aggregates and their iterators should be implemented.

Two general principles are applied in the iterator design pattern. The first is a prominent principle of good design namely *separation of concerns*. The other is a fundamental principle of generic programming namely *decoupling of data and operations*. The iterator design pattern suggests that the functionality to traverse an aggregate should be moved to an iterator while functionality to maintain the aggregate remain the responsibility of the aggregate itself. This way the principle of separation of concerns is applied because the functionality concerned with the maintenance of aggregates is separated form functionality concerned with traversal of the aggregates. At the same time the operation to traverse is decoupled from the data structures which are traversed, leading to the creation of a more generic traversal algorithm.

In this lecture we discuss how separation of concerns leads to better cohesion before we proceed to explain the design and implementation of the iterator design pattern.

L18.2 Iterator Design Pattern

L18.2.1 Identification

Name	**Classification**	**Strategy**
Iterator	Behavioural	Delegation
Intent		
Provide a way to access the elements of an aggregate object sequentially without exposing its underlaying representation ([3]:257)		

L18.2.2 Problem

A system has wildly different data structures that are often traversed for similar results. Consequently traversal code is duplicated but with minor differences because each aggregate has its own way to provide the functionality to access and traverse its objects. To eliminate such duplication, we need to abstract the traversal of these data structures so that algorithms can be defined that are capable of interfacing with them transparently [4].

L18.2.3 Structure

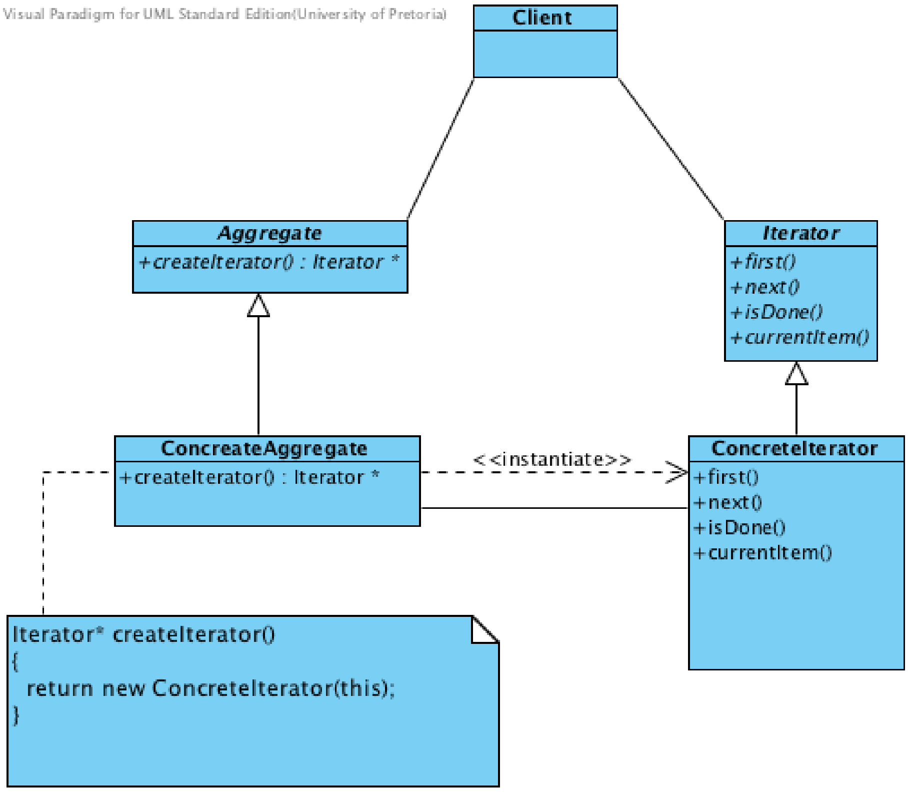
Visual Paradigm for UML Standard Edition(University of Pretoria)
Client
Aggregate
+createIterator() : Iterator *
Iterator
+first()
+next()
+isDone()
+currentItem()
ConcreateAggregate
+createIterator() : Iterator *
<<instantiate>>
ConcreteIterator
+first()
+next()
+isDone()
+currentItem()
Iterator* createIterator()
{
return new ConcreteIterator(this);
}

Figure 1: The structure of the Iterator Design Pattern

L18.2.4 Participants

Iterator

- Defines an interface for accessing and traversing elements.

Concrete Iterator

- Implements the Iterator interface
- Keeps track of the current position in the traversal of the aggregate

Aggregate

- Defines an interface for creating an Iterator object

Concrete Aggregate

- Implements the Iterator creation Interface to return an instance of the proper concrete iterator.

L18.3 Cohesion

When designing a system, it is important to keep its maintenance in mind. Making changes should be easy. One of the design principles that can be applied to avoid the need to change a class, is *separation of concerns*. This means that functionality concerning different aspects should be separated from one another by implementing them in different classes. Thus, the functionality provided by each class in the design should be related to one aspect only. If a single class implements various aspects of functionality, changes in any one of these aspects will result in having to change the class. On the other hand, if a class implements only one aspect of functionality, it will change if and only if that specific aspect changes. Note that the amount of change does not change, only the chance of having to change each class is reduced, meaning that changes are isolated to certain classes.

The term *cohesion* is used to refer to the internal consistency within parts of the design. In object-oriented design the level of cohesion of a system is determined by the level of cohesion of the classes that constitutes the system. A metric to measure the lack of cohesion in methods (LCOM) was proposed by [1]. It is recognised as the most used metric when trying to measure the goodness of a class written in some object-oriented language [5]. When calculating the LCOM of a class, two methods are considered to have a lack of cohesiveness when they operate on disjunct sets of attributes in a class. While it is valid to state that methods are not cohesive when they operate on different attributes, it is not conclusive that they are cohesive if they operate on the same same attributes. Consequently the presence of cohesiveness is much harder to observe than the absence thereof.

We say a class has high cohesion when its methods are related. These methods should be related not only by operating on the same attributes of a class, but more importantly they must be related in terms of the functions they perform. If methods perform functions that are related to different responsibilities, they should be separated by including them in different classes. However, separating responsibility in design is one of the most difficult things to do. Sometimes non-cohesiveness of a class is only realised when the class tends to change more often or in more than one way as the system grows.

Separation of concerns will increase overall cohesion of a system and may reduce the number of classes that has to change when needed. However, it will most likely increase the number of classes in the system. This, in turn will increase complexity as well as coupling between the classes. Finding the best design is illusive. When improving one thing one tends to worsen another! Through experience one learn how to find good solutions that has low coupling without compromising too much in terms of high cohesion.

L18.4 Iterator Pattern Explained

L18.4.1 Design

The Iterator design pattern applies separation of concerns specifically to aggregates. Usually aggregates have at least two functions. One being its maintenance, and the other its traversal. Maintenance of aggregates includes methods to add and remove elements and the like, while traversal of aggregates concerns only accessing the elements and knowing an order in which they should be accessed. The iterator design pattern describe a design that separates the mechanism to iterate through the aggregate from the other functions an aggregate may have.

The Iterator design pattern moves the responsibility of traversing objects away from the aggregate to another class called an iterator. The aggregate class, therefore, can have a simpler interface and implementation because it needs only to cater for maintenance of the aggregate and no longer for its traversal [2].

The iterator design pattern takes this good design a step further. Instead of just implementing every aggregate in two classes (one for maintenance, and one for traversal), this pattern is a design that provides a generic way to traverse the objects in aggregates that is independent of the structure of the various aggregates. This is achieved by defining two abstract interfaces – one for iteration and one for the rest of the functionality of aggregates. This way the system is more flexible when either aggregates or their iterations needs maintenance.

L18.4.2 Improvements achieved

- Iterators simplify the aggregate interface. All the functionality related to access and traversal is removed from the aggregate interface and placed in the iterator interface resulting in a smaller and more cohesive aggregate.

- Iterators contribute to the flexibility of your code – if you change the underlying container, it's easy to change the associated iterator. Thus, the code using aggregates becomes much easier to maintain. Most changes to the internal structure of the aggregates it uses will have no impact on the code that uses the aggregate.

- Iterators contribute to the reusability of your code – algorithms that were written to operate on a containers that use an iterator can easily be reused on other containers provided that they use compatible iterators. Thus, the same code can be used to traverse a variety of aggregate structures in the same application. This reduces duplication of code in applications that manipulate multiple aggregates.

- It is easy to provide different ways to iterate through the same structure for example traversing breadth first or depth first through a game tree or for example to have an iterator that might provide access only to those elements that satisfy specific constraints.

- It is possible to execute simultaneous yet independent iterations through the same structure.

L18.4.3 Disadvantage

A prominent disadvantage of the application of the iterator design pattern is that it becomes complicated the synchronise an aggregate with its iterator. Because the aggregate structure is completely independent of the iteration process, it is thus possible to apply changes to the aggregate while an independent thread iterates through the structure. Such situation is prone to error.

In the example in Section L18.6, `VectorSteppingTool` creates a copy of the state of the aggregate. In this case the iterator can not malfunction even if the aggregate is changed during the iteration process. However, the iteration will complete without reflecting the changes. On the other hand `LinkedListIterator` operates directly on its aggregate. If the `LinkedList` is changed, the iterator will reflect such changes immediately. However, it is prone to error if not synchronised properly. For example, if the current item of the iterator is deleted, the next call to `next()` will cause a segmentation fault. To prevent this the implementation should either disallow the deletion of the current item in all iterations (which might be difficult to implement), or update the current item in all iterations when an item is deleted. To implement this, iterators need to be registered as observers of the delete action – this is also not trivial.

L18.4.4 Real world example

The programming use of a controller to select TV channels provide a practical example of a filtering iterator. The TV set has its built in iterator that scan sequentially through all frequencies. The controller can be programmed to have a specific frequency associated with numbered buttons. This corresponds with a `skipTo()` method that is often implemented in iterators. After the channels have been associated with these numbers, the next and previous buttons can be used to request the next channel, without knowing its number. The use of these buttons correspond with the concept of a special iterator that can be used to step through the channels.

L18.4.5 Related Patterns

Factory Method

Both Iterator and Factory Method use a subclass to decide what object should be created. In fact `createIterator()` is an example of a factory method.

Memento

The memento pattern is often used in conjunction with the iterator pattern. An iterator can use a memento to capture the state of the aggregate. This memento is stored inside the iterator to be used for traversing the aggregate.

Adapter

Both patterns provides an interface through which operations are performed. They differ in the reason for providing this interface. The adapter do it because it would be otherwise impossible while the iterator do it specifically to generalise iteration of aggregates.

Composite

Recursive structures such as composites usually need iterators to traverse them sequentially. Although recursive traversal might be very easy to implement without extending the composite pattern, its is strongly advised to create a composite iterator as discussed in Section 3.

L18.5 Implementation Issues

L18.5.1 Accessing the elements of the aggregate

Concrete iterators must be able to access the elements in the aggregate. The concrete iterator may use this to implement the necessary access in one of the following ways:

- **Make a copy of the aggregate inside the iterator**. This is the most robust solution. This is execution-wise the most efficient, but memory-wise the least efficient. It also has the drawback of not being able to reflect on-the-fly changes to the aggregate.
- **Create an object storing the state of the aggregate inside the iterator**. This more or less boils down to storing a memento (See the memento design pattern) of the aggregate inside its iterator. This is also a robust solution. This might be more efficient than making a copy of the whole aggregate, but not always easy to implement. It suffers the same drawback of not being able to reflect changes to the aggregate that are made after the iterator was created.
- **Keep a pointer to the aggregate inside the iterator and use a call back mechanism to access the elements of the aggregate**. This solution is memory efficient, yet not as robust as the other methods. In this case the methods that needs to be called should be public in the aggregate, or alternatively the iterator can be declared a private/protected friend class of the aggregate and hence be given access to its private/protected methods. This solution will be able to reflect changes that are applied to the aggregate in real time, however it is prone to error if synchronisation between the aggregate and the iterator is not implemented properly. Such close coupling between the aggregate and the iteration also compromises the encapsulation of the aggregate.
- **Use the pimpl[1] principle**. This is the most efficient, both in terms of memory and execution time. It is also robust. How this is done is beyond the scope of this module.

L18.5.2 Additional functionality

The pattern as given in Figure 1, defines a minimal interface to the Iterator class containing only `first()`, `next()`, `isDone()` and `currentItem()`. When implementing the iterator design pattern it might sometimes be handy to implement some of the following additional methods in the interface:

- `remove()` – This method should remove the current item from the aggregate. It provides the means to synchronise the maintenance of the aggregate with its iterator by using a double dispatch[2].

[1] pointer to implementation

[2] More detail on the double dispatch mechanism is discussed in L34_Visitor

- `previous()` – This method should step backwards instead of forwards to enable iterations that can go in both directions. If this is supported one should also implement to different methods for the prescribed `isDone()`. One for reaching the end while moving forward, and one for reaching the beginning while moving backwards. This is usually implemented using method names like `hasNext()` and `hasPrevious()`.

- `skipTo()` – This method should position the iterator to an object matching specific criteria. This operation may be useful for sorted or indexed collections to enable the implementation of more complicated algorithms to operate on the aggregate. Examples of algorithms that may need this operation are binary search and quick sort.

L18.5.3 Internal and External iterators

Iterators are classified as either internal or external depending on who calls the methods that are declared for accessing and traversing the aggregate. If the client calls these methods the iterator is said to be external. An internal iterator, on the other hand, is controlled by the iterator itself. In this case the methods for traversing the aggregate can be declared private to prevent the client from calling them. Internal iterators are less flexible than external iterators because when it is the iterator that is stepping through the aggregate, you have to tell the iterator what to do with the elements while stepping through them. This means that you also need some way to pass an operation to the iterator.

L18.5.4 Allocation on the heap or on the stack

Iterator object may be allocated dynamically on the heap. This allows for the creation of polymorphic itertators. In this case the client is responsible for deleting them. This is error-prone, because it's easy to forgetto free a heap-allocated iterator object when you're finished with it. Hence they should be used only when there's a need for polymorphism. A more robust solution would be to allocate concrete iterators on the stack.

An alternative that provides the flexibility offered by heap allocation as well is the stability achieved through stack allocation offered by [3] is to apply the proxy design pattern to implement a stack-allocated proxy as a stand-in for the real iterator. The proxy can delete the iterator in its destructor. Thus when the proxy goes out of scope, the real iterator will get deallocated along with it. The proxy ensures proper cleanup, even in the face of exceptions.

L18.5.5 Using C++ STL Iterators

The application of the iterator design pattern was taken seriously by the designers of the C++ language. Standard iterators are implemented for the containers in the C++ STL, such as `vector<>, list<>, stack<>` and `map<>`. When using STL containers it is advisable to use the provided iterators instead of a counter to traverse such container to gain the benefits of using the iterator design pattern mentioned in Section L18.4.2. They may also be able provide a way to access the data in an STL container that don't have obvious means of accessing all of the data (for instance, maps).

When one use one of these containters, a variable of the container type is declared by including the type of the objects as a template parameter. For example use the following syntax to declare a vector of integers called `myVector`:

```
vector<int> myVector;
```

To declare an iterator appropriate for a particular STL templated class, you use the following syntax

`std::`*class_name*<*template_parameters*>`::iterator` *name*

where *name* is the name of the iterator variable you wish to create and the *class_name* is the name of the STL container you are using, and the *template_parameters* are the parameters to the template used to declare objects that will work with this iterator. Note that because the STL classes are part of the std namespace, you will need to either prefix every container class type with `std::`, or include `using namespace std;` at the top of your program. For example you can create an iterator for the vector `myVector` that was declared in the above mentioned example as follows:

```
std::vector<int>::iterator myIterator;
```

The two loops in the following code fragment are functionally equivalent. The first uses an integer counter to iterate through the vector that was declared in the above mentioned example, while the second uses the iterator that is declared here:

```
for(int myCounter = 0;
        myCounter< myVector.size(); myCounter++)
        cout << myVector[myCounter] << '\t';

for(myIterator = myVector.begin();
        myIterator < myVector.end(); myIterator++)
        cout << *myIterator<< '\t';
```

Note how the elements of the vector are accessed by using the `operator[]` in the first loop, while they are accessed by dereferencing the iterator in the second loop. To move from one element to the next, the increment operator, ++, is used in both cases. Iterators overload all operators. One can use the standard arithmetic shortcuts such as `--, +=` and `-=`, and also use `!=, ==, <, >, <=,` and `>=` to compare iterator positions within the container.

The following are some pitfalls to watch out for when using STL iterators:

- Iterators do not provide bounds checking; it is possible to overstep the bounds of a container, resulting in segmentation faults

- Different containers support different iterators, so it is not always possible to change the underlying container type without making changes to your code

- Iterators can be invalidated if the underlying container (the container being iterated over) is changed significantly

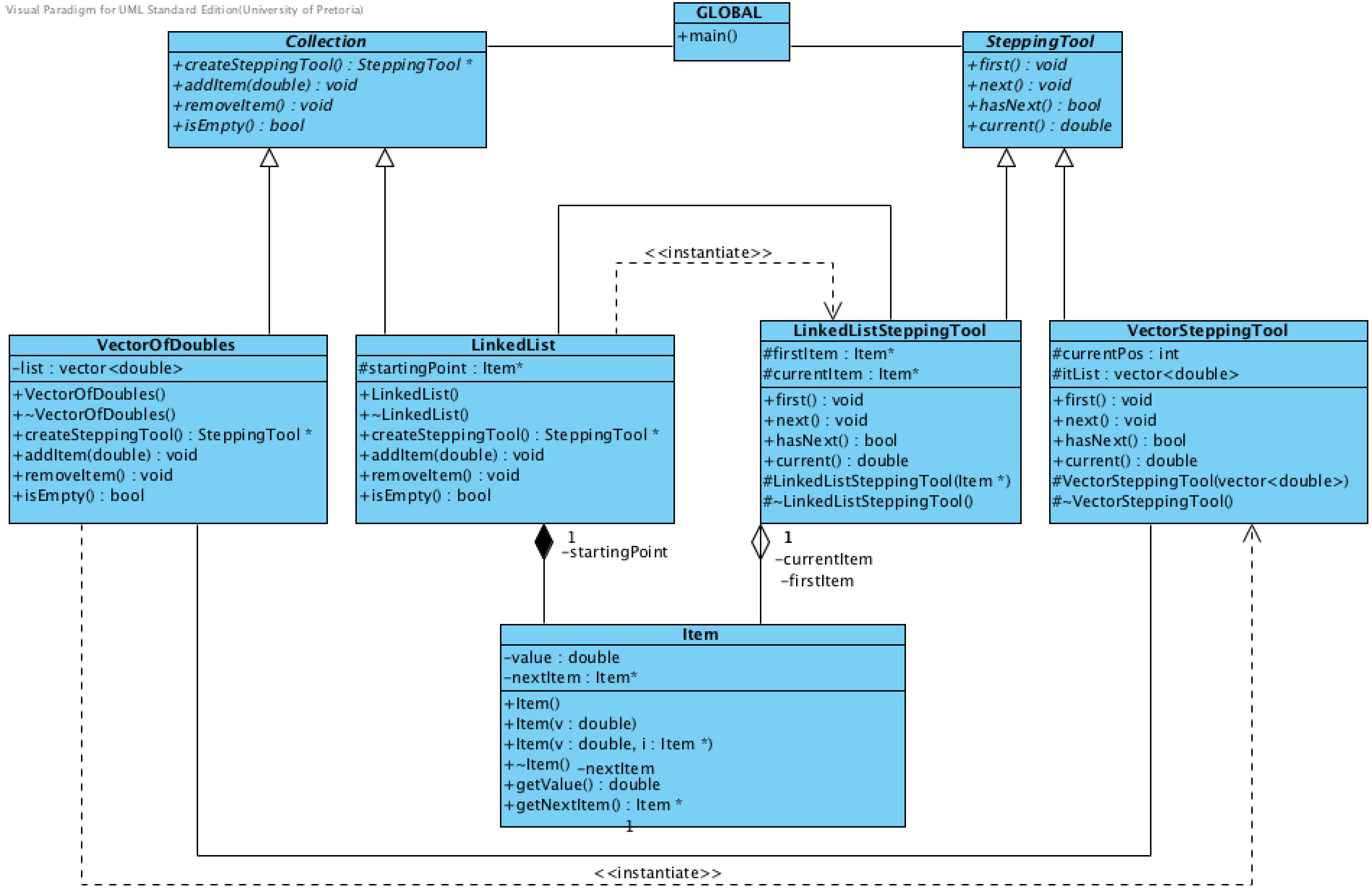

Figure 2: Class Diagram of a system illustrating the implementation of the iterator design pattern

L18.6 Example

Figure L18.5.5 is a class diagram of an application that implements the iterator design pattern. It implements two data structures and their respective iterators. The main program uses the same code to manipulate any one of these data structures. It also shows how two independent iterators can be used to traverse the same structure at the same time. The two data structures are a vector and a singly linked list.

Participant	**Entity in application**
Iterator	SteppingTool
Concrete Iterators	VectorSteppingTool, LinkedListSteppingTool
Aggregate	Collection
Concrete Aggregates	VectorOfDoubles, LinkedList
createIterator() :Iterator*	createSteppingTool():SteppingTool*
first(), next(), isDone(), currentItem()	first(), next(), hasNext():bool, current():double
Client	main()

Iterator

- The `SteppingTool` class acts as the iterator interface.
- It defines an interface for accessing and traversing elements exactly as prescribed by the pattern.

Concrete Iterator

- The concrete iterators that are implemented are `VectorSteppingTool` and `LinkedListSteppingTool`.
- Each concrete iterator has an instance variable to enable it to access the elements of its corresponding aggregate. In the case of the `VectorSteppingTool` the whole vector is duplicated[3]. In the case of `LinkedListSteppingTool` it only needs a pointer to its first element, because the rest of the list can be accessed by following the links in its items.
- Each concrete iterator has an instance variable to enable it to refer to the current item during traversal. In the case of the `VectorSteppingTool` it is an integer holding the index value of the current position. In the case of `LinkedListSteppingTool` it is a pointer to the current node in the linked list.
- Each concrete iterator implements the methods defined in the `SteppingTool` interface.

Aggregate

- The `Collection` class acts as the aggregate interface.
- It defines the `createSteppingTool()` method that ensures that an iterator can be created for each concrete aggregate as prescribed by the pattern.

[3] It might have been a better idea to keep only a pointer to the original vector. This way less memory will be used and dynamic changes to the vector can be supported

- It also defines methods to be able to maintain concrete objects of classed derived from this interface. It supports only one insertion and a default deletion of elements, as well as a means to determine if the collection is empty.

Concrete Aggregate

- The concrete aggregates are `VectorOfDoubles` and `LinkedList`.
- Each concrete aggregate implements a default constructor and destructor.
- Each concrete aggregate implements the creation of its specific iterator as prescribed by the pattern. In the case of the `VectorOfDoubles` the vector is passed as parameter to the constructor of `VectorSteppingTool` which in turn makes a copy of it. In the case of `LinkedList` a pointer to the head of the list is passed to the constructor of `LinkedListSteppingTool`. Note that both these cases do not pass a pointer to itself to the constructor of its iterator as suggested in the pattern definition. Instead the pass only the data needed by the iterator to be able to operate.
- `VectorOfDoubles` implements the other methods defined in the `Collection` interface to maintain the collection simply by delegating the actions to the appropriate methods of the `<vector>` class.
- `LinkedList` implements the other methods defined in the `Collection` interface to maintain the collection by creating and deleting `Item` objects and setting their pointers appropriately to maintain a singly linked list. Note in the class diagram that the `LinkedList` has the responsibility to create and delete `Item` objects, while the `LinkedListSteppingTool` only reads them and should not delete them on destruction.

Client

- In this application the client has a `VectorOfDoubles` object called `myCollection` which is initiated with the first 10 integers. Two instances of `VectorSteppingTool` is then used to iterated differently through `myCollection`. The programmer only need to change one line of code to change the instance of `VectorOfDoubles` to an instance of `LinkedList`. Owing to the implementation of the iterator design pattern, the use of the correct iterator for this `LinkedList` will be instantiated without having to change any other code.

The C^{++} code of this example can be found in a tarball called `L18_Iterator.tar.gz`.

L18.7 Exercises

1. Change the test harness (main program) of the system given in Section L18.6 to allow the insertion and deletion of nodes during iteration and observe the impact. (See Section L18.4.3).

2. Add a structure that stores a binary tree of double values to the system given in Section L18.6. The values should be inserted in such a way that the left child of every parent is smaller than its parent and the right child is larger than its parent.

Duplicates should be ignored. Implement different concrete SteppingTools to allow pre-order, in-order, and post-order traversal of your binary tree.

Write a new test harness. This program should insert random double values simultaneously into a `VectorOfDoubles` and into your binary tree in the order they are generated. Use a `VectorSteppingTool` to display the vector and your different binary iterators to show the different traversals of your binary tree.

3. Implement an iterator for the composite in L13_Composite

   - Implement the `createIteror`-method as an operation of the composite design pattern.
   - Define a composite iterator class as a concrete iterator in the Iterator design pattern.
   - Implement the operations that are defined in the Iterator to be recursive when it is the iterator of a composite.

References

[1] S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476 –493, jun 1994.

[2] Eric Freeman, Elisabeth Freeman, Bert Bates, and Kathy Sierra. *Head First Design Patterns*. O'Reilly Media, Sebastopol, CA95472, 1 edition, 2004.

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley, Reading, Mass, 1995.

[4] Vince Huston. Design patterns. `http://www.cs.huji.ac.il/labs/parallel/Docs/C++/DesignPatterns/`, n.d. [Online: Accessed 29 June 2011].

[5] Sami Mäkelä and Ville Leppänen. Observation on lack of cohesion metrics. In *Proceedings of the International Conference on Computer Systems and Technologies*, CompSysTech06. 2006.