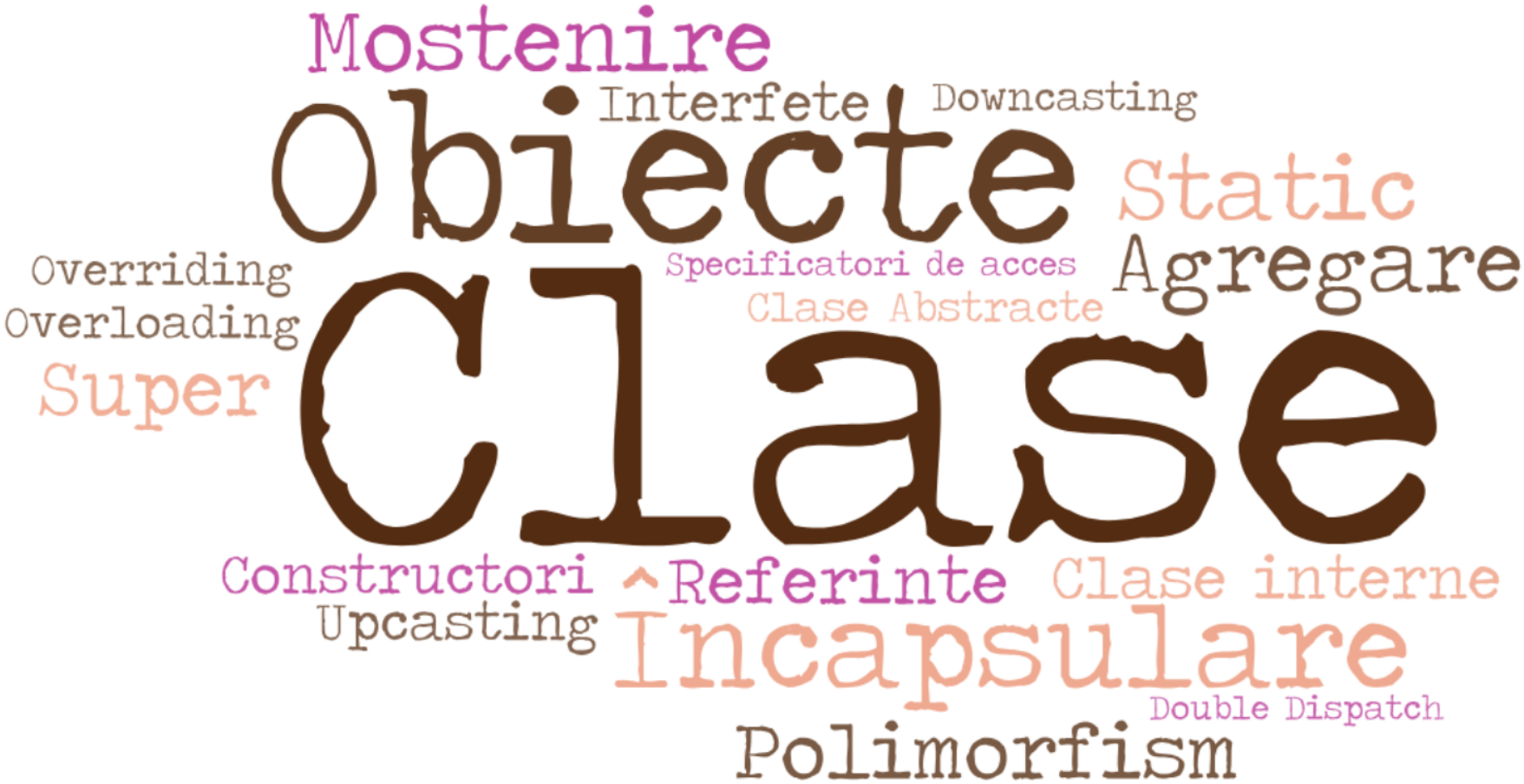
Polimorfism, Double Dispatch, Visitor

Alexandru Olteanu

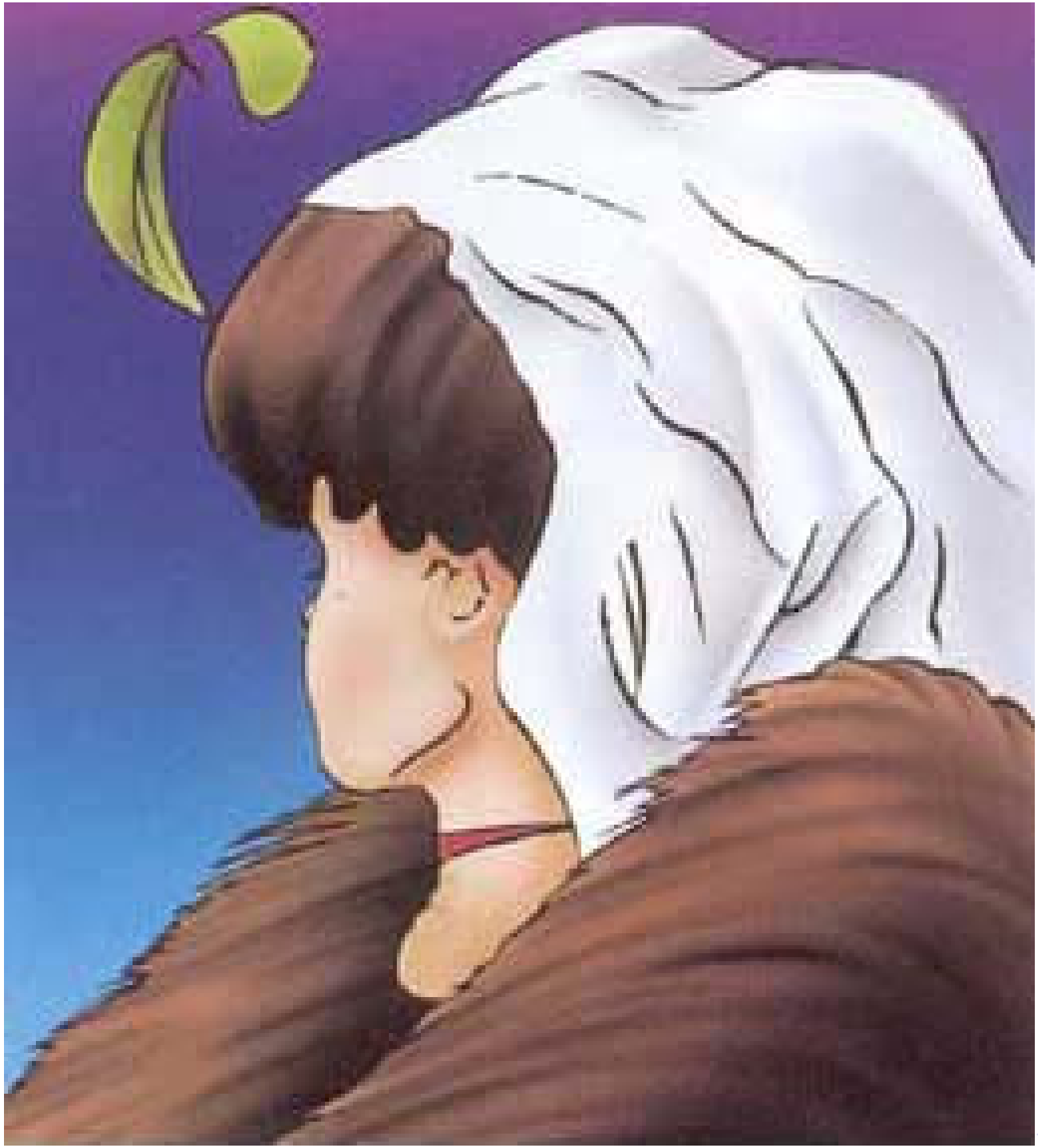
Universitatea Politehnica Bucuresti
Facultatea de Automatică si Calculatoare, Departamentul Calculatoare
alexandru.olteanu@upb.ro

OOP, 2020

Partea 1: Bazele teoretice ale OOP



Polimorfism



Definitia polimorfismului

Definition

Polimorfismul este caracteristica unei entitati de a avea forme diferite (structural, comportamental etc.) in contexte diferite.

Spre exemplu, in OOP, obiectele au aceasta caracteristica

Polimorfism in OOP

```
class Vehicle
interface Diesel
interface Electric
interface PublicTransport
class Hybrid extends Vehicle implements Electric, Diesel
class Bus extends Vehicle implements Diesel, PublicTransport
```

- A Hybrid IS-A Vehicle
- A Hybrid IS-A Electric
- A Hybrid IS-A Diesel
- A Hybrid IS-A Hybrid
- A Hybrid IS-A Object

Polimorfism in OOP

```
class Vehicle
interface Diesel
interface Electric
interface PublicTransport
class Hybrid extends Vehicle implements Electric, Diesel
class Bus extends Vehicle implements Diesel, PublicTransport
```

- A Bus IS-A Vehicle
- A Bus IS-A Diesel
- A Bus IS-A PublicTransport
- A Bus IS-A Bus
- A Bus IS-A Object

Polimorfism in OOP

```
class Vehicle
interface Diesel
interface Electric
interface PublicTransport
class Hybrid extends Vehicle implements Electric, Diesel
class Bus extends Vehicle implements Diesel, PublicTransport
```

Daca avem acelasi membru / metoda in mai multe "fatete" ale unui obiect, care se foloseste?

Polimorfism in OOP

- static polymorphism
- dynamic polymorphism
- parametric polymorphism

Static / Ad-hoc polymorphism (overloading)

Ad-hoc polymorphism (...) allows a polymorphic value to exhibit different behaviors when “viewed” at different types. The most common example of ad-hoc polymorphism is overloading, which associates a single function symbol with many implementations; the compiler (or the runtime system, depending on whether overloading resolution is static or dynamic) chooses an appropriate implementation for each application of the function, based on the types of the arguments.
B. Pierce, "Types and Programming Languages", MIT Press

Static / Ad-hoc polymorphism (overloading)

Static = compile time

```
1 + 2 = 3                          // integer addition
3 + 0.1415 = 3.1415                // float addition
[1,2,3] + [4,5] = [1,2,3,4,5]      // list concatenation
"bab" + "oon" = "baboon"           // string concatenation
red + yellow = orange              // custom class operator+
```

Dynamic polymorphism (overriding)

Late binding, or dynamic binding is a computer programming mechanism in which the method being called upon an object or the function being called with arguments is looked up by name at runtime

```
public class Bike {
    public void show() { System.out.println("bike"); }
}
public class Motorcycle extends Bike {
    public void show() { System.out.println("motor"); }
}
...
    public static void main(String args[]){
        Bike obj=new Motorcycle();
        obj.show();
    }
```

private, final and static methods and variables uses static binding

Functii virtuale

In C++, se va face dynamic binding (la run-time) doar pentru functiile marcate ca fiind virtuale
Daca o functie este supraincarcata fara a fi marcata drept virtuala, se va face static binding (la compilare)

▸ Despre functii virtuale in C++

Dynamic polymorphism (overriding)

Atentie! dynamic binding se refera la metode, nu la campuri

```
public class Bike {
    public int topSpeed = 30;
}
public class Motorcycle extends Bike {
    public int topSpeed = 150;
}
...
    public static void main(String args[]){
        Bike obj=new Motorcycle();
        System.out.println(obj.topSpeed);
    }
```

La afisare va fi folosit campul din referinta (in acest caz superclasa).

- Why are Instance variables of a superclass not overidden in Inheritance ?
- Performance

Dynamic polymorphism (overriding)

Atentie! dynamic binding se refera la metode, nu la campuri

```
public class Bike {
    public int topSpeed = 30;
}
public class Motorcycle extends Bike {
    // public int topSpeed = 150;
    public Motorcycle() {
      topSpeed = 150;
    }
}
...
    public static void main(String args[]){
        Bike obj=new Motorcycle();
        System.out.println(obj.topSpeed);
    }
```

Daca subclasa nu defineste campul cu acelasi nume ca in superclasa, in constructor se va folosi campul din superclasa (mai putin daca acesta este private)

Dynamic polymorphism (overriding)

Atentie! Overloading se rezolva la compile-time (fiecare metoda capata un nume unic) - vezi Static Polymorphism

```
public class Road extends Path {}
public class Motorcycle{
    public void ride(Road r) {
        System.out.println("motor on road");
    }
    public void ride(Path p) {
        System.out.println("motor on path");
    }
}
...
 public static void main(String args[]){
    Motorcycle m = new Motorcycle();
    Path r = new Road();
    m.ride(r);
 }
```

Double dispatch

The problem with Single Dispatch

```
public class Path {}
public class Road extends Path {}
public class Bike {
    public void ride(Road r) { sout("bike on road"); }
    public void ride(Path p) { sout("bike on path"); }
}
public class Motorcycle extends Bike{
    public void ride(Road r) { sout("motor on road"); }
    public void ride(Path p) { sout("motor on path"); }
}
...
 public static void main(String args[]){
    Bike b = new Bike(); Bike m = new Motorcycle();
    Path p = new Path(); Path r = new Road();
    b.ride(p); b.ride(r); m.ride(p); m.ride(r);
 }
```

Double dispatch

Mixing the second Dispatch on top of Single Dispatch

```
public class Path {
    public void echo(Bike b){ b.ride(this); }
}
public class Road extends Path {
    public void echo(Bike b){ b.ride(this); }
}
public class Bike {
    public void ride(Road r) { sout("bike on road"); }
    public void ride(Path p) { sout("bike on path"); }
}
public class Motorcycle extends Bike{
    public void ride(Road r) { sout("motor on road"); }
    public void ride(Path p) { sout("motor on path"); }
}
 public static void main(String args[]){
    Bike b = new Bike(); Bike m = new Motorcycle();
    Path p = new Path(); Path r = new Road();
    // b.ride(p); b.ride(r); m.ride(p); m.ride(r);
    p.echo(b); r.echo(b); p.echo(m); r.echo(m);
```

Parametric polymorphism / Generics

Parametric polymorphism (...), allows a single piece of code to be typed "generically", using variables in place of actual types, and then instantiated with particular types as needed. Parametric definitions are uniform: all of their instances behave the same. (...)
B. Pierce, "Types and Programming Languages", MIT Press

Parametric polymorphism / Generics

o singura implementare (structura de date, algoritm) pentru mai multe tipuri de date

```
List<Player> team = new ArrayList<Player>();
List<Integer> grades = new ArrayList<Integer>();
```

Parametric polymorphism / Generics

o singura implementare (structura de date, algoritm) pentru mai multe tipuri de date

```
List<Player> team = new ArrayList<Player>();
List<Integer> grades = new ArrayList<Integer>();

Collections.sort(team);
Collections.sort(grades);
```

Visitor

GoF Design Patterns

E. Gamma, R. Helm, R. Johnson, J. Vlissides (aka Gang of Four)
"Design Patterns: Elements of Reusable Object-Oriented Software"

Creational	Structural	Behavioral
Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Interpreter Iterator Mediator Memento Observer State Strategy Visitor

Description

Visitor represents 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.

Problem that the pattern solves

- o structura contine obiecte de clase variate, asupra carora vrem sa aplicam operatii ce depind de clasele lor concrete

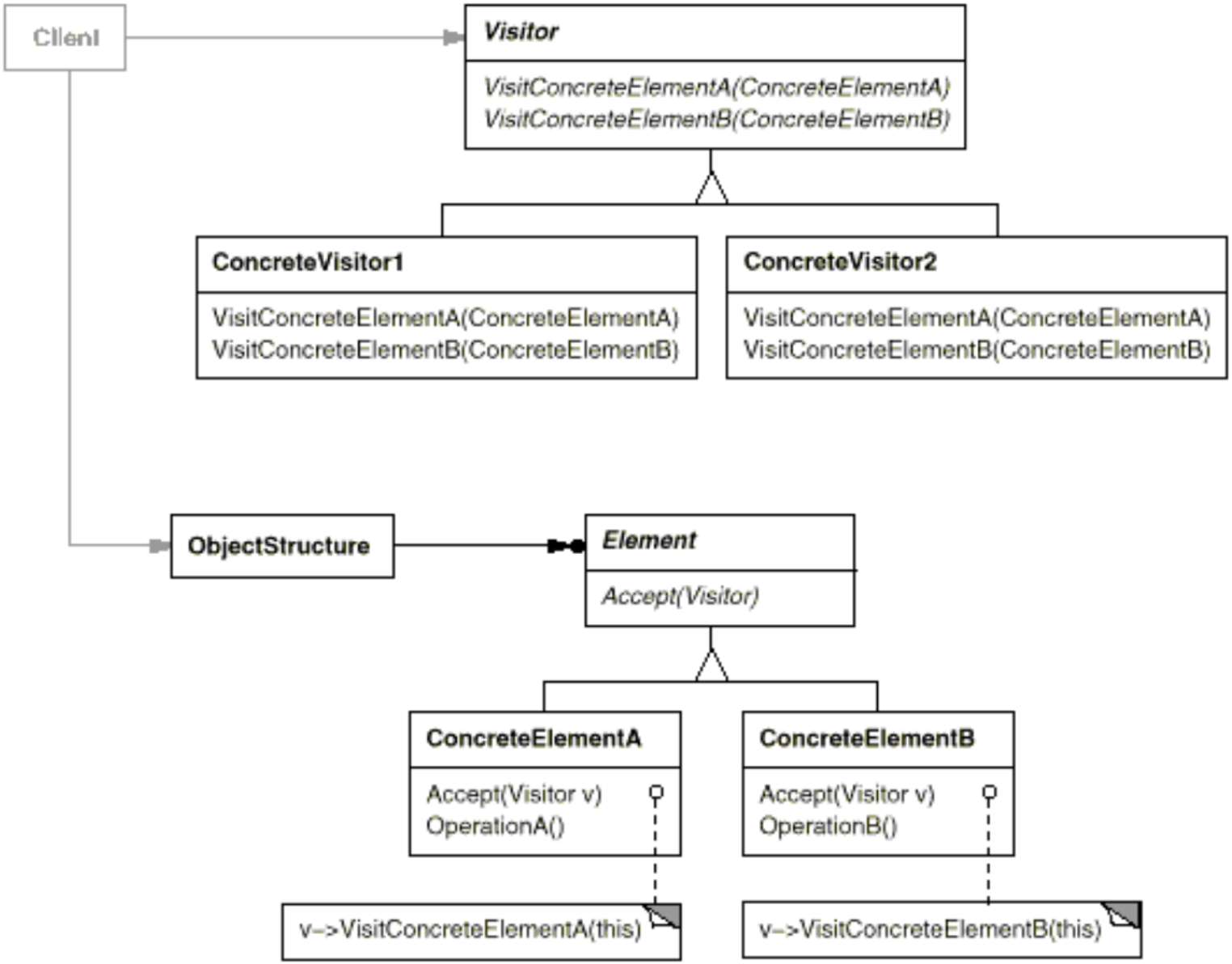
Problem that the pattern solves

- o structura contine obiecte de clase variate, asupra carora vrem sa aplicam operatii ce depind de clasele lor concrete
- operatiile pe care vrem sa le aplicam sunt diferite, fara legatura intre ele si ar trebui sa fie usor extensibile

Problem that the pattern solves

- o structura contine obiecte de clase variate, asupra carora vrem sa aplicam operatii ce depind de clasele lor concrete
- operatiile pe care vrem sa le aplicam sunt diferite, fara legatura intre ele si ar trebui sa fie usor extensibile
- clasele ce definesc structura de obiecte se schimba rar, dar vrem sa definim operatii noi usor (daca si clasele ce definesc structura de obiecte se schimba des, poate e mai bine sa definim operatiile in ele)

Solution Structure



Solution Details

Un client trebuie sa instantieze cate un vizitator concret pentru fiecare operatie si sa traverseze structura de obiecte vizitand fiecare obiect cu fiecare vizitator concret.

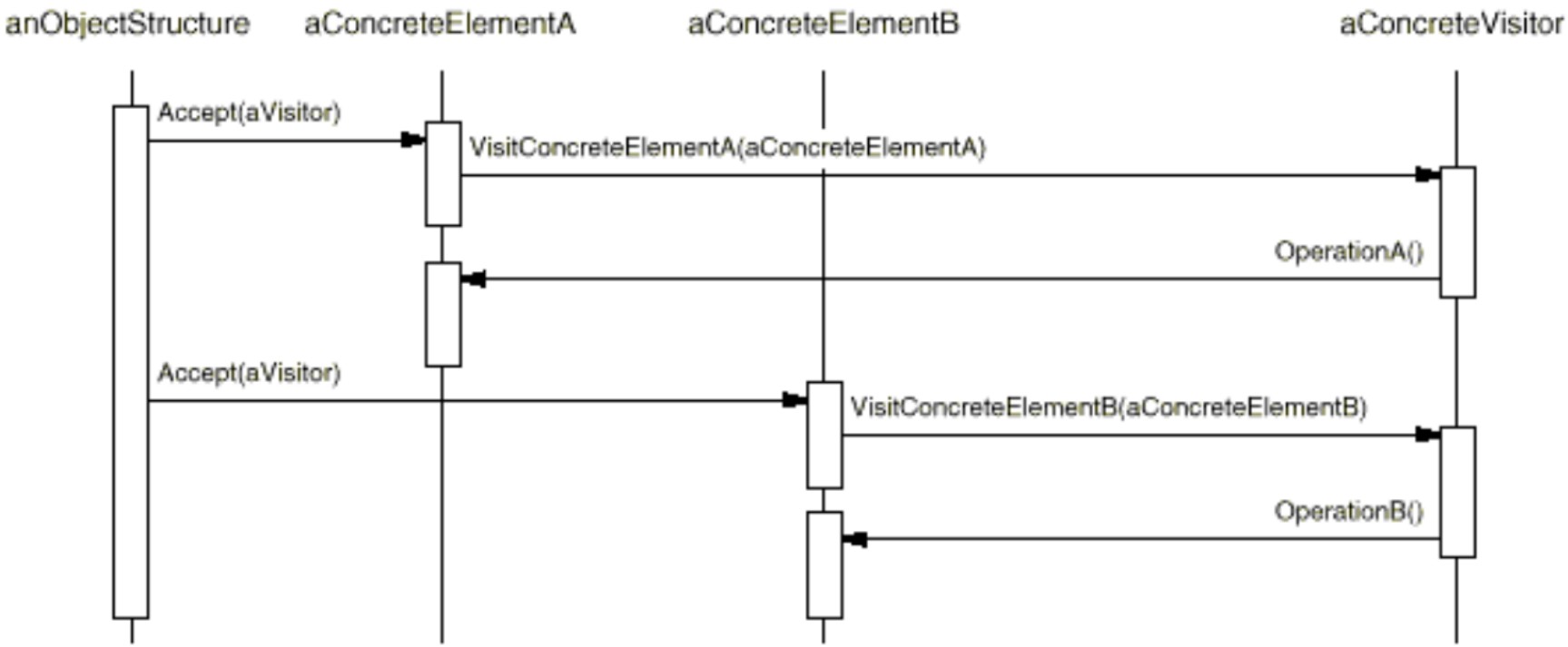
Solution Details

Un client trebuie sa instantieze cate un vizitator concret pentru fiecare operatie si sa traverseze structura de obiecte vizitand fiecare obiect cu fiecare vizitator concret.
Cum se face traversarea?

- de catre structura de obiecte
- de catre vizitator
- intr-un obiect iterator separat

Solution Details

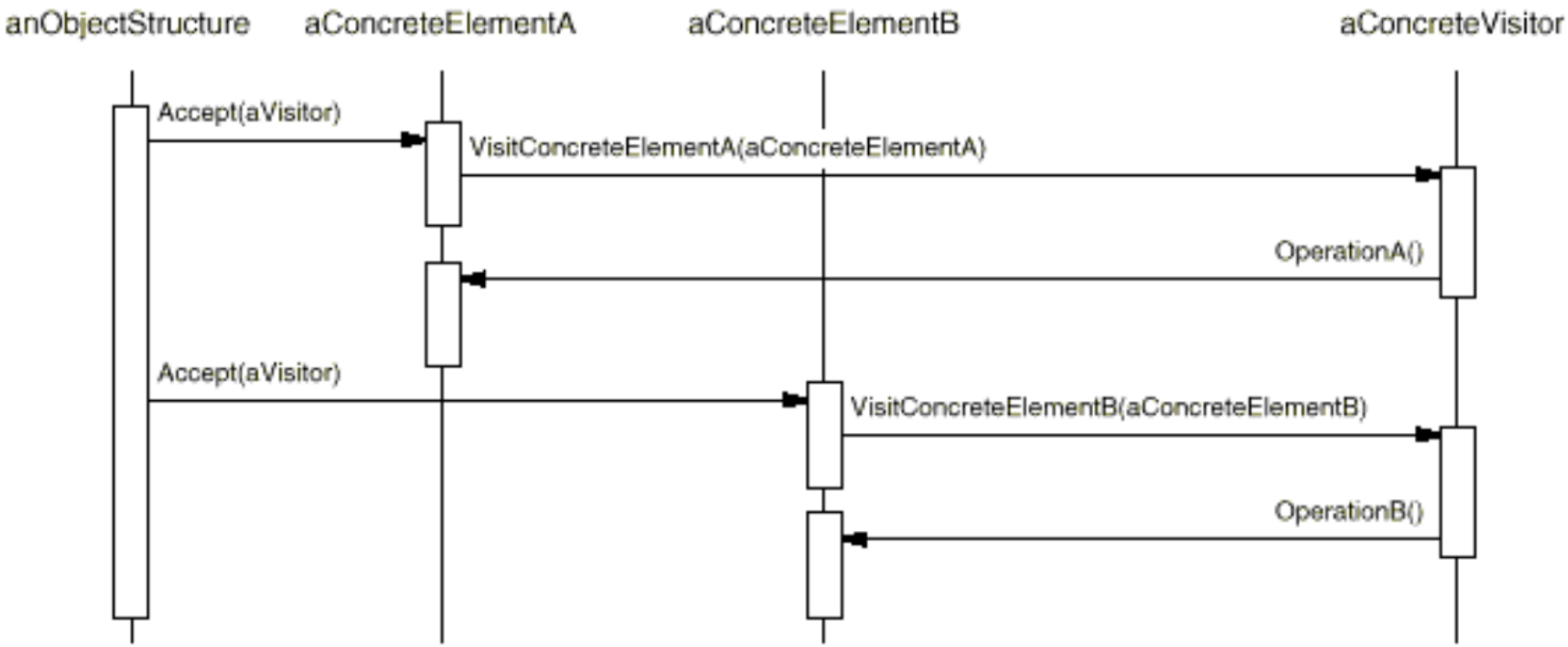
Vizitatorul concret ofera un context in care sa se desfasoare operatia, acumuleaza stare si poate apela metode din obiecte:



* E. Gamma, R. Helm, R. Johnson, J. Vlissides "Design Patterns: Elements of Reusable Object-Oriented Software"

Solution Details

Vizitatorul concret ofera un context in care sa se desfasoare operatia, acumuleaza stare si poate apela metode din obiecte:



* E. Gamma, R. Helm, R. Johnson, J. Vlissides "Design Patterns: Elements of Reusable Object-Oriented Software"

Exista un echilibru care variaza de la o solutie la alta intre ce se desfasoara in vizitator si ce este expus de obiect prin metode

Solution Details

Double Dispatch inseamna ca o actiune (precum Accept) depinde de tipul a doua elemente (in cazul Accept depinde de tipul vizitatorului si al obiectului vizitat).

Solution Details

Double Dispatch inseamna ca o actiune (precum Accept) depinde de tipul a doua elemente (in cazul Accept depinde de tipul vizitatorului si al obiectului vizitat).

	ConcreteVisitor1	ConcreteVisitor2
ConcreteElementA	VisitConcreteElementA	VisitConcreteElementA
ConcreteElementB	VisitConcreteElementB	VisitConcreteElementB

Exercitii

- Incercati sa adaugati o noua clasa de vizitator
- Incercati sa adaugati o noua clasa de obiecte vizitate

Consequences

- adaugarea de operatii este simpla (adaugarea unui nou vizitator vs modificarea fiecare clase de obiecte vizitate)

Consequences

- adaugarea de operatii este simpla (adaugarea unui nou vizitator vs modificarea fiecare clase de obiecte vizitate)
- vizitatorul grupeaza operatiile intr-un mod logic

Consequences

- adaugarea de operatii este simpla (adaugarea unui nou vizitator vs modificarea fiecare clase de obiecte vizitate)
- vizitatorul grupeaza operatiile intr-un mod logic
- adaugarea de noi clase de obiecte vizitate este mai dificila

Consequences

- adaugarea de operatii este simpla (adaugarea unui nou vizitator vs modificarea fiecare clase de obiecte vizitate)
- vizitatorul grupeaza operatiile intr-un mod logic
- adaugarea de noi clase de obiecte vizitate este mai dificila
- vizitarea structurii de obiecte vizitate se poate face in diferite moduri, dar in forma cea mai pura de Visitor, spre deosebire de Iterator, obiectele vizitate nu trebuie neaparat sa instantieze dintr-o ierarhie de clase

Consequences

- adaugarea de operatii este simpla (adaugarea unui nou vizitator vs modificarea fiecare clase de obiecte vizitate)
- vizitatorul grupeaza operatiile intr-un mod logic
- adaugarea de noi clase de obiecte vizitate este mai dificila
- vizitarea structurii de obiecte vizitate se poate face in diferite moduri, dar in forma cea mai pura de Visitor, spre deosebire de Iterator, obiectele vizitate nu trebuie neaparat sa instantieze dintr-o ierarhie de clase
- vizitatorul poate acumula stare

Consequences

- adaugarea de operatii este simpla (adaugarea unui nou vizitator vs modificarea fiecare clase de obiecte vizitate)
- vizitatorul grupeaza operatiile intr-un mod logic
- adaugarea de noi clase de obiecte vizitate este mai dificila
- vizitarea structurii de obiecte vizitate se poate face in diferite moduri, dar in forma cea mai pura de Visitor, spre deosebire de Iterator, obiectele vizitate nu trebuie neaparat sa instantieze dintr-o ierarhie de clase
- vizitatorul poate acumula stare
- implementarea de operatii in obiectele vizitate care sa permita vizitatorului sa isi faca treaba poate compromite incapsularea in clasele de obiecte vizitate

De citit

- Lab 7: Overriding, Overloading and Visitor pattern
- Scurt tutorial despre Double Dispatch
- Despre functii virtuale in C++