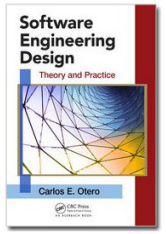
CHAPTER 7: STRUCTURAL AND BEHAVIORAL DESIGN PATTERNS

SESSION II: BEHAVIORAL DESIGN PATTERNS, ITERATOR, OBSERVER

Software Engineering Design: Theory and Practice
by Carlos E. Otero



SESSION'S AGENDA

- Behavioral Patterns in Detailed Design
- Iterator
  - ✓ Example…
- Observer…
  - ✓ Example…
- What's next…

BEHAVIORAL DESIGN PATTERNS

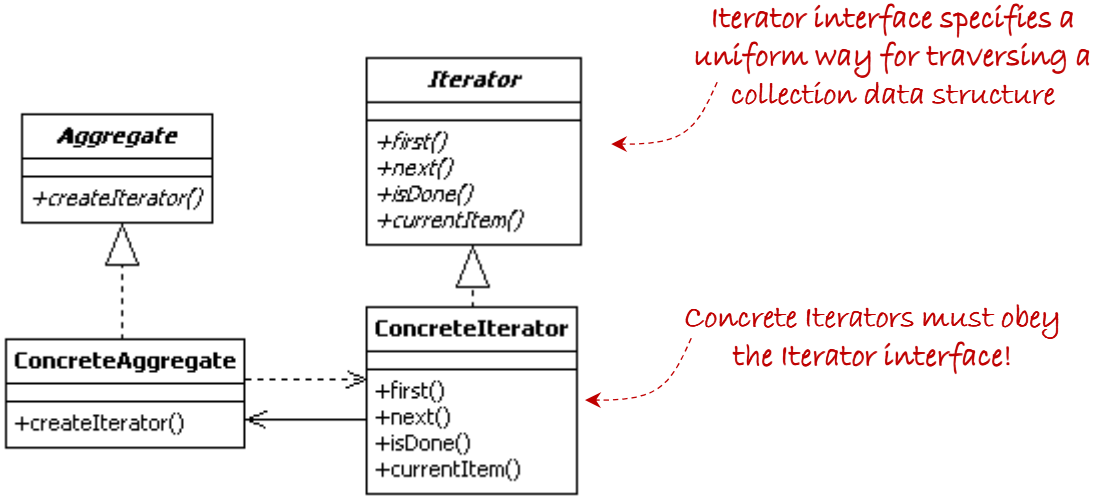
- Behavioral design patterns are patterns that deal with encapsulating behavior with objects, assigning responsibility, and managing object cooperation when achieving common tasks [1].
  - ✓ They include many of the mainstream design patterns used in modern object-oriented frameworks and play a key role in the design of systems by making them independent of specific behavior, which is made replaceable with objects throughout these design patterns.

- Popular behavioral design patterns include:
  - ✓ Iterator
  - ✓ Observer
  - ✓ Others not covered:
    - Command
    - Chain of responsibility
    - Interpreter
    - Mediator
    - Memento
    - State
    - Strategy
    - Template Method
    - Visitor

ITERATOR DESIGN PATTERN

- The Iterator design pattern is an object behavioral pattern that provides a standardized way for accessing and traversing objects in a collection data structure.

- According to the GoF, the intent of the Iterators to [1],
  - ✓ Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.



ITERATOR DESIGN PATTERN EXAMPLE

Problem

A company's software system manages inventory, financials, and all other information available from its two store branches. Each store carries specific computer products appropriate for their location's demographics. During design, the software system is decomposed into several components, which includes two components for deferring and abstracting design information relevant to requirements for each computer store branch. The detailed design of each component is carried out separately by two different software engineers; this results in two different versions of data structures for managing and providing store product information. Now, anytime the software system is called upon to display information about store products, it is required to identify between the two store branches so that the correct implementation for accessing store information can be executed. This problem is encountered every time a new computer store branch is added to the system; therefore, a uniform and standardized method for accessing computer store products from different collection data structure is highly desirable.

Iterator Design Pattern Example

```
class ComputerProduct
{
public:
    ComputerProduct(void);
    virtual ~ComputerProduct(void);

    // Return the product's id.
    int getProductId(void);

    // Return the product's price.
    int getPrice(void) const;

    // Return the product's description.
    string getDescription(void) const;

    // Other methods here...
private:
    int _id;
    int _price;
    string _description;
};
```

Consider this case, where code in two different computer store classes use different collection data structures, each with its own unique interface!

```
class SimpleComputerStore
{
public:
    SimpleComputerStore(void);
    virtual ~SimpleComputerStore(void);

    // Return a pointer to the simple product list.
    SimpleProductList* getProducts(void) {

        // Return the simple product list.
        return &_products;
    }
private:
    // The list of simple computer products.
    SimpleProductList _products;
};
```

```
class AdvancedComputerStore
{
public:
    AdvancedComputerStore(void);
    virtual ~AdvancedComputerStore(void);

    // Computer store methods...

    // Return a pointer to the advanced product list.
    AdvancedProductList* getProducts(void) {

        // Return the advanced product list.
        return &_products;
    }
private:
    // The computer products... in ProductList form.
    AdvancedProductList _products;
};
```

These could've been other collection data structures, arrays, etc.

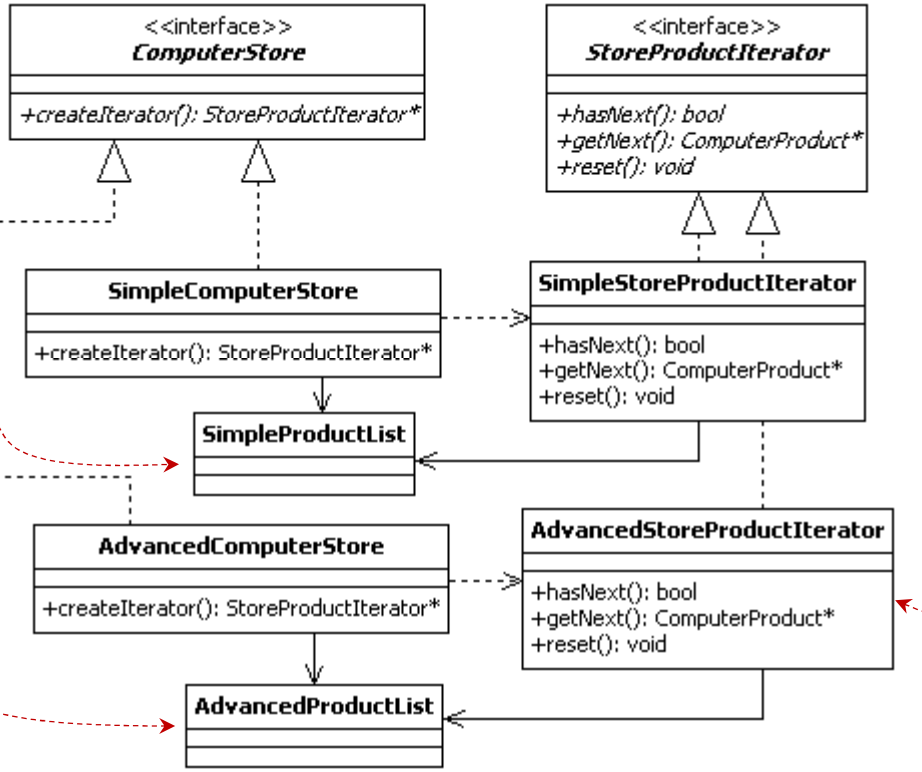
Iterator Design Pattern Example

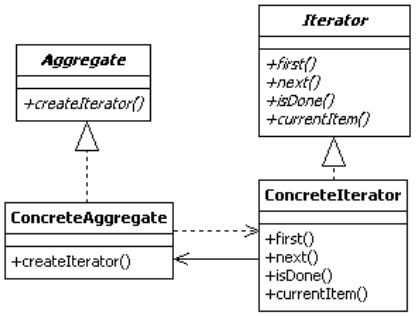
```
// Simple store.
SimpleComputerStore simpleStore;
ComputerProduct* pProduct = 0;
SimpleProductList* simpleStoreProducts = simpleStore.getProducts();

// Display simple store products.
for( int i = 0; i < simpleStoreProducts->size(); i++ ) {

  // Retrieve the product at index i.
  pProduct = simpleStoreProducts->getSimpleProduct(i);

  // Make sure pProduct is valid before using it!

  // Display product's information.
  cout<<"Product id: "<<pProduct->getProductId()<<endl
      <<"Product price: "<<pProduct->getPrice()<<endl
      <<"Product Description: "<<pProduct->getDescription().c_str()<<endl;
}

//Advanced store.
AdvancedComputerStore advancedStore;
AdvancedProductList* advancedProducts = advancedStore.getProducts();

// Display advanced store products.
for( int i = 0; i < advancedProducts->length(); i++ ) {

  // Retrieve the product at location i.
  pProduct = advancedProducts->getAdvancedProduct(i);

  // Make sure pProduct is valid before using it!

  // Display product's information.
  cout<<"Product id: "<<pProduct->getProductId()<<endl
      <<"Product price: "<<pProduct->getPrice()<<endl
      <<"Product Description: "<<pProduct->getDescription().c_str()<<endl;
}
// Repeat here for all other types of lists!
```

If a new store is added, we have to modify this code to add code for displaying products of the new store!

Different interfaces require different code!

Iterator Design Pattern Example





Notice how in this example we separate the collection data structures from the computer stores!

Notice the pattern! Applying the pattern to the computer store problem entails doing more of the same!

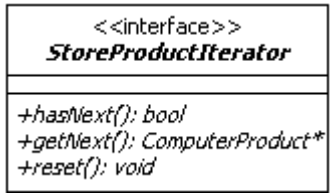
Concrete Iterators must obey the StoreProductIterator interface to provide a uniform way for traversing a collection data structure containing Computer Products

Let's see how this is implemented…

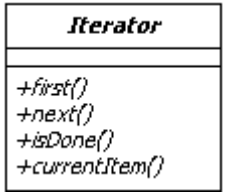
Iterator Design Pattern Example

```
// The base for all store product iterators.
class StoreProductIterator {

public:
    // Constructor.
    StoreProductIterator() { /*Intentionally left blank.*/}

    // Interface method for determining if more products are available.
    virtual bool hasNext(void) = 0;

    // Interface method for retrieving the next available product.
    virtual ComputerProduct* getNext(void) = 0;

    // Reset the iterator's position.
    virtual void reset(void) = 0;
};
```

Common interface for iterating through the products!



Notice that we're using different interface names than the original pattern!



Iterator Design Pattern Example

```
// The base for all store product iterators.
class StoreProductIterator {

public:
    // Constructor.
    StoreProductIterator() : _position(0) { /*Intentionally left blank.*/}

    // Interface method for determining if more products are available.
    virtual bool hasNext() = 0;

    // Interface method for retrieving the next available product.
    virtual ComputerProduct* getNext() = 0;

    // Reset the iterator's position.
    virtual void reset(void) { _position = 0; }

protected:
    // Give access to derived classes for setting the iterator's position.
    void setPosition(int position) { _position = position; }

    // Allow derived classes to retrieve the iterator's position.
    int getPosition(void) { return _position; }

private:
    // The iterator's current position.
    int _position;
};
```

For this problem, we've added some helper methods. It is OK to modify the pattern based on domain knowledge to fit the design to the application!

Iterator Design Pattern Example

The Iterator used for the simple store

```
class SimpleStoreProductIterator : public StoreProductIterator {

public:
    // Constructor.
    SimpleStoreProductIterator(SimpleProductList* products);

    // Determine if more products are available.
    bool hasNext(void);

    // If more products are available, get the next one.
    ComputerProduct* getNext(void);

private:
    // Pointer to the simple computer product list.
    SimpleProductList* _products;
};
```

Requires a simple product data collection!

Iterator Design Pattern Example

```
// Constructor.
SimpleStoreProductIterator::SimpleStoreProductIterator(SimpleProductList* products) {
    // For simplicity, assume a valid pointer.
    _products = products;
}

// Determine if more products are available.
bool SimpleStoreProductIterator::hasNext() {
  // The return value.
  bool isNextProductAvailable = false;

  if( getPosition() < _products->size() ) {

    isNextProductAvailable = true;
  }
  return isNextProductAvailable;
}

// If more products are available, get the next one.
ComputerProduct* SimpleStoreProductIterator::getNext() {
  // Temporary pointer to computer product.
  ComputerProduct* pProduct = 0;

  // Get the iterator's current position.
  int nextItem = getPosition();

  // Determine if there are more products.
  if( hasNext() ) {
    // Get the address of the next product and move the iterator's
    // position.
    pProduct = _products->getSimpleProduct(nextItem++);

    // Set the new position of the Iterator.
    setPosition(nextItem);
  }
  // Return the requested product.
  return pProduct;
}
```

Common interface allow clients to traverse through the _products objects in this collection, regardless of the difference that may exist in the _products interfaces!

Details / differences are hidden from clients !

Iterator Design Pattern Example

```
// Constructor.
AdvancedStoreProductIterator::AdvancedStoreProductIterator(AdvancedProductList* products) {
  // For simplicity, assume valid pointer.
  _products = products;
}

// Determine if more products are available.
bool AdvancedStoreProductIterator::hasNext() {
  // The return value.
  bool nextProductAvailable = false;

  if( getPosition() < _products->length() ) {
    nextProductAvailable = true;
  }
  return nextProductAvailable;
}

// If more products are available, get the next one.
ComputerProduct* AdvancedStoreProductIterator::getNext() {
  // Temporary pointer to computer product.
  ComputerProduct* pProduct = 0;

  // Get the iterator's current position.
  int nextItem = getPosition();

  // Determine if there are more products.
  if( hasNext() ) {
    // Get the address of the next product and move the iterator's
    // position.
    pProduct = _products->getAdvancedProduct(nextItem++);

    // Set the new position of the iterator.
    setPosition(nextItem);
  }
  // Return the requested product.
  return pProduct;
}
```

Common interface allow clients to traverse through the _products objects in this collection, regardless of the difference that may exist in the _products interfaces!

Details / differences are hidden from clients !

Iterator Design Pattern Example

```
// The interface for all computer stores.
class ComputerStore {

public:
  // The interface method to create an iterator.
  virtual StoreProductIterator* createIterator() = 0;
};
```

```
// Simple Computer Store
class SimpleComputerStore : public ComputerStore {

public:
  // Override the createIterator interface method to create the
  // appropriate iterator for simple computer stores.
  StoreProductIterator* createIterator() {

    // Create and return a simple store product iterator.
    return new SimpleStoreProductIterator(&_products);
  }

  // All other methods for simple computer stores.

private:
    // The simple product list.
    SimpleProductList _products;
};
```

```
// Advanced Computer Store
class AdvancedComputerStore : public ComputerStore {

public:
  // Override the createIterator interface method to create the
  // appropriate iterator for advanced computer stores.
  StoreProductIterator* createIterator() {

    // Create and return an advanced store product iterator.
    return new AdvancedStoreProductIterator(&_products);
  }

  // All other methods for advanced computer stores.

private:
  // The advanced product list.
  AdvancedProductList _products;
};
```

ITERATOR DESIGN PATTERN EXAMPLE

```
// Display the products using the iterator.
void displayProducts(StoreProductIterator* pIterator) {

  // Temporary pointer to hold a computer product.
  ComputerProduct* pProduct = 0;

  // Determine if there are more products to browse.
  while( pIterator->hasNext() ) {

    // Retrieve the next product.
    pProduct = pIterator->getNext();

    // Display the product's information.
    cout<<"\nProduct id: "<<pProduct->getProductId()<<endl
        <<"Product price: "<<pProduct->getPrice()<<endl
        <<"Product Description: "<<pProduct->getDescription().c_str();
  }
}
```

We can now display product information the same way for any store! With this design, if new stores are added, we can reuse the same displayProduct method to display products!

Iterators are used all over the Java framework, C#, and C++ standard template library!

The Java Iterator and ListIterator Interface

Java Iterator

Method Summary

boolean	**hasNext**() Returns `true` if the iteration has more elements.
Object	**next**() Returns the next element in the iteration.
void	**remove**() Removes from the underlying collection the last element returned by the iterator (optional operation).

Java ListIterator

Method Summary

void	**add**(Object o) Inserts the specified element into the list (optional operation).
boolean	**hasNext**() Returns `true` if this list iterator has more elements when traversing the list in the forward direction.
boolean	**hasPrevious**() Returns `true` if this list iterator has more elements when traversing the list in the reverse direction.
Object	**next**() Returns the next element in the list.
int	**nextIndex**() Returns the index of the element that would be returned by a subsequent call to `next`.
Object	**previous**() Returns the previous element in the list.
int	**previousIndex**() Returns the index of the element that would be returned by a subsequent call to `previous`.
void	**remove**() Removes from the list the last element that was returned by `next` or `previous` (optional operation).
void	**set**(Object o) Replaces the last element returned by `next` or `previous` with the specified element (optional operation).

JAVA ENUMERATION INTERFACE

java.util

Interface Enumeration

All Known Subinterfaces:

NamingEnumeration

All Known Implementing Classes:

StringTokenizer

---

public interface **Enumeration**

An object that implements the Enumeration interface generates a series of elements, one at a time. Successive calls to the `nextElement` method return successive elements of the series.

For example, to print all elements of a vector *v*:

```
for (Enumeration e = v.elements() ; e.hasMoreElements() ;) {
    System.out.println(e.nextElement());

}
```

Methods are provided to enumerate through the elements of a vector, the keys of a hashtable, and the values in a hashtable. Enumerations are alsc used to specify the input streams to a `SequenceInputStream`.

NOTE: The functionality of this interface is duplicated by the Iterator interface. In addition, Iterator adds an optional remove operation, and has shorter method names. New implementations should consider using Iterator in preference to Enumeration.

ITERATOR DESIGN PATTERN

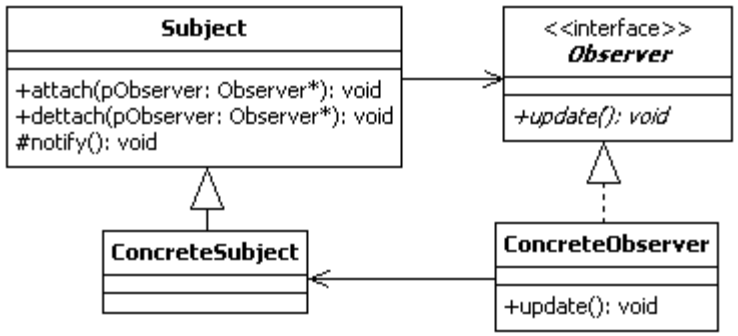
- The step-by-step approach for designing the Iterator design pattern is presented as follows:
  1. Identify and design the Iterator interface.
  2. For each class representing a collection data structure in the software system, design a concrete Iterator and associate it with it. Implement the concrete iterator's methods in terms of the collection data structure.
  3. Create the Aggregate interface, which includes the interface method to create Iterators.
  4. For each class representing a collection data structure, implement the Aggregate interface to instantiate and return a concrete Iterator.

- Benefits of the Iterator design pattern include:
  - ✓ The Iterator provides a consistent way for clients to iterate through the objects in a collection.
  - ✓ It abstracts the internals of the collection objects so that if they change, clients do not have to change.
  - ✓ Allows client code to be extended easily; numerous iterators can be created to support different traversals from the same or different collection structure.

OBSERVER DESIGN PATTERN

- The Observer design pattern is an object behavioral pattern that standardizes the operations between objects that interoperate using a one-to-many relationship.

- According to the GoF, the intent of the Observer is to [1]
  - ✓ Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

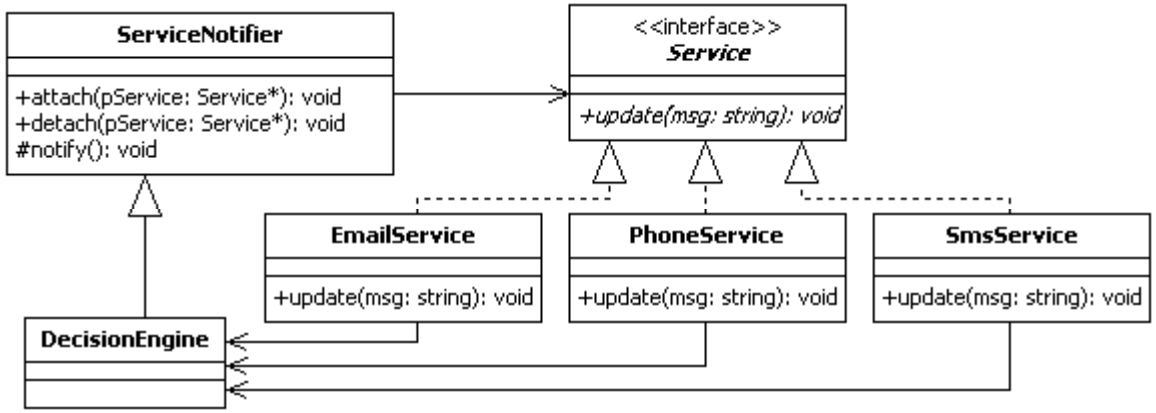


OBSERVER DESIGN PATTERN EXAMPLE

Problem

A local university is designing a system for weather-alert notification that allows students, faculty, and staff to receive notifications of class cancelations (due to weather) via email, voice call, or SMS text messages. Other methods of notification may be added in the future. The system is based on the weather data decision engine that interfaces with several weather related data sources, fuses the information, and automatically decides whether class cancellations are in effect. The university is interested in integrating the existing communication services (i.e., email, sms, and voice) with the decision engine so that these services can be triggered to initiate notification via their respective communication types. The design must be flexible so that other types of communication mechanisms can be added to the system in the future.

OBSERVER DESIGN PATTERN EXAMPLE



Observer Design Pattern Example

```
// Provide the registration mechanism for all observers.
void ServiceNotifier::attach(Service* pService) {

  // Add this observer to the list of registered observers.  Assume a
  // valid pointer.
  _services.push_back(pService);
};

// The trigger mechanism to notify all observers of class cancellation.
void ServiceNotifier::notify(string message) {

  // Get an Iterator that points to the beginning of the observers_ list.
  list<Service*>::iterator pIter = _services.begin();

  // Iterate through the list of observers and notify them.
  for( int i = 0; i < _services.size(); i++ ) {

    // Pass the message along to all registered observers.
    (*pIter++)->update(message);
  }
}

class EmailService : public Service {

public:
  // Once the Observable object changes, it will call this method.
  void update(string message) {

    // Open file containing all users registered for email notification.
    // Open connection to the Email server.
    // For all registered clients, notify them via email.
  }

  // ...
};
```

Observer Design Pattern Example

```
// Sends message as email.
EmailHandler emailHandler;

// Sends message as text message.
SmsHandler smsHandler;

// Translates message to speech and sends it via the voice interface.
VoiceHandler voiceHandler;

// Assume that the decision engine object is a singleton.
DecisionEngine::getInstance()->register(&emailHandler);
DecisionEngine::getInstance()->register(&smsHandler);
DecisionEngine::getInstance()->register(&voiceHandler);
```

Register observers with the subject, in this case, the DecisionEngine

When a decision is made, the notify method is called to iterate through the list of registered observers to update them with the latest news!

```
void DecisionEngine::notify()
{
    // Iterate through the list of observers.
    list<Service*>::iterator pIter = _registeredServices.begin();

    for( int i = 0; i < _registeredServices.size(); i++ )
    {
        // Assume that a decision has been made and a string
        // _message created to notiy observers of the news.
        (*pIter++)->update(_message);
    }
}
```

JAVA OBSERVER INTERFACE

java.util

Interface Observer

---

public interface **Observer**

A class can implement the `Observer` interface when it wants to be informed of changes in observable objects.

Since:

JDK1.0

See Also:

`Observable`

Method Summary	
void	**update**(Observable o, Object arg) This method is called whenever the observed object is changed.

JAVA OBSERVABLE

java.lang.Object
  └ java.util.Observable

---

public class **Observable**
extends Object

Observers are almost always used within the MVC architecture!

This class represents an observable object, or "data" in the model-view paradigm. It can be subclassed to represent an object that the application wants to have observed.

An observable object can have one or more observers. An observer may be any object that implements interface `Observer`. After an observable instance changes, an application calling the `Observable`'s `notifyObservers` method causes all of its observers to be notified of the change by a call to their `update` method.

The order in which notifications will be delivered is unspecified. The default implementation provided in the Observerable class will notify Observers in the order in which they registered interest, but subclasses may change this order, use no guaranteed order, deliver notifications on separate threads, or may guarantee that their subclass follows this order, as they choose.

Note that this notification mechanism is has nothing to do with threads and is completely separate from the `wait` and `notify` mechanism of class `Object`.

When an observable object is newly created, its set of observers is empty. Two observers are considered the same if and only if the `equals` method returns true for them.

JAVA OBSERVABLE METHODS

Method Summary	
void	addObserver(Observer o) Adds an observer to the set of observers for this object, provided that it is not the same as some observer already in the set.
protected void	clearChanged() Indicates that this object has no longer changed, or that it has already notified all of its observers of its most recent change, so that the hasChanged method will now return false.
int	countObservers() Returns the number of observers of this Observable object.
void	deleteObserver(Observer o) Deletes an observer from the set of observers of this object.
void	deleteObservers() Clears the observer list so that this object no longer has any observers.
boolean	hasChanged() Tests if this object has changed.
void	notifyObservers() If this object has changed, as indicated by the hasChanged method, then notify all of its observers and then call the clearChanged method to indicate that this object has no longer changed.
void	notifyObservers(Object arg) If this object has changed, as indicated by the hasChanged method, then notify all of its observers and then call the clearChanged method to indicate that this object has no longer changed.
protected void	setChanged() Marks this Observable object as having been changed; the hasChanged method will now return true.

OBSERVER DESIGN PATTERN

- The steps required to apply the Observer design pattern include:
  1. Design the Subject interface and implement code for attaching, detaching, and notifying observer objects. The code for keeping track of observers can be done using existing linked-lists data structures.
  2. For classes that manage information of interests to observers, inherit from the subject class created in step 1.
  3. Design the Observer interface, which includes the abstract update interface method.
  4. For all observers in the system, implement the Observer interface, which requires implementing the update method.
  5. At run-time, create each observer and attach them to the subject. When changes occur, the subject iterates through its list of registered objects, and calls their update method.

- Benefits of the Observer design pattern:
  - ✓ Flexibility for adding new services to the system.
  - ✓ Since specific services are compartmentalized, maintain and modifying existing system services becomes easier.

WHAT'S NEXT…

- In this session, we presented behavioral design patterns, including:
  - ✓ Iterator
  - ✓ Observer

- This concludes the presentation of design patterns in detailed design. In the next module, we will present a different form of design which occurs (mostly) at the function-level. We refer to this form of design as construction design.

REFERENCES

- [1] Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston: Addison-Wesley, 1995.