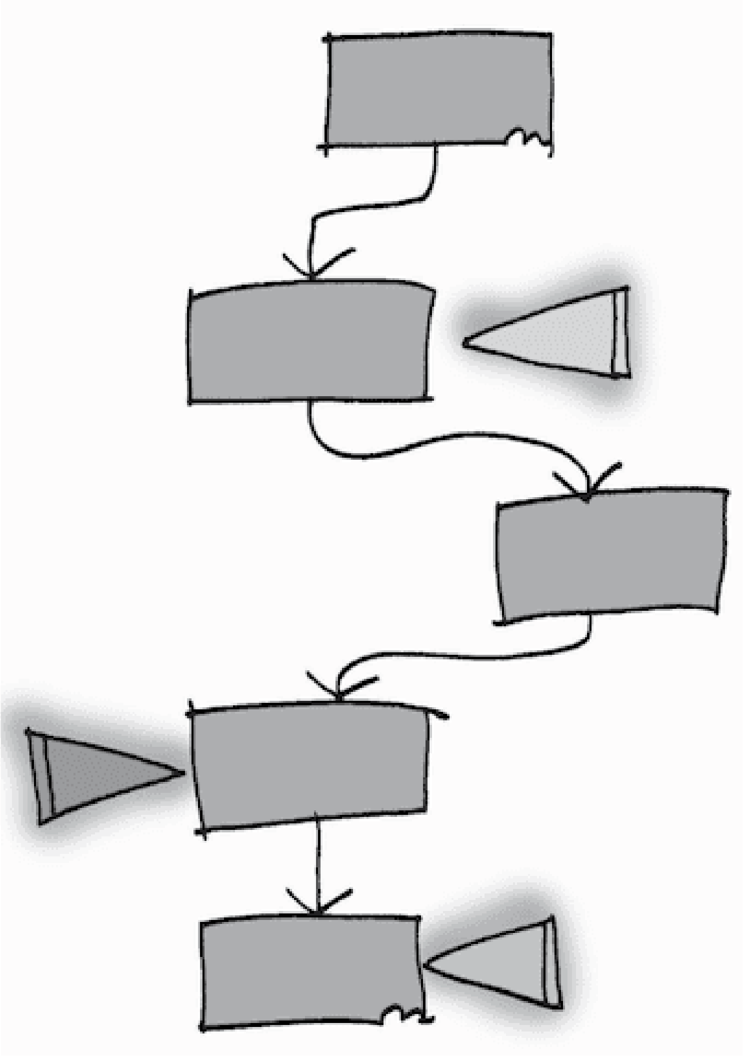
Iterator Pattern

Repetition of Processing



Iterator Pattern

We need to iterate over an aggregation without knowing the details of the aggregation.

- The Spotify or Apple Music playlist is an aggregate, and we can easily iterate over them simply by scrolling.
- We don't need to know any detailed knowledge of the data structure.

The Problem

```
for i in [1,2,3,4,5]:
  print(i)
```

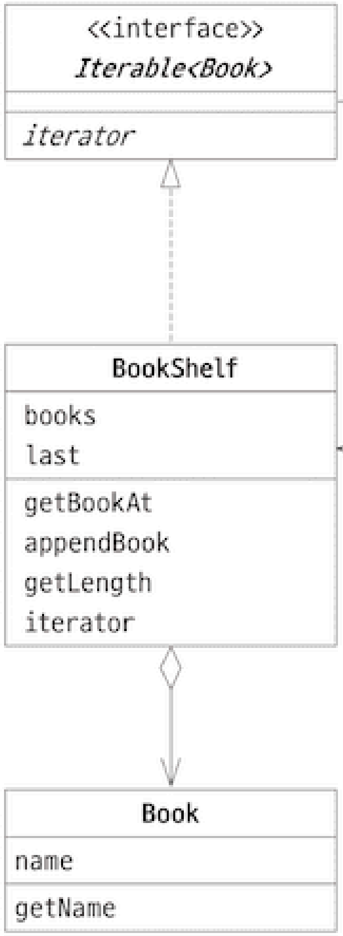
- The variable **i** is used to access values in a **list** (or aggregation of integer values).
- This code is smelly as we need to know too many details, such as an index or list information.

The challenge: how to **access elements** of a collection **sequentially** without exposing the **internal structure** of the aggregation?

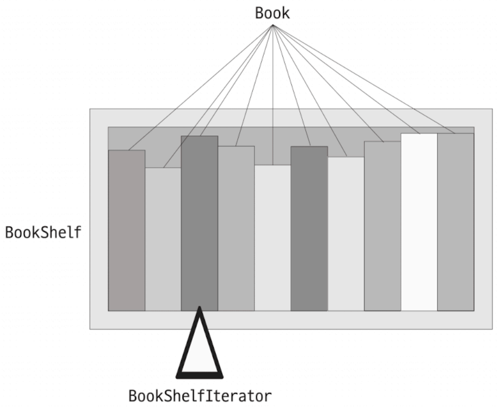
The *Iterator* as the Solution

- We have an abstraction *iterator* that iterates over an *aggregate* to do some action.
- We do not need to `know` about the *aggregate*, we only need to `use` the *iterator* to do something on the *aggregate*.
- In this design, the *aggregate* should create the *iterator* conceptually.
    - However, the *iterator* should own the *aggregate* for easy implementation.

The Solution (Design)



BookShelf Example



- We have a **Bookshelf** (Aggregation) that has many books.
- We need to *iterate* over the books in the **bookshelf**.

Step 1: The Players

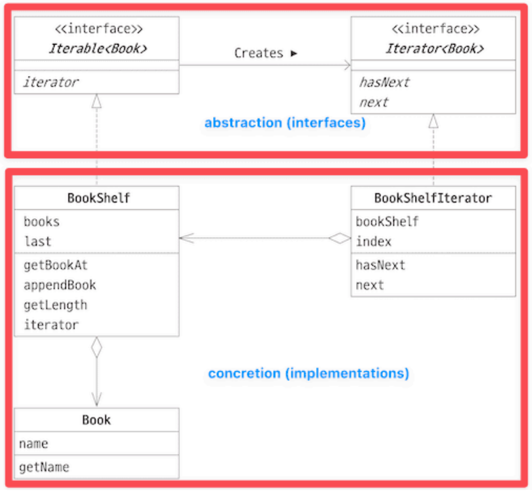
In this design, we have players.

- *Aggregate (Abstract Bookshelf)*
    - **ConcreteAggregate (Bookshelf)**
- *Iterator (Abstract)*
    - **ConcreteIterator**

Relationships Among Players

- It makes sense that the Aggregate should make the Iterator.
- It also makes sense that the Iterator should own the Aggregate for implementation of the algorithm.

Step 2: Abstraction and Concretion



- We need to see the `separation` (the dotted line) between *interfaces* and **implementations**.

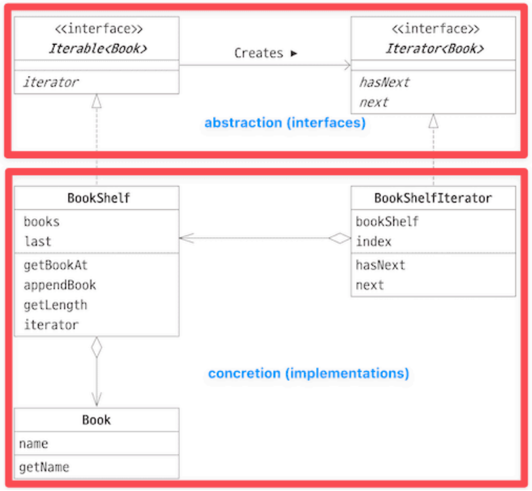
Abstractions

We have an *Iterator* that `iterates` over an **aggregate**.

- *Iterator (Abstract)*
- *Aggregate (Abstract Bookshelf)*

In short, we use the *iterator* to process targets in the *aggregate*.

- We call *aggregate iterable* too.



- Notice that we get the *iterator (I)* from the *aggregate (A)*.
  - It is as if we `hire` a **librarian** who `knows` well about the **library**.
  - The librarian (A) is from the library (A).

Concretions

- We have Bookshelf, and a special iterator of the BookShelf (BookShelfIterator) that `knows` about it.
  - ConcreteAggregate (Bookshelf)
  - ConcreteIterator
  - Target (Book)
- What is the easiest way to `implement` the relationship between the I and A?
  - It is an ownership relation (aggregation).

Code

- Main Method
- Book/Book_shelf
- BookIterator

Main Method

```python
from book import Book
from book_shelf import BookShelf

def main():
    print("=== Iterator Pattern Example ===\n")

    book_shelf = BookShelf(4)
    book_shelf.append_book(Book("Around the World in 80 Days"))
    ...

    print("Using explicit iterator:")
    it = book_shelf.iterator()
    while it.has_next():
        book = it.next()
        print(f"   {book.get_name()}")
    print()
```

Step 1: Create a bookshelf and add books

```
book_shelf = BookShelf(4)
book_shelf.append_book(Book("..."))
...
```

- In this example, we reserve the space for only four books, but the book_shelf can store as many as possible.

Step 2: Get an iterator from the bookshelf

```
it = book_shelf.iterator()
```

- The bookshelf returns an iterator through the `iterator()` method.

Step 3: Using the iterator, get all the books

```
while it.has_next():
    book = it.next()
```

- The iterator can access all the books using `has_next()` and `next()` method (interface).

Discussion

1. **What problem does this pattern solve?**

   It lets us access elements of a collection **sequentially** without exposing its internal structure.

2. **Do we need to know the details of the aggregation, such as the size or an index?**

   No. The iterator abstracts those details.

   - `hasNext()` → checks if more elements exist
   - `next()` → returns the next element

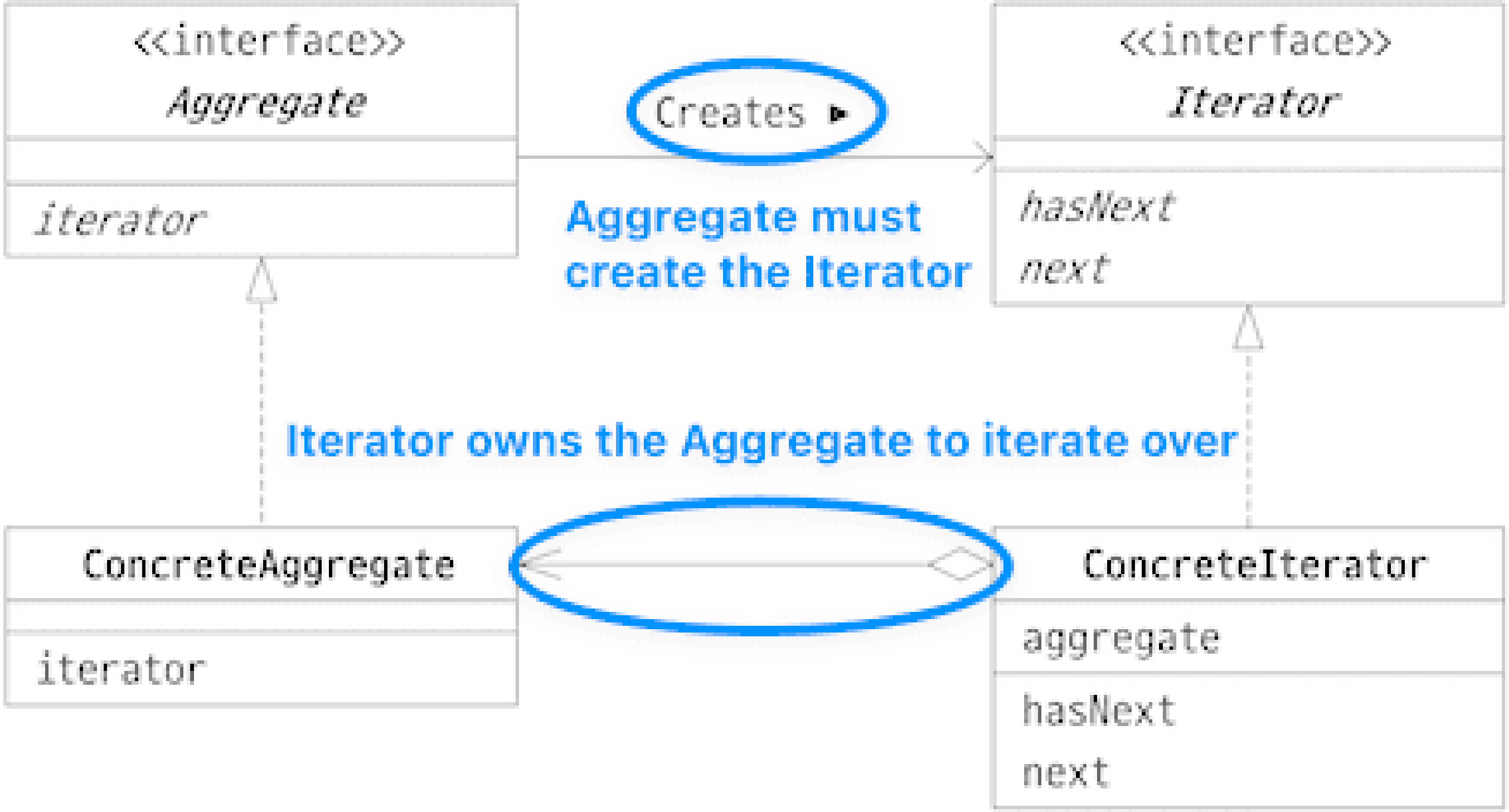
And specifics also.

3. **Which languages use this pattern?**
    - **Java**: Built-in `Iterator` and `Iterable` interfaces
    - **Python**: `for ... in ...` and `iter()` follow this pattern
    - **C#**: `foreach` relies on `IEnumerable` and `IEnumerator`

Example in Python:

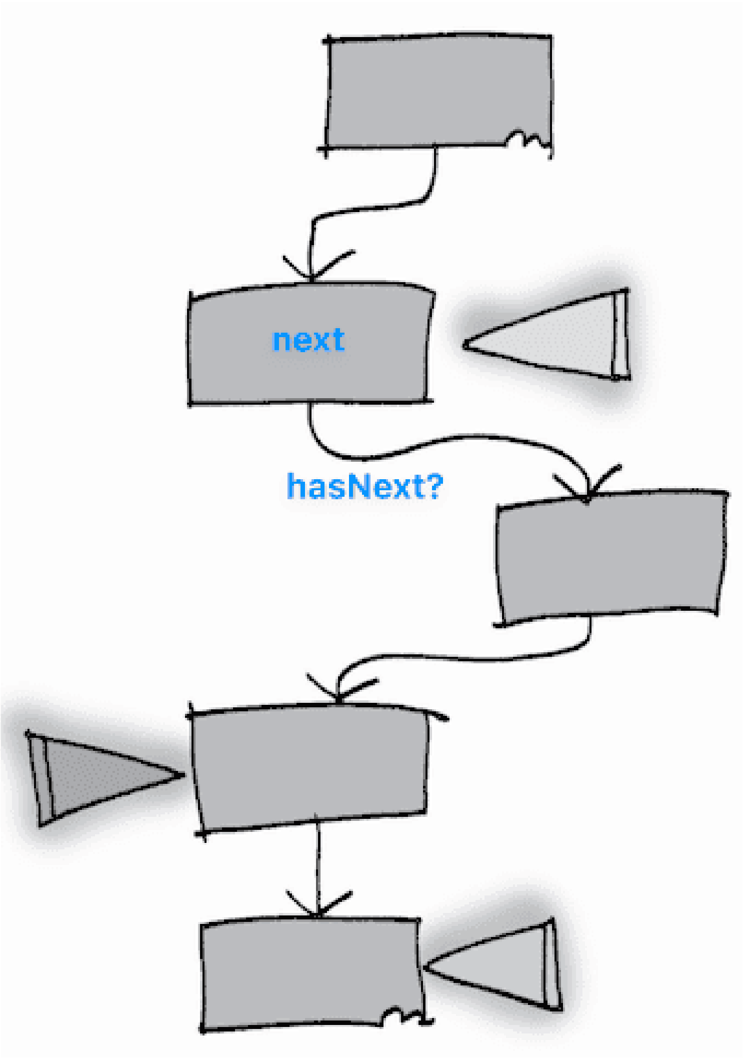
```
for i in it:   # syntactic sugar for iterator
    print(i)

# equivalent to:
while it.hasNext():
    i = it.next()
    print(i)
```

UML



Aggregate must create the Iterator

Iterator owns the Aggregate to iterate over


next
hasNext?