

Design Patterns

Object Oriented Programming (PCC-CS503)
@UD

Sep 2023

Content:

- Inheritance in OO design
- Design patterns
 - . Introduction and classification
 - . Iterator pattern

Inheritance in OO Design

- Advantage: Code reuse
- Readability Issue: Difficult to understand code at deeper level of inheritance with many levels of inheritance ← All ancestors to be checked
- Flexibility and Complexity Issue: All subclasses are tightly coupled with superclass:
 - Need to inherit all methods and attributes from parent class, even if not needed
 - Is-a relationship doesn't always work, e.g. circle-ellipse
- Choose inheritance for modelling “is-a” relationship, and composition “has-a”. Design preference: Composition over inheritance in the absence of clear hierarchy

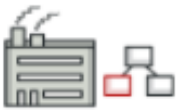


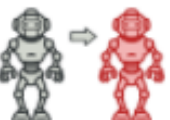

Introduction: Design Pattern

- Best practices for solving common design problems within certain constraints
 - Used by experienced OOP developers, e.g. “Let’s use *that* pattern for solving this problem”
 - Assume a class needed with a single instance, and that single object used by all other classes. Solution: ???
- An idea, but not any particular implementation
- Needed for making the code flexible, reusable maintainable
- Formally written by **Gang of Four** in ‘1994: Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides

Classification

Creational patterns

These patterns provide various object creation mechanisms, which increase flexibility and reuse of existing code.

	
Factory Method	Abstract Factory
	
Builder	Prototype
	
Singleton	





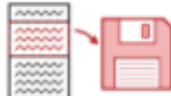

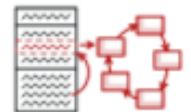



Structural patterns

These patterns explain how to assemble objects and classes into larger structures while keeping these structures flexible and efficient.

	
Adapter	Bridge
	
Composite	Decorator
	
Facade	Flyweight

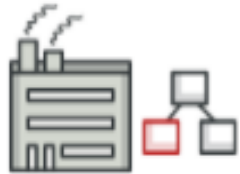
Behavioral patterns

These patterns are concerned with algorithms and the assignment of responsibilities between objects.

			
Chain of Responsibility	Command	Iterator	Mediator
			
Memento	Observer	State	Strategy
			
Template Method	Visitor		

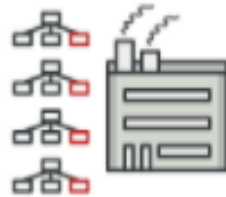
Creational Design Patterns

Creational patterns provide various object creation mechanisms, which increase flexibility and reuse of existing code.



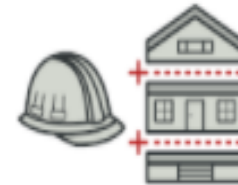
Factory Method

Provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.



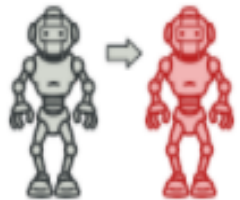
Abstract Factory

Lets you produce families of related objects without specifying their concrete classes.



Builder

Lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.



Prototype

Lets you copy existing objects without making your code dependent on their classes.



Singleton

Lets you ensure that a class has only one instance, while providing a global access point to this instance.

Creational Design Pattern: Singleton Example

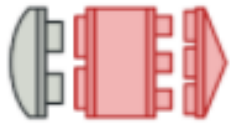
```
// Thread Synchronized Java implementation of
// singleton design pattern
class Singleton
{
    private static Singleton obj;

    private Singleton() {}

    // Only one thread can execute this at a time
    public static synchronized Singleton getInstance()
    {
        if (obj==null)
            obj = new Singleton();
        return obj;
    }
}
```

Structural Design Patterns

Structural patterns explain how to assemble objects and classes into larger structures while keeping these structures flexible and efficient.



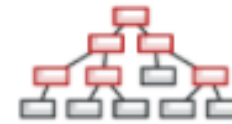
Adapter

Allows objects with incompatible interfaces to collaborate.



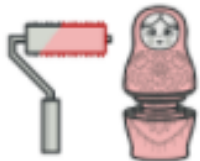
Bridge

Lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.



Composite

Lets you compose objects into tree structures and then work with these structures as if they were individual objects.



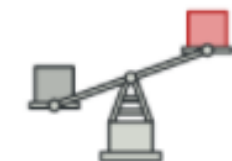
Decorator

Lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.



Facade

Provides a simplified interface to a library, a framework, or any other complex set of classes.



Flyweight

Lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all of the data in each object.



Proxy

Lets you provide a substitute or placeholder for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.

Behavioral Design Patterns

Behavioral design patterns are concerned with algorithms and the assignment of responsibilities between objects.



Chain of Responsibility

Lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.



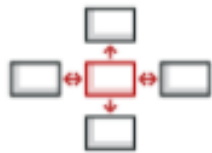
Command

Turns a request into a stand-alone object that contains all information about the request. This transformation lets you parameterize methods with different requests, delay or queue a request's execution, and support undoable operations.



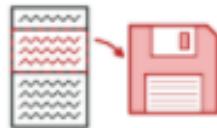
Iterator

Lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.).



Mediator

Lets you reduce chaotic dependencies between objects. The pattern restricts direct communications between the objects and forces them to collaborate only via a mediator object.



Memento

Lets you save and restore the previous state of an object without revealing the details of its implementation.



Observer

Lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.



State

Lets an object alter its behavior when its internal state changes. It appears as if the object changed its class.



Strategy

Lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.

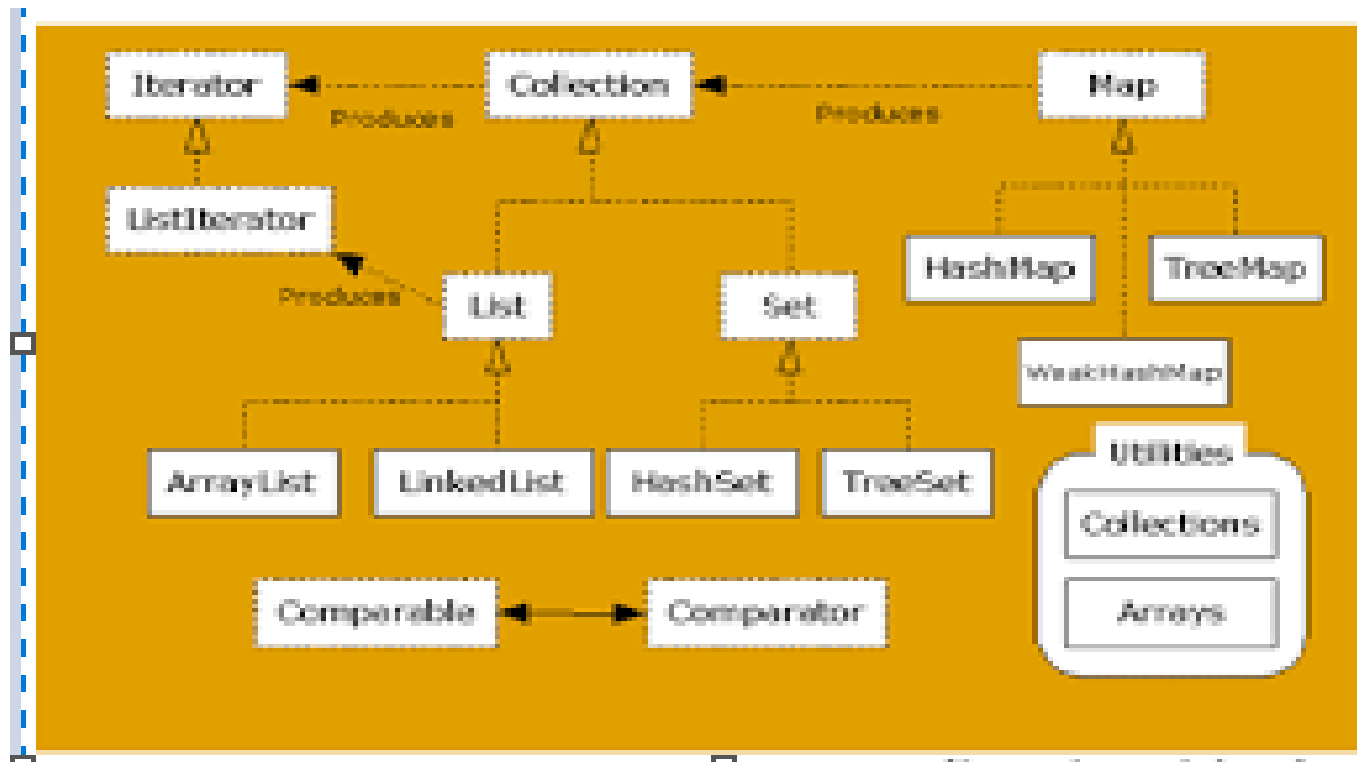


Template Method

Defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure.

Iterator Pattern

- Needed to traverse thru elements of a collection
- Methods declared by Iterator: `hasNext()`, `next()`, `remove()`
- Steps for using iterator:
 1. Obtain an Iterator by calling the collection's **iterator()** method
 2. Loop as long as **hasNext()** returns true
 3. Obtain each element by calling **next()**



Iterable vs. Iterator

SL	Iterable	Iterator
1.	Interface for producing an Iterator. If a collection is iterable, then it can be iterated using an iterator and hence can be used in a for-each loop.	Interface to cycle thru elements. Related to Collection.
2.	The target element of the forEach loop should be iterable. An Iterable should be able to produce any number of valid Iterators.	Use Iterator to get the objects one by one from the Collection: public interface Collection<E> extends Iterable <E>;
3.	Contains only one method iterator(): interface Iterable <T> { Iterator <T> iterator(); // prototype } For example: public class Person implements Iterable { private List<Person> persons = new ArrayList<Person>(); public Iterator <Person> iterator() { // body return this.persons.iterator(); } }	Contains three methods, hasNext(), next(), remove(): public interface Iterator <E> { boolean hasNext(); E next(); void remove(); }
4.	Present in java.lang package	Present in java.util package

Demo: Iterator Pattern

```
import java.util.*;
public class IteratorDemo {
    public static void main(String[] args) {
        // Create a list of strings
        ArrayList<Integer> al = new ArrayList<Integer>();

        // Add entries to the array list
        al.add(50); al.add(10); al.add(30); al.add(60); al.add(20);

        System.out.println("Original List> Size: " + al.size() );
        System.out.println("Original List> Elements: " + al); // use toString() representation

        al.sort(Comparator.naturalOrder()); // Sort arraylist
        Collections.sort( al ); // Looks simpler
        System.out.println("Sorted List> Elements: " + al); // use toString() representation

        Iterator<Integer> itr = al.iterator(); // Obtain an iterator
        // Use Iterator to remove Gamma from the list
        while (itr.hasNext()) // Check whether next element exists
            if ( itr.next().equals(50) ) // Pick up next element
                itr.remove(); // Remove element

        int sum = 0; // Add element values
        for (int x: al) {
            System.out.print(x + " "); // Display next element
            sum = sum + x;
        }
        System.out.printf("\nSum: %d\n", sum);

        System.out.print( "Iterator used on: " + itr.getClass() );
    }
}
```

Additional reading: Builder Design Pattern

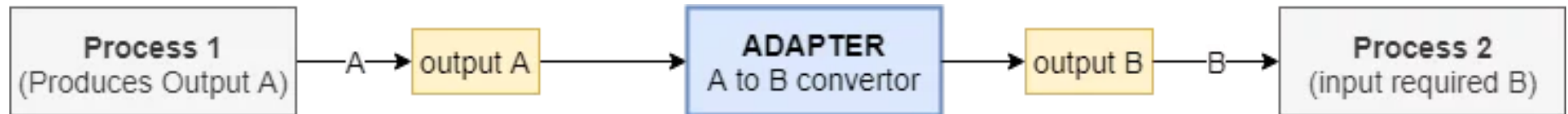
- Consider building a pizza; too many parameters:

Pizza(Size size, Boolean onion, Boolean cheese, Boolean olives, Boolean tomato, Boolean corn, Boolean mushroom, Sauce sauceType);

- Separate required fields from optional ones and move construction logic out of the object class to a separate static inner class (Builder class)
- Builder class has a constructor only for mandatory parameters and setter methods for all the optional parameters.
- A build() method to combine everything and return immutable complete object:
Pizza pizza=new Pizza.Builder(Size.medium).onion(true).olives(true).build();
- No public constructor and objects created only using the Builder class.

Additional Reading -- Adapter Pattern

- Allows incompatible interfaces between classes to work together without modifying their source code



- Above figure shows the basic structure and flow of the adapter design pattern with Process1 producing OutputA and Process2 requiring input from Process1.
- If Process 1's output is not of the same form as that required by Process2, an adapter is introduced between both the processes. It converts OutputA into a format which Process2 can use i.e OutputB.
- Objects must be similar for adapter creation, e.g. adapting a Student object to Department object is unlikely, but Adapting a Student object to Employee object is possible

Additional Reading: Use of iterable

```
import static java.lang.String.format;
import java.util.*;
// Person class
class Person {
    private String firstName, lastName;
    private int age;
    public Person(){ }
    public Person(String firstName, String lastName, int age) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
    }
    public String getFirstName() { return firstName; }
    public String getLastName() { return lastName; }
    public int getAge() { return age; }
    public void setFirstName(String firstName) { this.firstName =
firstName; }
    public void setLastName(String lastName) { this.lastName =
lastName; }
    public void setAge(int age) { this.age = age; }
    @Override
    public String toString() { return format("First Name:\t%s\tLast
Name:\t%s\tAge:\t%d", firstName, lastName, age); }
```

Additional Reading: Use of iterable

```
// PersonArrayList class
public class PersonArrayList implements Iterable<Person> {
    private List<Person> persons;
    private static final int MIN_AGE = 10;
    public PersonArrayList() { persons = new ArrayList<Person>(MIN_AGE); }
    public PersonArrayList(int age) { persons = new ArrayList<Person>(age); }
    public void addPerson(Person p) { persons.add(p); }
    public void removePerson(Person p) { persons.remove(p); }
    public int age() { return persons.size(); }
    @Override
    public Iterator<Person> iterator() { return persons.iterator(); }

    public static void main(String[] args) {
        Person p1 = new Person("Adithya", "Sai", 20);
        Person p2 = new Person("Jai", "Dev", 30);
        PersonArrayList pList = new PersonArrayList();
        pList.addPerson(p1);
        pList.addPerson(p2);
        for (Person person : pList) {
            System.out.println(person);
        }
    }
}
```