Laborator 6

1. Justificarea introducerii interfetelor – complete decoupling si design patternul strategy

package interfaces.classprocessor;

import java.util.\*;

class Processor

{

public String name() {

return getClass().getSimpleName();

}

Object process(Object input) { return input; }

}

class Upcase extends Processor

{

String process(Object input)

{ // Covariant return

return ((String)input).toUpperCase();

}

}

class Downcase extends Processor {

String process(Object input) {

return ((String)input).toLowerCase();

}

}

class Splitter extends Processor {

String process(Object input) {

// The split() argument divides a String into pieces:

return Arrays.toString(((String)input).split(" "));

}

}

public class Apply {

public static void process(Processor p, Object s) {

print("Using Processor " + p.name());

print(p.process(s));

}

public static String s = "Disagreement with beliefs is by definition incorrect";

public static void main(String[] args) {

process(new Upcase(), s);

process(new Downcase(), s);

process(new Splitter(), s);

}

}

//: interfaces/filters/Waveform.java

package interfaces.filters;

public class Waveform {

private static long counter;

private final long id = counter++;

public String toString() { return "Waveform " + id; }

} ///:~

//: interfaces/filters/Filter.java

package interfaces.filters;

public class Filter {

public String name() {

return getClass().getSimpleName();

}

public Waveform process(Waveform input) { return input; }

} ///:~

//: interfaces/filters/LowPass.java

package interfaces.filters;

public class LowPass extends Filter {

double cutoff;

public LowPass(double cutoff) { this.cutoff = cutoff; }

public Waveform process(Waveform input) {

return input; // Dummy processing

}

} ///:~

//: interfaces/filters/HighPass.java

package interfaces.filters;

public class HighPass extends Filter {

double cutoff;

public HighPass(double cutoff) { this.cutoff = cutoff; }

public Waveform process(Waveform input) { return input; }

} ///:~

//: interfaces/filters/BandPass.java

package interfaces.filters;

public class BandPass extends Filter {

double lowCutoff, highCutoff;

public BandPass(double lowCut, double highCut) {

lowCutoff = lowCut;

highCutoff = highCut;

}

public Waveform process(Waveform input) { return input; }

} ///:~

Atentie: Apply.process nu poate fi aplicata pe obiecte din ierarhia Filter (cu toate, ca teoretic, nu ar trebui sa fie probleme). => clasa Apply se rescrie, Processor este transformata in interfata

public interface Processor {

String name();

Object process(Object input);

}

public abstract class StringProcessor implements Processor{

public String name() {

return getClass().getSimpleName();

}

public abstract String process(Object input);

public static String s =

"If she weighs the same as a duck, she’s made of wood";

public static void main(String[] args) {

Apply.process(new Upcase(), s);

Apply.process(new Downcase(), s);

Apply.process(new Splitter(), s);

}

}

class Upcase extends StringProcessor {

public String process(Object input) { // Covariant return

return ((String)input).toUpperCase();

}

}

class Downcase extends StringProcessor {

public String process(Object input) {

return ((String)input).toLowerCase();

}

}

class Splitter extends StringProcessor {

public String process(Object input) {

return Arrays.toString(((String)input).split(" "));

}

}

1. Desing patternul Adapter: se doreste ca o clasa importata ( de exemplu Filter) sa poata fi utilizata in conjunctie cu interfata definite de noi (interfata Processor)

class FilterAdapter implements Processor {

Filter filter;

public FilterAdapter(Filter filter) {

this.filter = filter;

}

public String name() { return filter.name(); }

public Waveform process(Object input) {

return filter.process((Waveform)input);

}

}

public class FilterProcessor {

public static void main(String[] args) {

Waveform w = new Waveform();

Apply.process(new FilterAdapter(new LowPass(1.0)), w);

Apply.process(new FilterAdapter(new HighPass(2.0)), w);

Apply.process(new FilterAdapter(new BandPass(3.0, 4.0)), w);

}

}

1. Design patternul Factory

**interface** Service {

**void** method1();

**void** method2();

}

**interface** ServiceFactory {

Service getService();

}

**class** Implementation1 **implements** Service {

Implementation1() {} // Package access

**public** **void** method1() { System.*out*.println("Implementation1 method1");}

**public** **void** method2() { System.*out*.println("Implementation1 method2");}

}

**class** Implementation1Factory **implements** ServiceFactory {

**public** Service getService() {

**return** **new** Implementation1();

}

}

**class** Implementation2 **implements** Service {

Implementation2() {} // Package access

**public** **void** method1() {System.*out*.println("Implementation2 method1");}

**public** **void** method2() {System.*out*.println("Implementation2 method2");}

}

**class** Implementation2Factory **implements** ServiceFactory {

**public** Service getService() {

**return** **new** Implementation2();

}

}

**public** **class** Factories {

**public** **static** **void** serviceConsumer(ServiceFactory fact) {

Service s = fact.getService();

s.method1();

s.method2();

}

**public** **static** **void** main(String[] args) {

*serviceConsumer*(**new** Implementation1Factory());

// Implementations are completely interchangeable:

*serviceConsumer*(**new** Implementation2Factory());

}

}

Atentie: implementarile sunt absolut interschimbabile intre ele la utilizarea lor (in metoda ServiceConsumer)

1. Crearea claselor interioare. Se oberva faptul ca de obicei, clasa exterioara are o metoda care returneeaza o referinta catre clasa inner

**public** **class** Parcel2 {

**class** Contents {

**private** **int** i = 11;

**public** **int** value() { **return** i; }

}

**class** Destination {

**private** String label;

Destination(String whereTo) {

label = whereTo;

}

String readLabel() { **return** label; }

}

**public** Destination to(String s) {

**return** **new** Destination(s);

}

**public** Contents contents() {

**return** **new** Contents();

}

**public** **void** ship(String dest) {

Contents c = contents();

Destination d = to(dest);

System.*out*.println(d.readLabel());

}

**public** **static** **void** main(String[] args) {

Parcel2 p = **new** Parcel2();

p.ship("Tasmania");

Parcel2 q = **new** Parcel2();

// Defining references to inner classes:

Parcel2.Contents c = q.contents();

Parcel2.Destination d = q.to("Borneo");

}

}

Utilizarea claselor interioare pentru a implementa design patternul Iterator

**interface** Selector {

**boolean** end();

Object current();

**void** next();

}

**public** **class** Sequence {

**private** Object[] items;

**private** **int** next = 0;

**public** Sequence(**int** size) { items = **new** Object[size]; }

**public** **void** add(Object x) {

**if**(next < items.length)

items[next++] = x;

}

**private** **class** SequenceSelector **implements** Selector {

**private** **int** i = 0;

**public** **boolean** end() { **return** i == items.length; }

**public** Object current() { **return** items[i]; }

**public** **void** next() { **if**(i < items.length) i++; }

}

**public** Selector selector() {

**return** **new** SequenceSelector();

}

**public** **static** **void** main(String[] args) {

Sequence sequence = **new** Sequence(10);

**for**(**int** i = 0; i < 10; i++)

sequence.add(Integer.*toString*(i));

Selector selector = sequence.selector();

**while**(!selector.end()) {

System.*out*.print(selector.current() + " ");

selector.next();

}

}

}

In exemplul de mai jos, rog studentii sa converteasca clasele Contents si Destination de mai sus in interfete

**class** Parcel4 {

**private** **class** PContents **implements** Contents {

**private** **int** i = 11;

**public** **int** value() { **return** i; }

}

**protected** **class** PDestination **implements** Destination {

**private** String label;

**private** PDestination(String whereTo) {

label = whereTo;

}

**public** String readLabel() { **return** label; }

}

**public** Destination destination(String s) {

**return** **new** PDestination(s);

}

**public** Contents contents() {

**return** **new** PContents();

}

}

**public** **class** TestParcel {

**public** **static** **void** main(String[] args) {

Parcel4 p = **new** Parcel4();

Contents c = p.contents();

Destination d = p.destination("Tasmania");

// Illegal -- can’t access private class:

//! Parcel4.PContents pc = p.new PContents();

}

} ///:~

1. Creati o interfata cu cel putin o metoda, intr-un package separate. Creati o alta clasa intr-un alt package. In aceasta, adaugati o clasa interioara care implementeaza interfata. In cel de-al treilea package, mosteniti o noua clasa (din clasa din packageul al doilea), si adaugati in interiorul acesteia o metoda care returneaza un obiect de tipul clasei interioare protected, realizand upcast la interfata la momentul comenzii return.
2. Inner class in interiorul unei metode

Clasa PDestination este vizibila si poate fi accesata doar in interiorul metodei destination

**public** **class** Parcel5 {

**public** Destination destination(String s) {

**class** PDestination **implements** Destination {

**private** String label;

**private** PDestination(String whereTo) {

label = whereTo;

}

**public** String readLabel() { **return** label; }

}

**return** **new** PDestination(s);

}

**public** **static** **void** main(String[] args) {

Parcel5 p = **new** Parcel5();

Destination d = p.destination("Tasmania");

}

} ///:~

1. Clase interioare anonime

**public** **class** Parcel7 {

**public** Contents contents() {

**return** **new** Contents() { // definitia unei clase interioare anonime

**private** **int** i = 11;

**public** **int** value() { **return** i; }

}; // Semicolon required in this case

}

**public** **static** **void** main(String[] args) {

Parcel7 p = **new** Parcel7();

Contents c = p.contents();

}

} ///:~

**In clasele anonime, argumentele trebuie sa aiba referintele final pentru a putea fi utilizate**

**public** **class** Parcel9 {

// Argument must be final to use inside anonymous inner class:

**public** Destination destination(**final** String dest) {

**return** **new** Destination() {

**private** String label = dest;

**public** String readLabel() { **return** label; }

};

}

**public** **static** **void** main(String[] args) {

Parcel9 p = **new** Parcel9();

Destination d = p.destination("Tasmania");

}

} ///:~

Clasele anonime trebuie sa realizeze construirea in blocul non-static de intializare

**abstract** **class** Base {

**public** Base(**int** i) {

System.*out*.println("Base constructor, i = " + i);

}

**public** **abstract** **void** f();

}

**public** **class** AnonymousConstructor {

**public** **static** Base getBase(**int** i) {

**return** **new** Base(i) {

{ System.*out*.println("Inside instance initializer"); }

**public** **void** f() {

System.*out*.println("In anonymous f()");

}

};

}

**public** **static** **void** main(String[] args) {

Base base = *getBase*(47);

base.f();

}

}