

Design Patterns Report

Name: Tạ Hữu Bình

Student ID: 20190094

I. Observer

a. Name – Alias: Observer (Publish-Subscribe) Pattern.

b. Classification: Behavioral patterns.

c. Intent:

Observer is a behavioral design pattern that defines a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.

d. Motivation:

Supposed that we have two types of objects: a Customer and a Store. The customer is very interested in a particular brand of product (say, it's a new model of the iPhone) which should become available in the store very soon.

The customer could visit the store every day and check product availability. But while the product is still en route, most of these trips would be pointless.

On the other hand, the store could send tons of emails (which might be considered spam) to all customers each time a new product becomes available. This would save some customers from endless trips to the store. At the same time, it'd upset other customers who aren't interested in new products.

It looks like there exists a conflict. Either the customer wastes time checking product availability or the store wastes resources notifying the wrong customers.

e. Solution

The object that has some interesting state is often called subject, but since it's also going to notify other objects about the changes to its state, we'll call it publisher. All other objects that want to track changes to the publisher's state are called subscribers.

The Observer pattern suggests that we add a subscription mechanism to the publisher class so individual objects can subscribe to or unsubscribe from a stream of events coming from that publisher. Fear not! Everything isn't as complicated as it sounds. In reality, this mechanism consists of:

1) An array field for storing a list of references to subscriber objects.

2) Several public methods which allow adding subscribers to and removing them from that list.



Figure 1: A subscription mechanism lets individual objects subscribe to event notifications.

Now, whenever an important event happens to the publisher, it goes over its subscribers and calls the specific notification method on their objects.

Real apps might have dozens of different subscriber classes that are interested in tracking events of the same publisher class. We wouldn't want to couple the publisher to all of those classes. Besides, we might not even know about some of them beforehand if our publisher class is supposed to be used by other people.

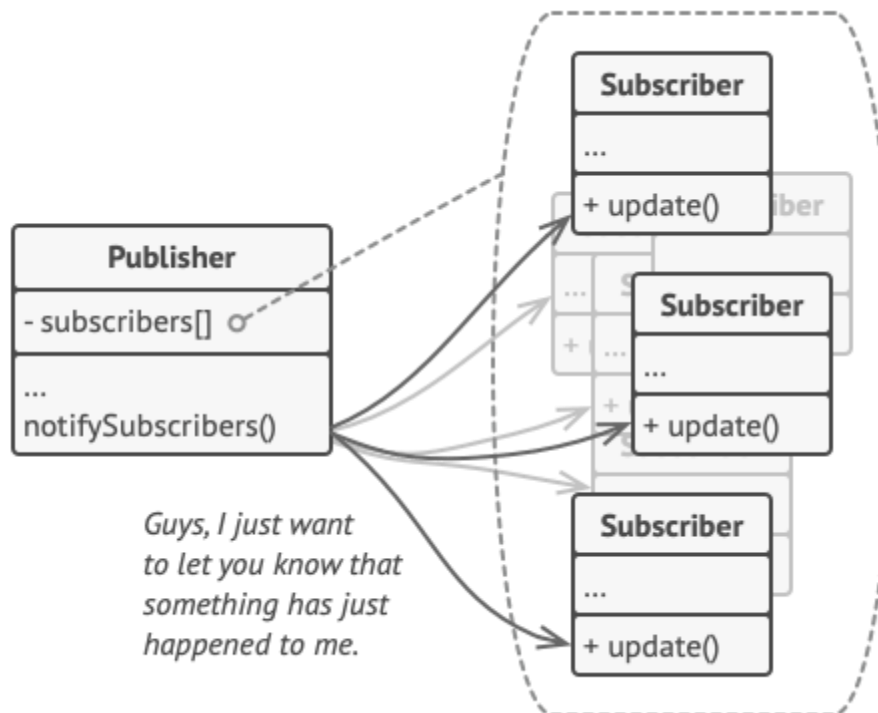


Figure 2: Publisher notifies subscribers by calling the specific notification method on their objects.

That's why it's crucial that all subscribers implement the same interface and that the publisher communicates with them only via that interface. This interface should

declare the notification method along with a set of parameters that the publisher can use to pass some contextual data along with the notification.

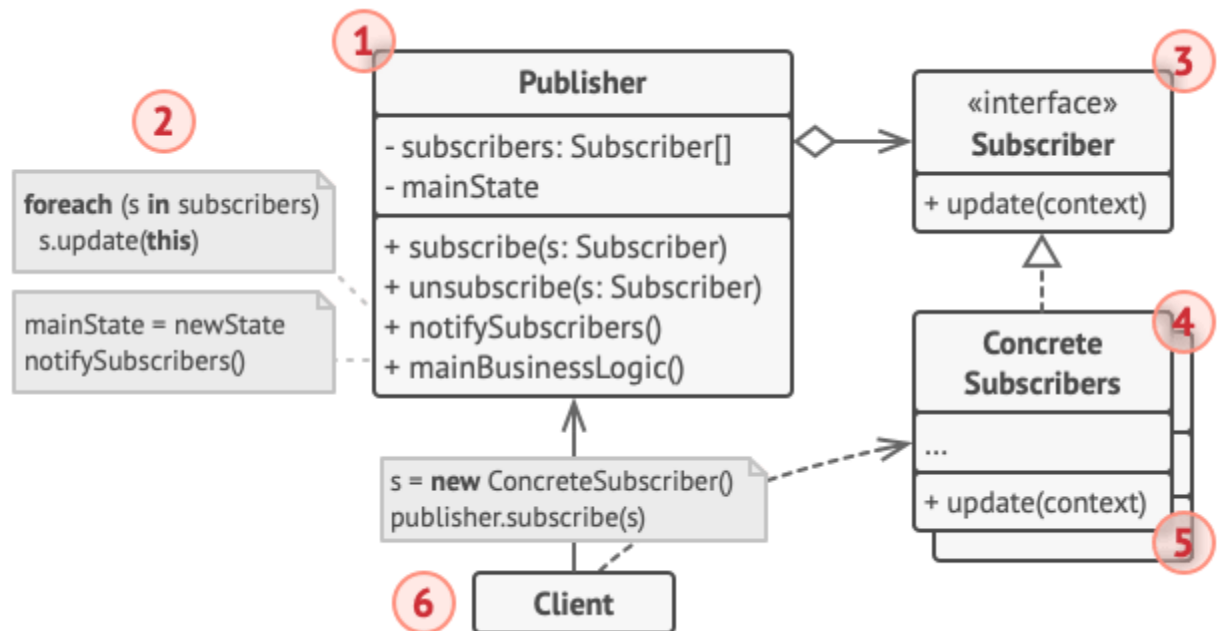


Figure 3: Structure of Observer Pattern.

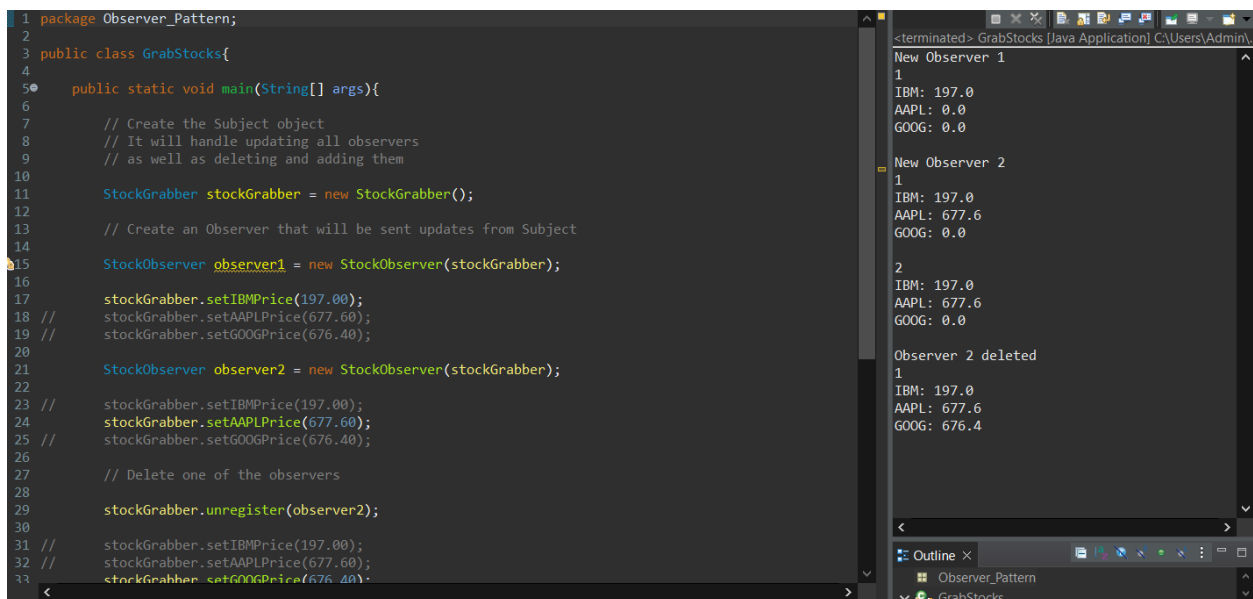
The structure of Observer Pattern includes:

1. The Publisher issues events of interest to other objects. These events occur when the publisher changes its state or executes some behaviors. Publishers contain a subscription infrastructure that lets new subscribers join and current subscribers leave the list.
2. When a new event happens, the publisher goes over the subscription list and calls the notification method declared in the subscriber interface on each subscriber object.
3. The Subscriber interface declares the notification interface. In most cases, it consists of a single *update* method. The method may have several parameters that let the publisher pass some event details along with the update.
4. Concrete Subscribers perform some actions in response to notifications issued by the publisher. All of these classes must implement the same interface so the publisher isn't coupled to concrete classes.
5. Usually, subscribers need some contextual information to handle the update correctly. For this reason, publishers often pass some context data as arguments of the notification method. The publisher can pass itself as an argument, letting subscriber fetch any required data directly.
6. The Client creates publisher and subscriber objects separately and then registers subscribers for publisher updates.

Illustrated source code of applying Observer pattern in a Stock Notification System is provided at:

https://github.com/Tahuubinh/TKXDPM.CNTN.20221-20190094.TaHuuBinh/tree/release/lab07/Design%20Pattern/src/Observer_Pattern

StockGrabber implements Subject to define methods making it become a publisher. StockObserver implements Observer to define methods making it become a subscriber. Anytime StockGrabber makes some changing, all objects of StockObserver get the information. Run "GrabStocks.java" to see the illustrated result as below:



The screenshot displays an IDE with two panels. The left panel shows the source code for the Observer pattern, and the right panel shows the execution output.

```
1 package Observer_Pattern;
2
3 public class GrabStocks{
4
5     public static void main(String[] args){
6
7         // Create the Subject object
8         // It will handle updating all observers
9         // as well as deleting and adding them
10
11         StockGrabber stockGrabber = new StockGrabber();
12
13         // Create an Observer that will be sent updates from Subject
14
15         StockObserver observer1 = new StockObserver(stockGrabber);
16
17         stockGrabber.setIBMPrice(197.00);
18         stockGrabber.setAAPLPrice(677.60);
19         stockGrabber.setGOOGPrice(676.40);
20
21         StockObserver observer2 = new StockObserver(stockGrabber);
22
23         stockGrabber.setIBMPrice(197.00);
24         stockGrabber.setAAPLPrice(677.60);
25         stockGrabber.setGOOGPrice(676.40);
26
27         // Delete one of the observers
28
29         stockGrabber.unregister(observer2);
30
31         stockGrabber.setIBMPrice(197.00);
32         stockGrabber.setAAPLPrice(677.60);
33         stockGrabber.setGOOGPrice(676.40);
34     }
35 }
```

The right panel shows the execution output for the application:

```
<terminated> GrabStocks [Java Application] C:\Users\Admin\...
New Observer 1
1
IBM: 197.0
AAPL: 0.0
GOOG: 0.0

New Observer 2
1
IBM: 197.0
AAPL: 677.6
GOOG: 0.0

2
IBM: 197.0
AAPL: 677.6
GOOG: 0.0

Observer 2 deleted
1
IBM: 197.0
AAPL: 677.6
GOOG: 676.4
```

f. Pros and cons

Pros:

- + Open/Closed Principle. We can introduce new subscriber classes without having to change the publisher's code (and vice versa if there's a publisher interface).
- + We can establish relations between objects at runtime.

Cons:

- Subscribers are notified in random order.

g. Applicability

- ✓ **Use the Observer pattern when changes to the state of one object may require changing other objects, and the actual set of objects is unknown beforehand or changes dynamically:** We can often experience this problem when working with classes of the graphical user interface. For example, we

created custom button classes, and we want to let the clients hook some custom code to our buttons so that it fires whenever a user presses a button. The Observer pattern lets any object that implements the subscriber interface subscribe for event notifications in publisher objects. We can add the subscription mechanism to our buttons, letting the clients hook up their custom code via custom subscriber classes.

- ✓ **Use the pattern when some objects in our app must observe others, but only for a limited time or in specific cases:** The subscription list is dynamic, so subscribers can join or leave the list whenever they need to.

II. Iterator

a. Name – Alias: Iterator (Cursor) Pattern.

b. Classification: Behavioral patterns.

c. Intent:

Iterator is a behavioral design pattern that helps traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.).

d. Motivation

Collections are one of the most used data types in programming. Nonetheless, a collection is just a container for a group of objects.



Figure 4: Various types of collections.

Most collections store their elements in simple lists. However, some of them are based on stacks, trees, graphs and other complex data structures.

But no matter how a collection is structured, it must provide some way of accessing its elements so that other code can use these elements. There should be a way to go through each element of the collection without accessing the same elements over and over.

This may sound like an easy job if we have a collection based on a list. We just loop over all of the elements. But how do we sequentially traverse elements of a complex data structure, such as a tree? For example, one day we might be just fine with depth-first traversal of a tree. Yet the next day we might require breadth-first traversal. And the next week, we might need something else, like random access to the tree elements.

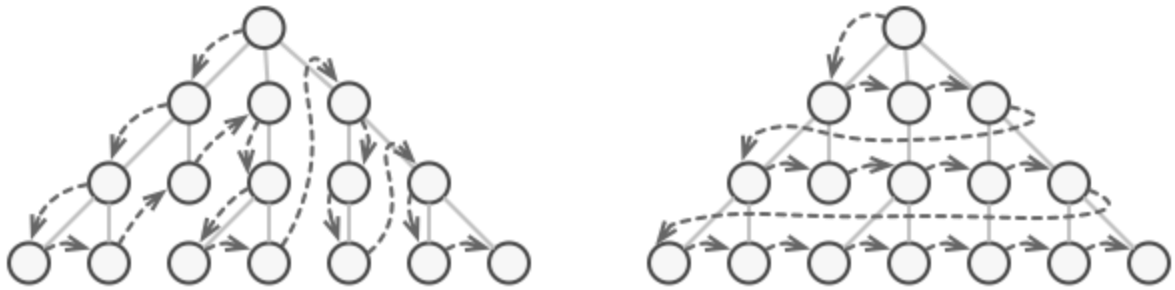


Figure 5: The same collection can be traversed in several different ways.

Adding more and more traversal algorithms to the collection gradually blurs its primary responsibility, which is efficient data storage. Additionally, some algorithms might be tailored for a specific application, so including them into a generic collection class would be weird.

On the other hand, the client code that's supposed to work with various collections may not even care how they store their elements. However, since collections all provide different ways of accessing their elements, we have no option other than to couple our code to the specific collection classes.

e. Solution

The main idea of the Iterator pattern is to extract the traversal behavior of a collection into a separate object called an iterator.

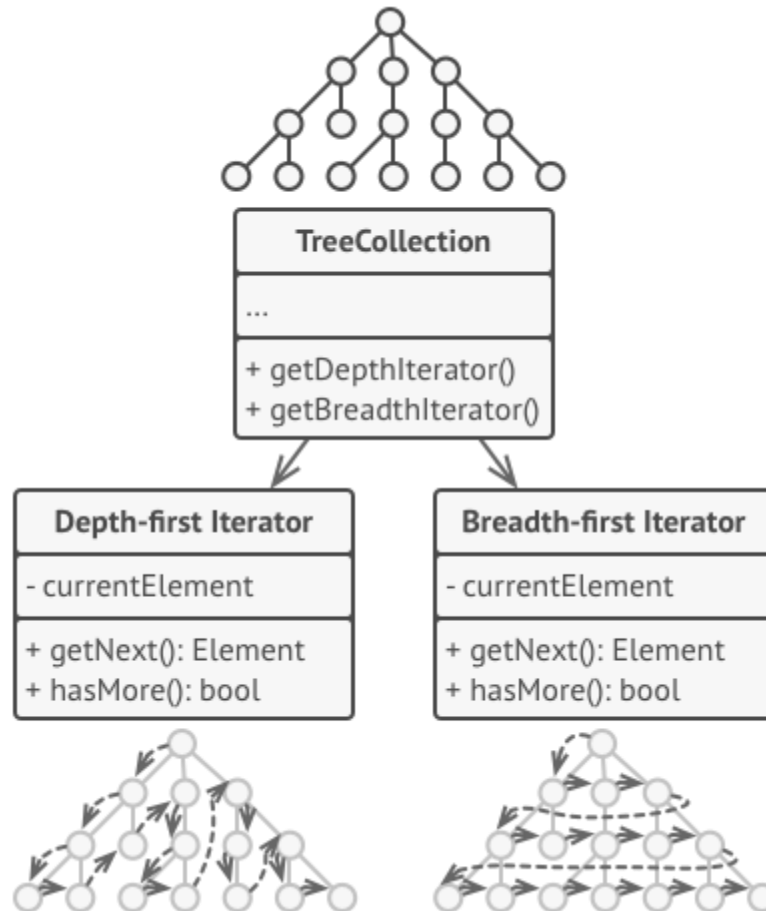


Figure 6: Iterators implement various traversal algorithms. Several iterator objects can traverse the same collection at the same time.

In addition to implementing the algorithm itself, an iterator object encapsulates all of the traversal details, such as the current position and how many elements are left till the end. Because of this, several iterators can go through the same collection at the same time, independently of each other.

Usually, iterators provide one primary method for fetching elements of the collection. The client can keep running this method until it doesn't return anything, which means that the iterator has traversed all of the elements.

All iterators must implement the same interface. This makes the client code compatible with any collection type or any traversal algorithm as long as there's a proper iterator. If we need a special way to traverse a collection, we just create a new iterator class, without having to change the collection or the client.

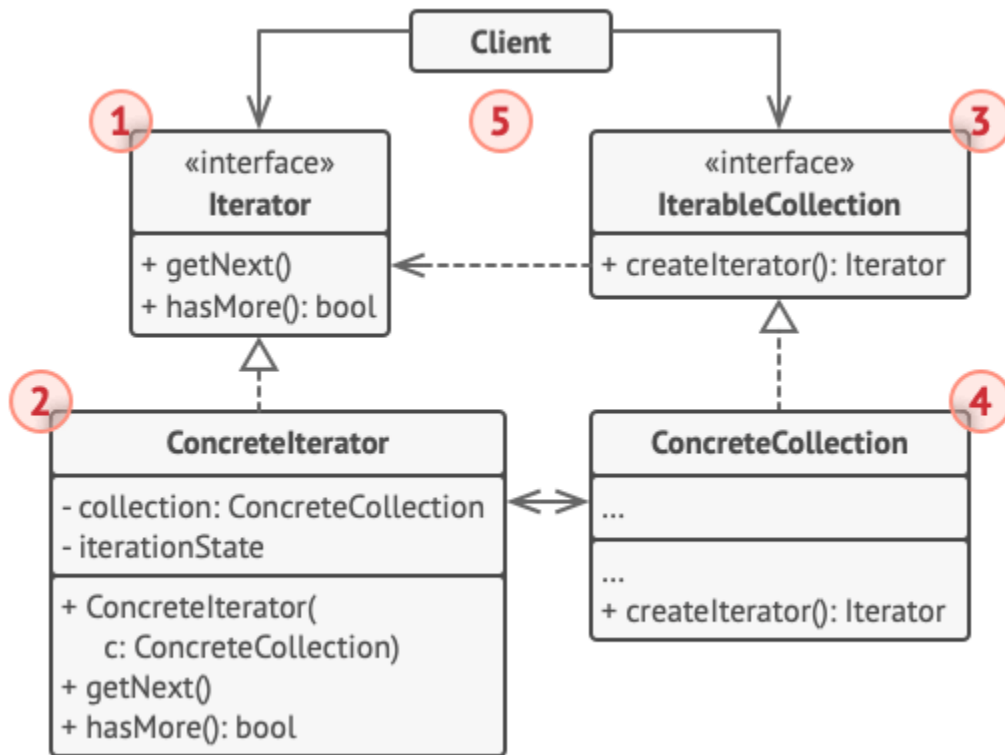


Figure 7: Structure of Iterator Pattern.

The structure of Observer Pattern includes:

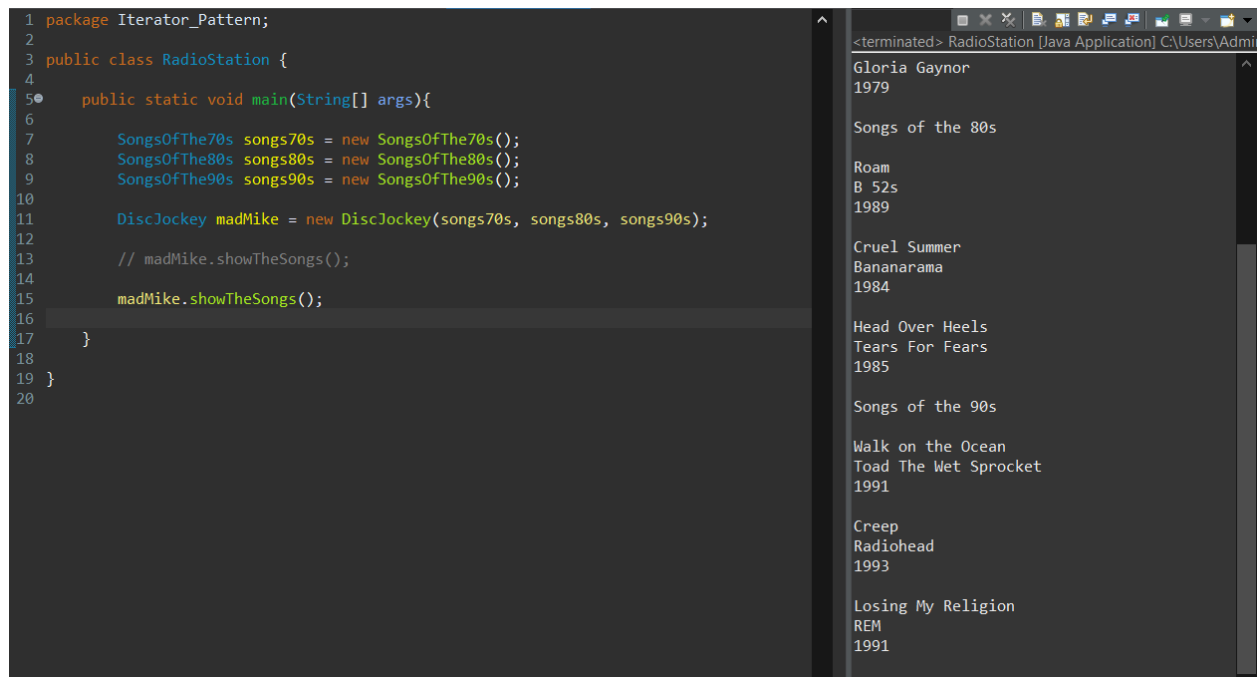
1. The Iterator interface declares the operations required for traversing a collection: fetching the next element, retrieving the current position, restarting iteration, etc.
2. Concrete Iterators implement specific algorithms for traversing a collection. The iterator object should track the traversal progress on its own. This allows several iterators to traverse the same collection independently of each other.
3. The Collection interface declares one or multiple methods for getting iterators compatible with the collection. Note that the return type of the methods must be declared as the iterator interface so that the concrete collections can return various kinds of iterators.
4. Concrete Collections return new instances of a particular concrete iterator class each time the client requests one. We might be wondering, where's the rest of the collection's code? Don't worry, it should be in the same class. It's just that these details aren't crucial to the actual pattern, so we're omitting them.
5. The Client works with both collections and iterators via their interfaces. This way the client isn't coupled to concrete classes, allowing us to use various collections and iterators with the same client code.

That's why it's crucial that all subscribers implement the same interface and that the publisher communicates with them only via that interface.

Illustrated source code of applying Iterator pattern in a Song Management System is provided at:

https://github.com/Tahuubinh/TKXDPM.CNTN.20221-20190094.TaHuuBinh/tree/release/lab07/Design%20Pattern/src/Iterator_Pattern

SongsOfThe70s, SongsOfThe80s, SongsOfThe90s are classes implements SongIterator to make the same kind of iterator though they utilized different data structures to stores information of related songs. All instances of songs belong to SongInfo class. The method 'showTheSongs' in DiscJockey class utilizes iterators to print information of each songs. Run "GrabStocks.java" to see the illustrated result as below:



The screenshot displays a Java IDE with two panels. The left panel shows the source code for the Iterator pattern implementation. The right panel shows the output of the application, which lists songs grouped by decade.

```
1 package Iterator_Pattern;
2
3 public class RadioStation {
4
5     public static void main(String[] args){
6
7         SongsOfThe70s songs70s = new SongsOfThe70s();
8         SongsOfThe80s songs80s = new SongsOfThe80s();
9         SongsOfThe90s songs90s = new SongsOfThe90s();
10
11         DiscJockey madMike = new DiscJockey(songs70s, songs80s, songs90s);
12
13         // madMike.showTheSongs();
14
15         madMike.showTheSongs();
16
17     }
18 }
19
20
```

The output window shows the following text:

```
<terminated> RadioStation [Java Application] C:\Users\Admi
Gloria Gaynor
1979

Songs of the 80s

Roam
B 52s
1989

Cruel Summer
Bananarama
1984

Head Over Heels
Tears For Fears
1985

Songs of the 90s

Walk on the Ocean
Toad The Wet Sprocket
1991

Creep
Radiohead
1993

Losing My Religion
REM
1991
```

f. Pros and cons

Pros:

- + Single Responsibility Principle. We can clean up the client code and the collections by extracting bulky traversal algorithms into separate classes.
- + Open/Closed Principle. We can implement new types of collections and iterators and pass them to existing code without breaking anything.
- + We can iterate over the same collection in parallel because each iterator object contains its own iteration state.
- + For the same reason, we can delay an iteration and continue it when needed.

Cons:

- Applying the pattern can be an overkill if our app only works with simple collections.
- Using an iterator may be less efficient than going through elements of some specialized collections directly.

g. Applicability

- ✓ **Use the Iterator pattern when our collection has a complex data structure under the hood, but we want to hide its complexity from clients (either for convenience or security reasons):** The iterator encapsulates the details of working with a complex data structure, providing the client with several simple methods of accessing the collection elements. While this approach is very convenient for the client, it also protects the collection from careless or malicious actions which the client would be able to perform if working with the collection directly.
- ✓ **Use the pattern to reduce duplication of the traversal code across our app:** The code of non-trivial iteration algorithms tends to be very bulky. When placed within the business logic of an app, it may blur the responsibility of the original code and make it less maintainable. Moving the traversal code to designated iterators can help us make the code of the application more lean and clean.
- ✓ **Use the Iterator when we want our code to be able to traverse different data structures or when types of these structures are unknown beforehand:** The pattern provides a couple of generic interfaces for both collections and iterators. Given that our code now uses these interfaces, it'll still work if we pass it various kinds of collections and iterators that implement these interfaces.