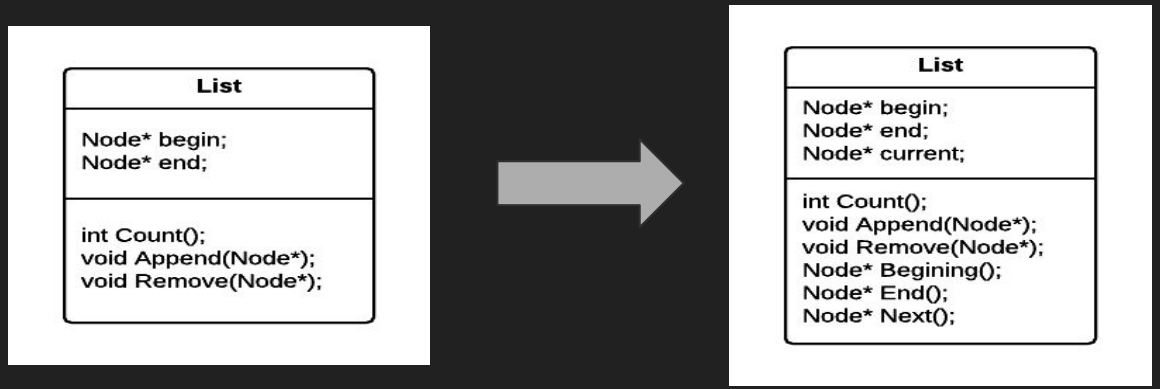
Iterator Pattern

Behavioral, Object focused

Motivation

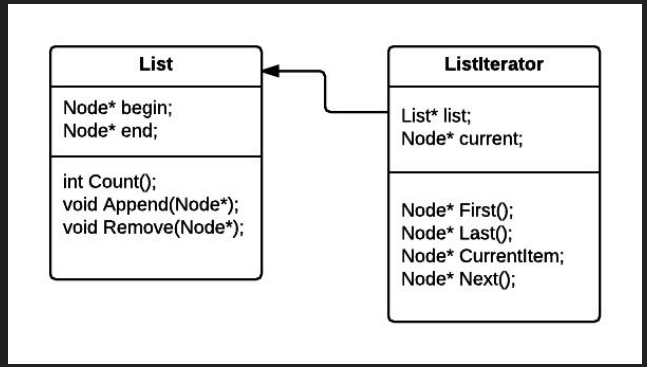
- Adding access and traversal functions to a list object can become burdensome
- Additionally, the same list may need to be iterated over in several ways by several separate objects

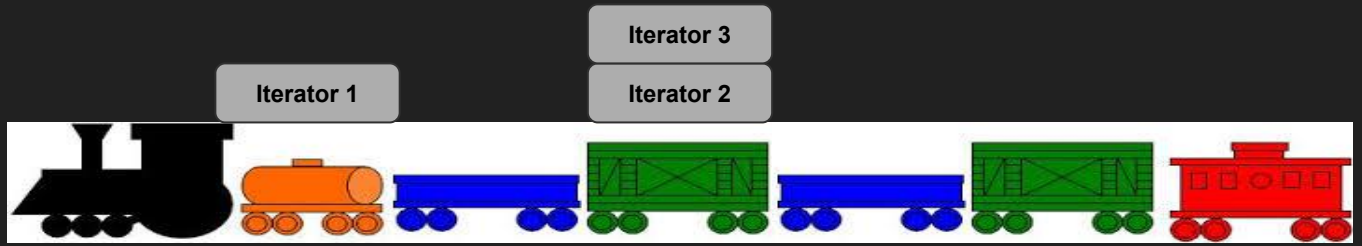


What if we want multiple traversals?

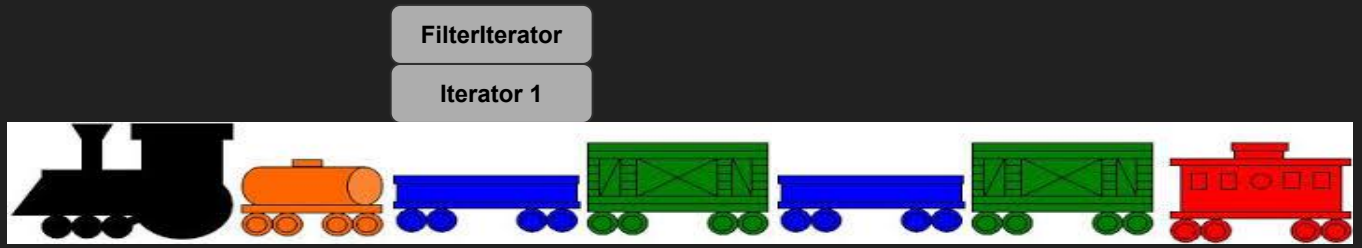
Motivation

- Extract the `Iterator` as its own object
- Now there can be multiple iterators for the same list!
- Different iterators can iterate in different fashions

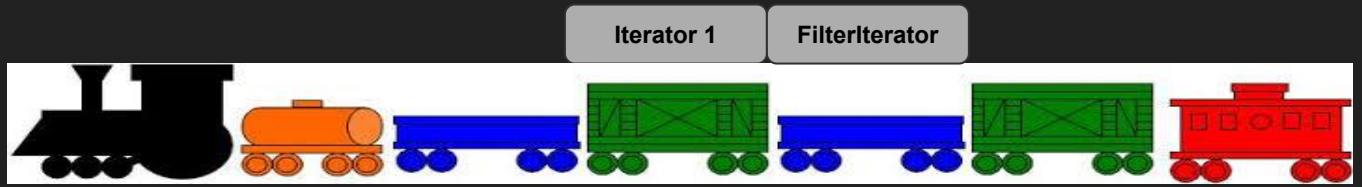




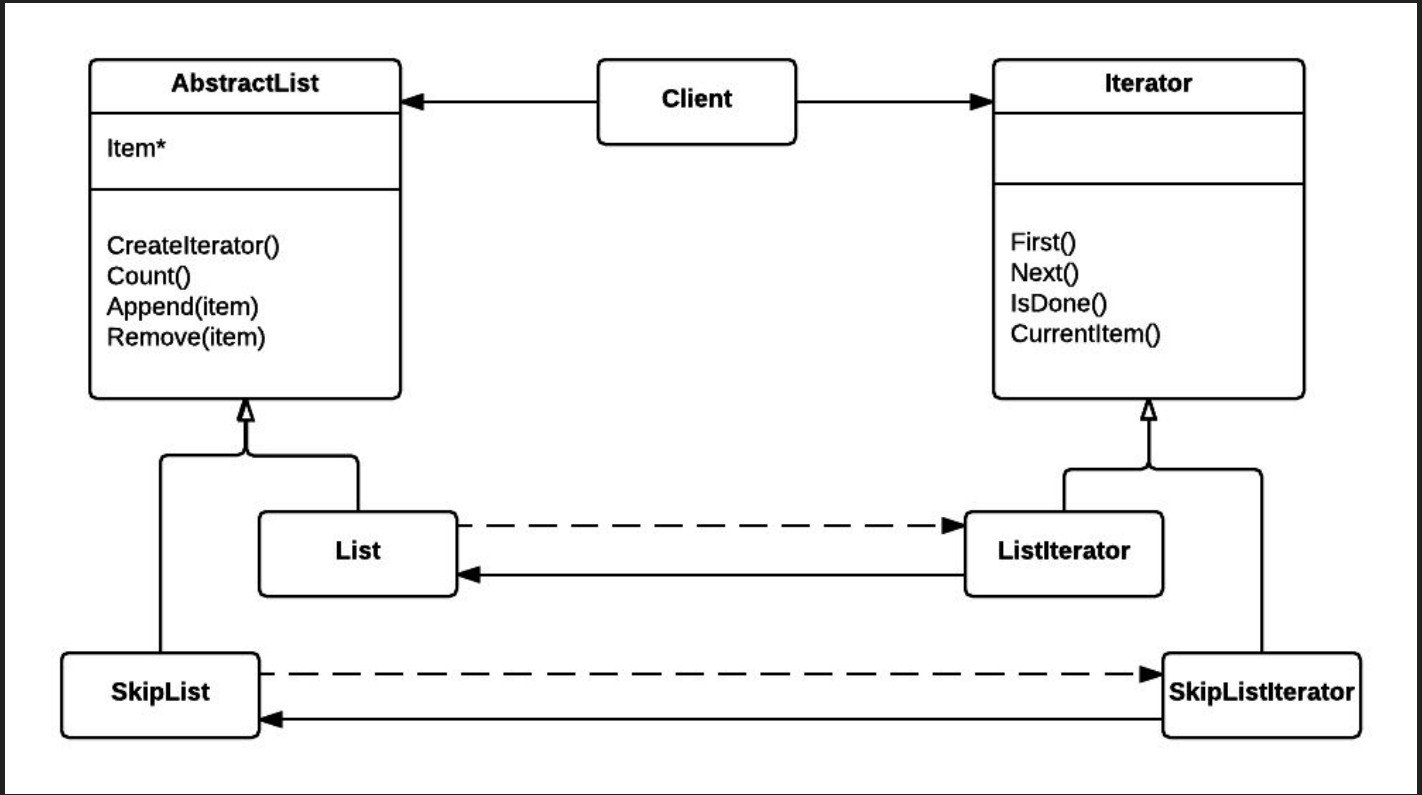
Motivation



- `FilterIterator->next()` moves the iterator to the next **blue** car
- `Iterator->next()` moves the iterator to the next car

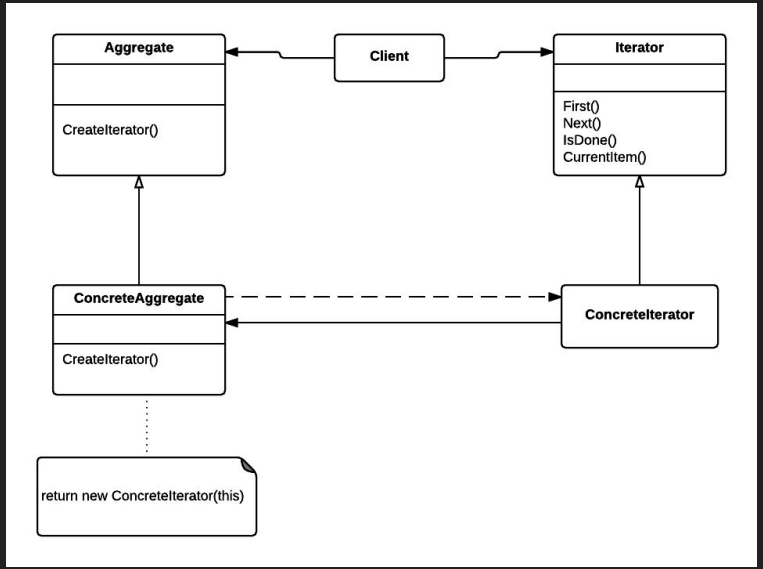


Motivation


AbstractList
Item*
CreateIterator()
Count()
Append(item)
Remove(item)
Client
Iterator
First()
Next()
IsDone()
CurrentItem()
List
ListIterator
SkipList
SkipListIterator

Structure

- **Aggregate -** Defines the interface for creating an `Iterator` object
- **ConcreteAggregate -** Implements `Iterator` creation
- **Iterator -** Defines the interface for accessing and traversing elements
- **ConcreteIterator -** Implements `Iterator`



Consequences

- **Pros:**
  - Supports variations in the traversal of an aggregate. Remember the Iterator and FilterIterator from the train example
  - Iterators simplify the Aggregate interface. The Aggregate no longer needs to support the many alternatives for traversing itself.
  - Multiple traversals can be pending on any given aggregate. The traversal is no longer maintained by the aggregate, multiple iterator objects can be instantiated
- **Cons:**
  - Nothing really (space & “complexity”), this is why it’s a part of the STL for most every container

Example from “Design Patterns: Elements of Reusable Object-Oriented Software”

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

Spell Checking and Hyphenation

- There are many methods for performing spell checking and hyphenation
- We don’t want to hard code this functionality into the document structure
  - There may be other analysis we want to perform and changing the `Glyph` class every time is impractical

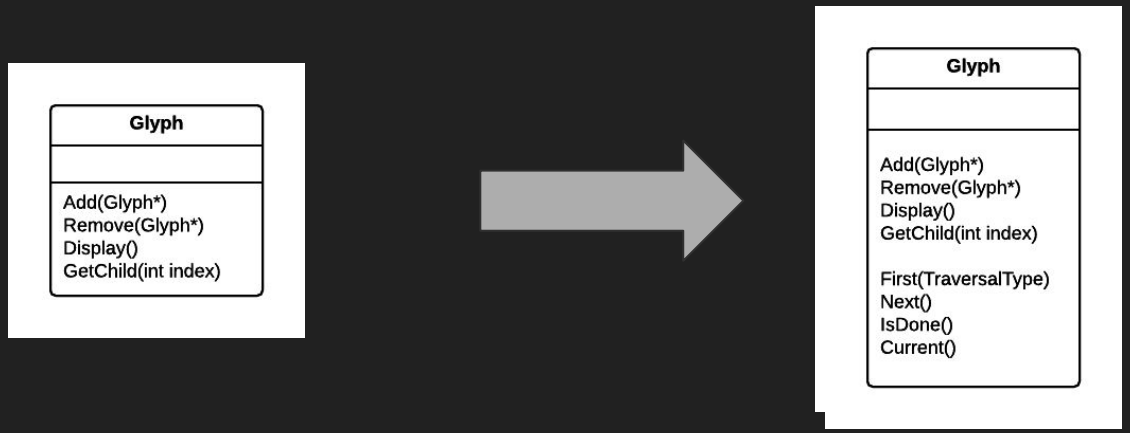
- There are two pieces involved in solving the analysis problem:
  1. Accessing the information to be analyzed
  2. Doing the analysis on that information

Accessing Scattered Information

- Many different kinds of analysis require examining the text character by character
- The access method needs to have knowledge about the structure the text is stored in
- Different traversal methods need to be supported for different types of analysis
  - Preorder
  - Postorder
  - etc.

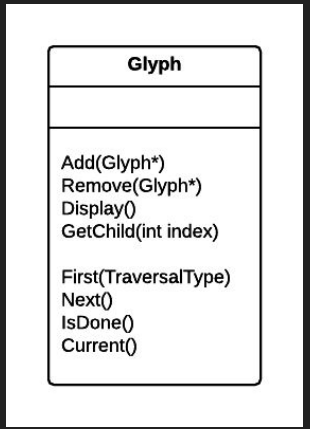
Encapsulating Access and Traversal in `Glyph`

- Currently, the `Glyph` interface uses an integer index to refer to children
- We can augment the Glyph class to have multiple access and traversal capabilities, and provide a way to choose between them
- This can be specified using an `enum` which is passed in to ensure all glyphs are doing the same kind of traversal



Encapsulating Access and Traversal in `Glyph`

- Operations `First`, `Next`, and `IsDone` control the traversal
  - **`First`** initializes the traversal with a specific `TraversalType`
  - **`Next`** advances to the next glyph in the traversal, based on the `TraversalType`
  - **`IsDone`** reports whether or not the traversal is over or not




```
The following code could be used to the traverse the document (rooted at g)

Glyph* g; // Root of the document
for (g->First(PREORDER);!g->IsDone; g->Next()) {
     Glyph* current = g->getCurrent();
     //Do some analysis
}
```

Encapsulating Access and Traversal in `Glyph`

- The interface is no longer biased towards a particular kind of collection
- The client doesn't have to implement common traversals, the `Glyph` class does it now
- **However**, there are some problems with this implementation
  - Adding new traversals require extending enumerated values and/or adding new operations
  - Moving the traversal mechanism into the Glyph class hierarchy makes it hard to modify or extend without causing a lot of classes to need redesign

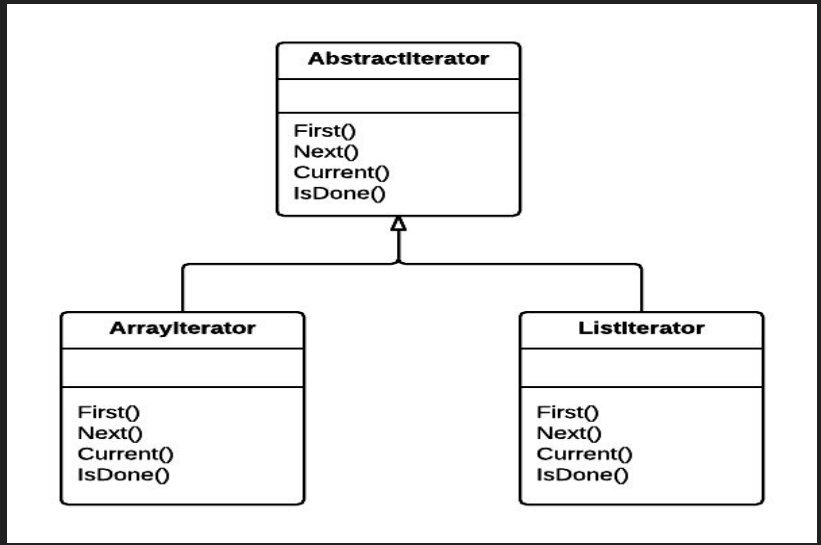
We always try to reduce the need for code redesign

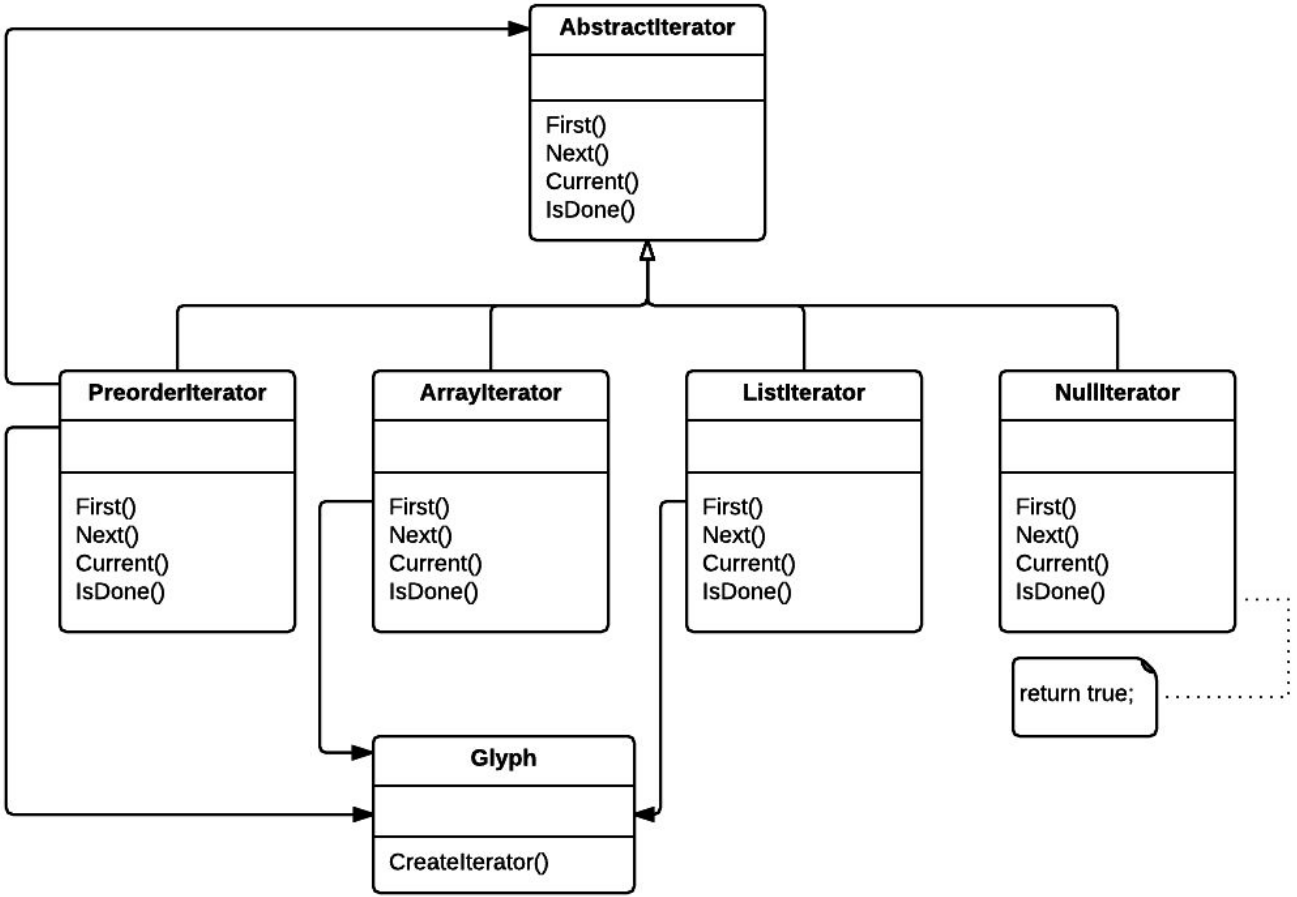
Encapsulating Access and Traversal

- Let's instead encapsulate the concept that **varies**, in this case the access and traversal mechanism!
  - Notice a *pattern* yet?
- Introducing an **Iterator** class of objects whose sole purpose is traversal and access using different mechanisms
- Inheritance will allow us to access different data structures uniformly and support new kinds of traversals as well!

Iterator Class and Subclasses

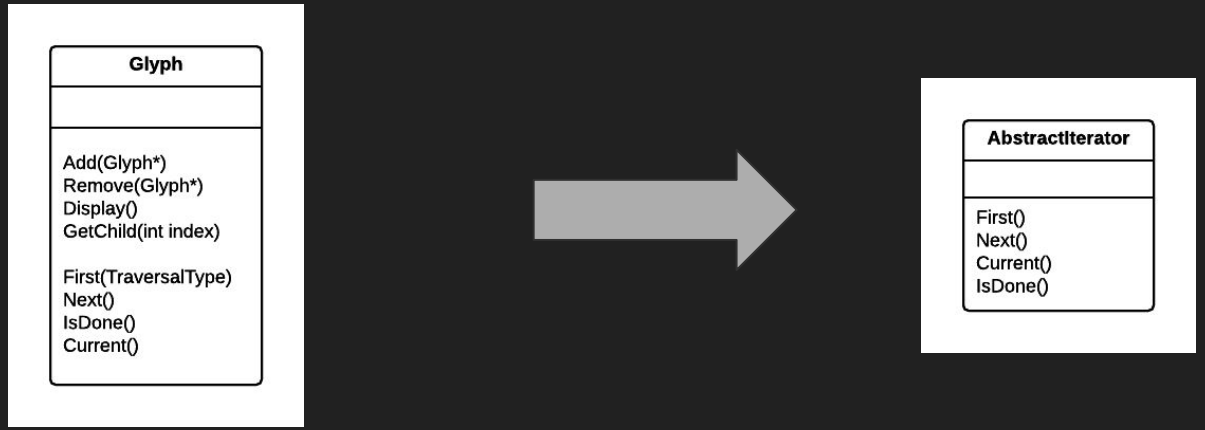
1. First, let's create an abstract `Iterator` class to define a general interface
2. Now we can have concrete `Iterator` **subclasses** such as `ArrayIterator` and `ListIterator` to implement the given common interface




AbstractIterator
First()
Next()
Current()
IsDone()
PreorderIterator
First()
Next()
Current()
IsDone()
ArrayIterator
First()
Next()
Current()
IsDone()
ListIterator
First()
Next()
Current()
IsDone()
NullIterator
First()
Next()
Current()
IsDone()
return true;
Glyph
CreateIterator()

Iterator Class and Subclasses

- Each iterator subclass will have a reference to the structure it traverses
- Subclass instances are initialized with this reference when they are created (by the structure itself)
- The iterator interface now provides the **`First`**, **`Next`**, **`IsDone`**, and **`Current`** functions for controlling the traversal



Iterator Class and Subclasses

- **`First`** points to the first element in the structure, the list in the case of `ListIterator`
- **`Next`** advances the iterator to the next item in the list
- **`IsDone`** returns whether or not the list pointer points beyond the last element in the list
- **`Current`** dereferences the iterator to return the `Glyph` it points to

Iterator Class and Subclasses

- Now we can access the children of a glyph structure without knowing it's representation:

```
Glyph* g;
Iterator<Glyph*>* iter = g->CreateIterator();
for (iter->First();!iter->IsDone();iter->Next()) {
    Glyph* child = iter->Current();
    //Do some analysis on the current child
}
```

Iterator Class and Subclasses

- The **`CreateIterator`** returns a **`NullIterator`** instance by default
- The **`NullIterator`** is a degenerate iterator for glyphs with no children (leaf glyphs)
- **`NullIterator->IsDone()`** will always return true (because leaf glyphs cannot be traversed)
- A glyph subclass that has children will override the `CreateIterator` to return a different iterator subclass
- If a `Row` subclass of Glyph stores its children in a list `children`, then it's `CreateIterator` operation would be:

```
Iterator<Glyph*>* Row::CreateIterator() {
    return new ListIterator<Glyph*>(children);
}
```

Extending Iterators

- Iterators for **preorder** and **inorder** traversals implement their traversals on top of glyph-specific iterators
- The iterators are supplied by the root glyph in the structure they traverse
- The `CreateIterator` called on the glyph in the structure provides an iterator, and they can use a stack to keep track of resulting iterators

Extending Iterators

- The **PreorderIterator** gets the iterator from the root glyph, initializes it to point to its first element, and then pushes it onto the stack:

```
void PreorderIterator::First() {
    Iterator<Glyph*>* iter = this->root->CreateIterator();
    if (iter) {
        iter->First();
        this->iterators.RemoveAll();
        this->iterators.Push(iter);
   }
}
```

Extending Iterators

- The **`Current`** function would simply call the `Current` function for the iterator on the top of the stack:

```
Glyph* PreorderIterator::Current() const {
    if (this->iterators.Size() > 0) {
        return this->iterators.Top()->Current();
    } else {
        return NULL;
    }
}
```

Extending Iterators

- The **`Next`** operation
  1. gets the top iterator on the stack and asks its current item to create an iterator so we can descend the glyph structure
  2. Sets the new iterator to the first item in the traversal and pushes it onto the stack
  3. Tests the latest iterator; if it's `IsDone` operation returns true, then we've finished traversing the subtree (or leaf)
  4. Pops the top iterator off the stack and repeats this process until it finds the next incomplete traversal, if there is one (otherwise we're done)

Extending Iterators

```
void PreorderIterator::Next() {
    Iterator<Glyph*>* iter =
          this->iterators.Top()->Current()->CreateIterator();
    iter->First();
    this->iterators.Push(iter);
    while (this->iterators.Size() > 0 &&
          this->iterators.Top()->IsDone()) {
        delete this->iterators.Pop();
        this->iterators.Top()->Next();
    }
}
```

Iterator Pattern Review

- The **Iterator Pattern** captures techniques for supporting access and traversal over object structures
- It's applicable not only to composite structures but to collections as well
- It abstracts the traversal algorithm and shields clients from the internal structure of the objects they traverse