**Create a general purpose GUI component**

## Design Problem

Class *ButtonItemDispaly* has to know the item collection mechanism for traversal.

If the item collection type is changed, the client (Class *ButtonItemDispaly*) has to change the source code to traverse it. Current design didn’t provide a uniform interface for traversing different collection, so we cannot create a general purpose GUI component that will be able to iterate through any collection of the application.

## Candidate design patterns

1. CompositeIntent: Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
2. FactoryIntent: Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
3. IteratorIntent:

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Provide a uniform interface for traversing different collection.

Allows to modify the collection implementation without making any changes outside of collection

Enables to create a general purpose GUI component that will be able to iterate through any collection of the application

Selected pattern:Iterator

## Motivation

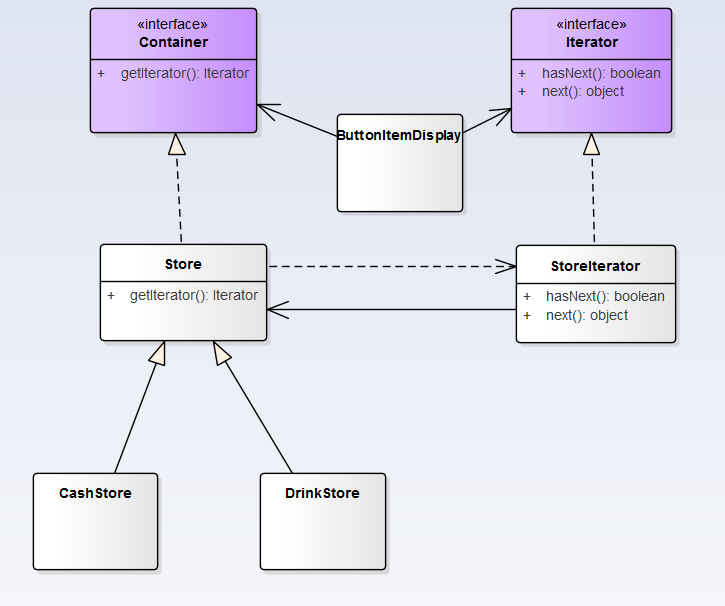
One of the most common data structures in software development is what is generic called a collection. A collection is just a grouping of some objects. They can have the same type or they can be all cast to a base type like object. A collection can be a list, an array, a tree and the examples can continue.

But what is more important is that a collection should provide a way to access its elements without exposing its internal structure. We should have a mechanism to traverse in the same way a list or an array. It doesn't matter how they are internally represented.

The idea of the iterator pattern is to take the responsibility of accessing and passing through the objects of the collection and put it in the iterator object. The iterator object will maintain the state of the iteration, keeping track of the current item and having a way of identifying what elements are next to be iterated.

## 

## Class diagram



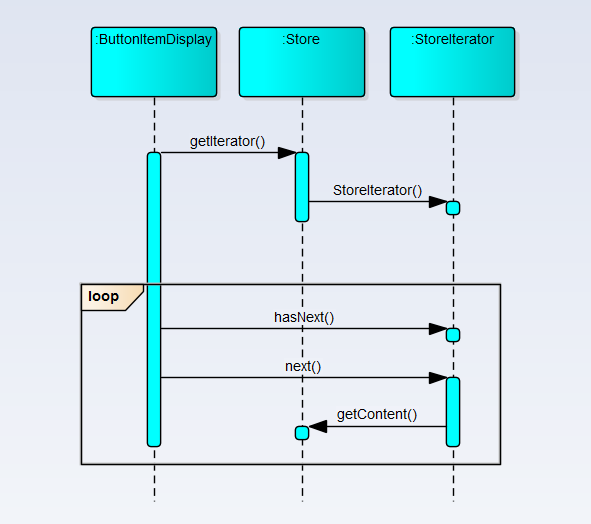
Iterator - This interface represent the Abstract Iterator, defining the iterator

StoreIterator - This is the implementation of Iterator (implements the Iterator interface)

Container - This is an interface defining the Aggregate

Store- An implementation of the Container

### Sequence Diagram



### Implementation

**External vs. internal iterators.**

External Iterators - when the iteration is controlled by the collection object we say that we have an external Iterator.

In our implementation an external iterator is implemented. In the following code an external iterator is used:

*while(iterator.hasNext())*

*{*

*StoreItem storeItem =(StoreItem) iterator.next();*

*StoreObject ob = storeItem.getContent();*

*}*

Internal Iterators - When the iterator controls it we have an internal iterator

When an internal iterator is used it means that the code is be run is delegated to the aggregate object.

The main idea is to pass the code to be executed to the collection. Then the collection will call internally the doSomething method on each of the components. In C++ it's possible to send the doMethod method as a pointer. In C# .NET or VB.NET it is possible to send the method as a delegate. In java the Functor design pattern has to be used. The main idea is to create a base Interface with only one method (doSomething). Then the method will be implemented in a class which implements the interface and the class will be passed to the collection to iterate.

### We use External Iterators in the Implementation

**Who defines the traversal algorithm?**

The algorithm for traversing the aggregate can be implemented in the iterator or in the aggregate itself. When the traversal algorithm is defined in the aggregate, the iterator is used only to store the state of the iterator. This kind of iterator is called a cursor because it points to the current position in the aggregate.

The other option is to implement the traversal algorithm in the iterator. This option offers certain advantages and some disadvantages. For example it is easier to implement different algorithms to reuse the same iterators on different aggregates and to subclass the iterator in order to change its behaviour. The main disadvantage is that the iterator will have to access internal members of the aggregate. In Java and .NET this can be done, without violating the encapsulation principle, by making the iterator an inner class of the aggregate class.

In our implementation, we use inner class for the iterator, and define traversal algorithm in the iterator.

*private class StoreIterator implements Iterator {*

*int index;*

*@Override*

*public boolean hasNext() {*

*if(index < items.length){*

*return true;*

*}*

*return false;*

*}*

*public Object next() {*

*if(this.hasNext()){*

*return items[index++];*

*}*

*return null;*

*}*

*}*

**Robust Iterators**

A robust iterator is that elements are not accessed twice or ignored when elements are added or removed from an aggregate during iteration.

In our implementation, we don't need a robust iterator. We can just throw an exception when an element is added to the collection stating that the operation has failed.

**Iterator and multithreading**

In our implementation, the collection return a new iterator (using in our example the getIterator method). Usually this step is not affected when it is used in multithreading environments because it returns a new iterator object.

Step two: The iterator is used for iterating through the objects. Since the iterators are different objects this step is not a problematic one in multithreading environments.

It seems that the iterator does not raise special problems when a collection is used from different threads.

.