

1. Singleton Pattern

Ensure a class has only one instance, and provide a global point of access to it.

```
public class Singleton {
    private static final Singleton singleton = new Singleton();
    //
    private Singleton(){
    }
    //
    public static Singleton getSingleton(){
        return singleton;
    }
    //
    static
    public static void doSomething(){
    }
}
```

Web

IO

static

```
public class Singleton {
    private static Singleton singleton = null;
    //
    private Singleton(){
    }
    //
    public static Singleton getSingleton(){
        if(singleton == null){
            singleton = new Singleton();
        }
    }
}
```

```

        }
        return singleton;
    }
}

```

```

getSingleton
synchronized

```

```

synchronized

```

```

getSingleton

```

2.

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Product

Creator

ConcreteCreator

```

public class ConcreteCreator extends Creator {
    public <T extends Product> T createProduct(Class<T> c){
        Product product=null;
        try {
            product =
(Product)Class.forName(c.getName()).newInstance();
        } catch (Exception e) {
            //
        }
        return (T)product;
    }
}

```

ProductFactory

prMap

jdbc

3. Abstract Factory Pattern

Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

```
public abstract class AbstractCreator {  
    //    A  
    public abstract AbstractProductA createProductA();  
    //    B  
    public abstract AbstractProductB createProductB();  
}
```

4. Template Method Pattern

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

AbstractClass

final

ConcreteClass1 ConcreteClass2

↳

↳

5. Builder Pattern

Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Product

BenzModel BMWModel
Builder

CarBuilder

ConcreteBuilder

BenzBuilder

BMWBuilder

Director

Builder

6. Proxy Pattern

Provide a surrogate or placeholder for another object to control access to it.

Subject

RealSubject

Proxy

GamePlayerProxy

getProxy

$\frac{1}{2}$

$\frac{1}{2}$

Subject

Advice

Client

7. Prototype Pattern

Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

```
public class PrototypeClass implements Cloneable{
    //      Object
    @Override
    public PrototypeClass clone(){
        PrototypeClass prototypeClass = null;
        try {
            prototypeClass = (PrototypeClass)super.clone();
        } catch (CloneNotSupportedException e) {
            //
        }
    }
}
```

```

    }
    return prototypeClass;
}
}

```

Cloneable

clone

new

13.4

new

Object

clone

int long char string

```
thing.arrayList = (ArrayList<String>)this.arrayList.clone();
```

8.

Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

Mediator

Concrete Mediator

Colleague

Self-

Method

Dep-Method

```
public abstract class Mediator {  
    //  
    protected ConcreteColleague1 c1;  
    protected ConcreteColleague2 c2;  
    //    getter/setter  
    public ConcreteColleague1 getC1() {  
        return c1;  
    }  
    public void setC1(ConcreteColleague1 c1) {  
        this.c1 = c1;  
    }  
    public ConcreteColleague2 getC2() {  
        return c2;  
    }  
    public void setC2(ConcreteColleague2 c2) {  
        this.c2 = c2;  
    }  
    //  
    public abstract void doSomething1();  
    public abstract void doSomething2();  
}
```

ps

getter/setter

9.

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Receive

Group
Command

Invoker

DOS GUI

10.

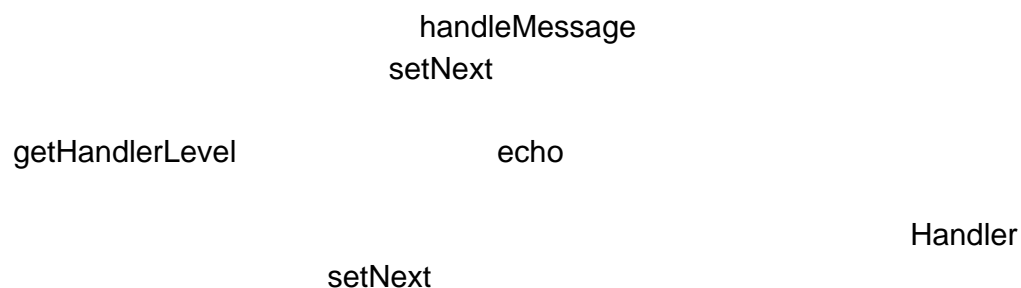
Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

```
public abstract class Handler {
    private Handler nextHandler;
    //
    public final Response handleMessage(Request request){
        Response response = null;
        //
        if(this.getHandlerLevel().equals(request.getRequestLevel())){
            response = this.echo(request);
        }
    }
}
```

```

        }else{ //
            //
            if(this.nextHandler != null){
                response =
this.nextHandler.handleMessage(request);
            }else{
                //
            }
        }
        return response;
    }
    //
    public void setNext(Handler _handler){
        this.nextHandler = _handler;
    }
    //
    protected abstract Level getHandlerLevel();
    //
    protected abstract Response echo(Request request);
}

```



11. Decorator Pattern

Attach additional responsibilities to an object dynamically keeping the same interface. Decorators provide a flexible alternative to subclassing for extending functionality.

Component

Component

Component

ConcreteComponent

ConcreteComponent

Decorator

private

Component

ConcreteDecoratorA

ConcreteDecoratorB

12. Strategy Pattern

Define a family of algorithms, encapsulate each one, and make them interchangeable.

Context

Strategy

AlgorithmInterface

algorithm $\frac{1}{2}$ $\frac{1}{2}$
ConcreteStrategy

```

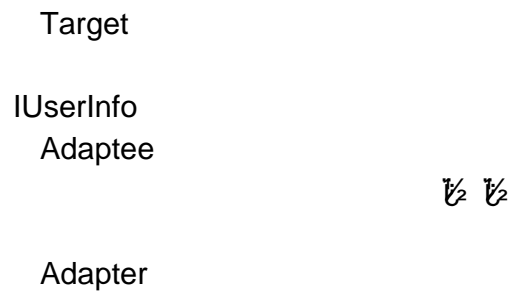
public enum Calculator {
    //
    ADD("+"){
        public int exec(int a,int b){
            return a+b;
        }
    },
    //
    SUB("-"){
        public int exec(int a,int b){
            return a - b;
        }
    };
    String value = "";
    //
    private Calculator(String _value){
        this.value = _value;
    }
    //
    public String getValue(){
        return this.value;
    }
    //
    public abstract int exec(int a,int b);
}

```

public final static

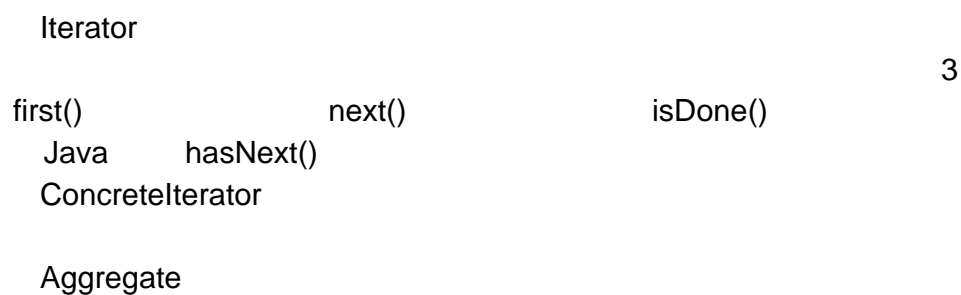
13. Adapter Pattern

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.



14. Iterator Pattern

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.



Java

Concrete Aggregate

iterator()

createIterator()

ps

collection

java

java

15. (Composite Pattern)

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Component

getInfo

Leaf

Composite

```

public class Composite extends Component {
    //
    private ArrayList<Component> componentArrayList = new
ArrayList<Component>();
    //
    public void add(Component component){
        this.componentArrayList.add(component);
    }
    //
    public void remove(Component component){
this.componentArrayList.remove(component);
    }
    //
    public ArrayList<Component> getChildren(){
        return this.componentArrayList;
    }
}

```

16. Observer Pattern

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Subject

Observer

update

ConcreteSubject

ConcreteObserver

```
public abstract class Subject {  
    //  
    private Vector<Observer> obsVector = new Vector<Observer>();  
    //  
    public void addObserver(Observer o){  
        this.obsVector.add(o);  
    }  
    //  
    public void delObserver(Observer o){  
        this.obsVector.remove(o);  
    }  
    //  
    public void notifyObservers(){  
        for(Observer o: this.obsVector){  
            o.update();  
        }  
    }  
}
```

```
}  
}
```

½ ½

17. Facade Pattern

Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Facade

subsystem

½ ½

18. Memento Pattern

Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

Originator

Memento javabean
Originator

Caretaker javabean

rollback

for

clone

HashMap BeanUtils backupProp HashMap
restoreProp

```
BeanUtil
public class BeanUtils {
    // bean                      Hashmap
    public static HashMap<String, Object> backupProp(Object bean){
        HashMap<String, Object> result = new
HashMap<String, Object>();
        try {
```

```

        // Bean
        BeanInfo
beanInfo=Introspector.getBeanInfo(bean.getClass());
        //
        PropertyDescriptor[]
descriptors=beanInfo.getPropertyDescriptors();
        //
        for(PropertyDescriptor des: descriptors){
            //
            String fieldName = des.getName();
            //
            Method getter = des.getReadMethod();
            //
            Object fieldValue=getter.invoke(bean, new
Object[]{});
            if(!fieldName.equalsIgnoreCase("class")){
                result.put(fieldName, fieldValue);
            }
        }
    } catch (Exception e) {
        //
    }
    return result;
}
// HashMap bean
public static void restoreProp(Object bean, HashMap<String, Object>
propMap){
try {
        // Bean
        BeanInfo beanInfo =
Introspector.getBeanInfo(bean.getClass());
        //
        PropertyDescriptor[] descriptors =
beanInfo.getPropertyDescriptors();
        //
        for(PropertyDescriptor des: descriptors){
            //
            String fieldName = des.getName();
            //
            if(propMap.containsKey(fieldName)){
                //
                Method setter = des.getWriteMethod();
                setter.invoke(bean, new
Object[]{propMap.get(fieldName)});

```

```

        }
    }
} catch (Exception e) {
    //
    System.out.println("shi t");
    e.printStackTrace();
}
}
}
}

```

Originator IMemento Memento IMemento

19. Visitor Pattern

Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Visitor visit

ConcreteVisitor

Element accept

ConcreteElement accept visitor.visit(this)

ObjectStruture List Set Map

½ ½

20.

Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

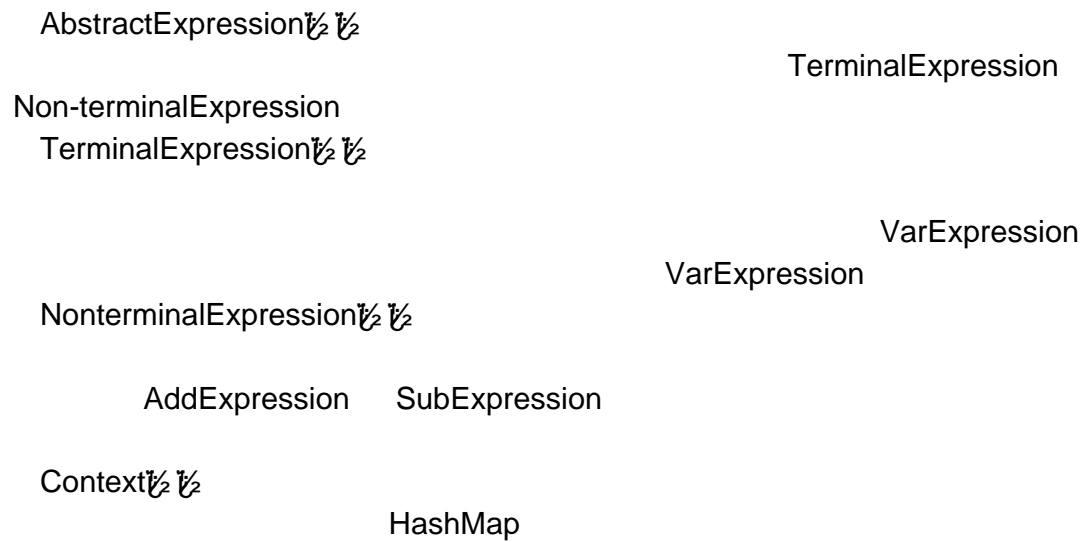
State ½ ½

ConcreteState ½ ½

Context ½ ½

21. Interpreter Pattern

Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.



shell JRuby Groovy Java

22. Flyweight Pattern

Use sharing to support large numbers of fine-grained objects efficiently.

intrinsic extrinsic

Flyweight

ConcreteFlyweight

unsharedConcreteFlyweight

FlyweightFactory

```
public class FlyweightFactory {  
    //  
    private static HashMap<String, Flyweight> pool = new  
    HashMap<String, Flyweight>();  
    //  
    public static Flyweight getFlyweight(String Extrinsic){  
        //  
        Flyweight flyweight = null;  
        //  
        if(pool.containsKey(Extrinsic)){  
            flyweight = pool.get(Extrinsic);  
        }else{  
            //  
            flyweight = new ConcreteFlyweight1(Extrinsic);  
            //  
            pool.put(Extrinsic, flyweight);  
        }  
        return flyweight;  
    }  
}
```

java

String int

23. Bridge Pattern

Decouple an abstraction from its implementation so that the two can vary independently.

Abstraction

Implementor

RefinedAbstraction

ConcreteImplementor

N

Single Responsibility Principle

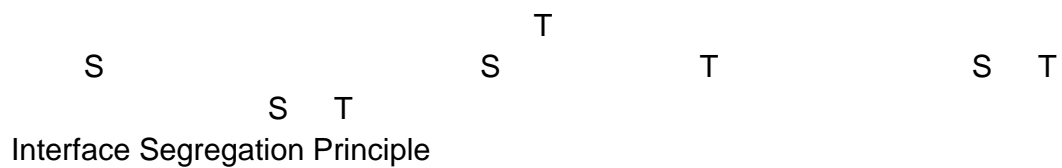
ps

Liskov Substitution Principle

Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.

- 1.
- 2.
- 3.

4.



Dependence Inversion Principle

java

Setter

setter

Open Closed Principle

$a*b*c$

$a*b+c$

JSP

Swing