Medii de proiectare și programare

2024-2025
Curs 7

Conținut curs 7

- Networking C# (continuare)
- Remote Procedure Call
- Aplicații distribuite cross-platform
- Google Protobuff

Sincronizarea threadurilor

- Diferite tipuri:
  - *Blocarea exclusivă*: doar un singur thread poate executa o porțiune de cod la un moment dat.
    - `lock`, `Mutex`, and `SpinLock`.
  - *Blocarea nonexclusivă:* limitarea concurenței.
    - `Semaphore` and `ReaderWriterLock`.
  - *Semnalizarea*: un thread poate bloca execuția până la primirea unei notificării de la unul sau mai multe threaduri.
    - `ManualResetEvent`, `AutoResetEvent`, `CountdownEvent` și `Barrier`.

Sincronizarea threadurilor –Blocarea

- Instrucțiunea `lock`:

```
lock(locker_obj){
  //code to execute
}
```

- `locker_obj` - tip referință.
- Doar un singur thread poate obține accesul la un moment dat. Dacă mai multe threaduri încearcă să obțină accesul, ele sunt puse într-o coadă și primesc accesul pe baza regulii “primul venit-primul servit”.
- Dacă un alt thread a obținut deja accesul, threadul curent nu își continuă execuția până nu obține accesul.

Sincronizarea threadurilor - Signaling

- *Event wait handles* - construcții simple pentru semnalizare:
  - **`EventWaitHandle`**- reprezintă un eveniment pentru sincronizarea threadurilor. Unul sau mai multe threaduri blochează execuția folosind un **`EventWaitHandle`** până când un alt thread apelează metoda **`Set`** permițând execuția unuia sau mai multor threaduri aflate în așteptare.
  - **`AutoResetEvent`**, **`ManualResetEvent`**
  - **`CountdownEvent`** (Framework 4.0)
- **`AutoResetEvent`** notifică un thread aflat în așteptare de apariția unui eveniment (doar un singur thread).
  - **`Set()`** - eliberează un thread aflat în așteptare
  - **`WaitOne()`** - threadul așteaptă apariția unui eveniment

Observații:

1. Dacă **`Set`** este apelată când nici un thread nu se află în așteptare, handle -ul așteaptă până când un thread apelează metoda **`WaitOne`**.
2. Apelarea metodei **`Set`** de mai multe ori când nici un thread nu este în așteptare nu va permite mai multor threaduri obținerea accesului când apelează metoda **`WaitOne`**.

Sincronizarea threadurilor - Signaling

- **`ManualResetEvent`** (asemănător **`AutoResetEvent)-`** notifică toate threadurile aflate în așteptare la apariția unui eveniment.
- Crearea unui event wait handle:
  - Constructori:

```
AutoResetEvent waitA=new AutoResetEvent(false);
AutoResetEvent waitA=new AutoResetEvent(true); //calls Set
```

```
ManualResetEvent waitM=new ManualResetEvent(false);
```

  - Clasa **`EventWaitHandle`**

```
var auto = new EventWaitHandle (false, EventResetMode.AutoReset);
var manual = new EventWaitHandle (false, EventResetMode.ManualReset);
```

- Distrugerea unui wait handle:
  - Apelul metodei **`Close`** pentru eliberarea resurselor sistemului de operare.
  - Ștergerea referințelor pentru a permite garbage collector-ului distrugerea obiectului.

Exemplu Signaling

```
class WaitHandleExample
{
 static EventWaitHandle waitHandle = new AutoResetEvent (false);
 static void Main()
 {
    new Thread (Worker).Start();
    Thread.Sleep (1000);     // Pause for a second...
    waitHandle.Set();    // Wake up the Worker.
 }
 static void Worker()
 {
    Console.WriteLine ("Waiting...");
    waitHandle.WaitOne();    // Wait for notification
    Console.WriteLine ("Notified");
 }
}
```

Task-uri C#

- ***Limitările threadurilor***:
  - Se pot transmite ușor date unui thread, dar nu se poate obține la fel de ușor rezultatul execuției unui thread.
  - Dacă execuția threadului aruncă o excepție, tratarea excepției și retransmiterea ei este mai dificil de implementat.
  - Nu se poate seta ca un thread să execute altceva când și-a încheiat execuția.
- Un **Task C#** reprezintă o operație concurentă care poate fi (sau nu) executată folosind threaduri.
  - Taskurile pot fi compuse.
  - Pot folosi un container de threaduri pentru a reduce timpul necesar pornirii execuției.

- Tipul **`Task`** a fost introdus începând cu Framework 4.0 ca făcând parte din biblioteca pentru programare paralelă.

- **`System.Threading.Tasks`** namespace.

Pornirea execuției unui Task

- Framework 4.5 - Metoda statică **`Task.Run`** (pornirea execuției unui task folosind threaduri) - parametru de tip **`Action`** delegate:

```
Task.Run (() => Console.WriteLine ("Ana"));
```

- Framework 4.0 - Metoda statică **`Task.Factory.StartNew:`**

```
Task.Factory.StartNew(() =>Console.WriteLine("Ana"));
```

- Implicit, taskurile folosesc threaduri din containere deja create.
- Folosirea metodei **`Task.Run`** - similară cu execuția explicită folosind threaduri:

```
new Thread (() => Console.WriteLine ("Ana")).Start();
```

- **`Task.Run`** returnează un obiect **`Task`** care poate fi folosit pentru monitorizarea progresului.
- Nu este necesară apelarea metodei **`Start`**.

Obținerea rezultatului execuției

- **`Task<TResult>`** subclasă a clasei **`Task`** care permite returnarea rezultatului execuției.
- **`Task<TResult>`** poate fi obținut apelând **`Task.Run`** folosind un delegate de tip **`Func<TResult>`** (sau o expresie lambda compatibilă).
- Rezultatul poate fi obținut folosind proprietatea **`Result`**.
- Dacă taskul nu și-a încheiat execuția, apelul proprietății Result va bloca execuția threadului curent până la terminarea execuției taskului:

```
Task<int> task = Task.Run(()=>{int x=2; return 2*x; });
int result = task.Result;   // Blocks if not already finished
 Console.WriteLine (result);    // 4
```

Taskuri și excepții

- Taskurile propagă excepțiile (threadurile nu).
- Dacă un task aruncă o excepție, excepția este rearuncată către codul care apelează metoda **`Wait()`** a clasei Task sau care apelează proprietatea **`Result`** a clasei **`Task<TResult>`**:
- Excepția va fi inclusă într-o excepție de tip **`AggregateException`** de către CLR:

```
// Start a Task that throws a NullReferenceException:
Task task = Task.Run (() => { throw null; });
try{
  task.Wait();
}catch (AggregateException aex){
  if (aex.InnerException is NullReferenceException)
              Console.WriteLine ("Null!");
    else throw;
}
```

Actualizare GUI

- Interfețele grafice de tip Windows Forms folosesc obiecte de tip Control.
- Aceste obiecte pot fi modificate (actualizate, șterse, etc) doar de threadul care le-a creat.
- Nerespectarea acestei reguli are rezultate neașteptate sau aruncă excepții.
- Dacă se dorește apelarea unui membru (metoda, atribut, proprietate) a obiectului X creat într-un thread Y, cererea trebuie transmisă threadului Y (folosind metoda **`Invoke`** sau **`BeginInvoke`** a obiectului X).

Actualizare GUI

- **`Invoke`** și **`BeginInvoke`** au un parametru de tip delegate care referă metoda corespunzătoare obiectului de tip Control care se dorește a se executa.
  - **`Invoke`** execuție sincronă: execuția apelantului este blocată până la actualizarea controlului.
  - **`BeginInvoke`** execuție asincronă: execuția apelantului continuă, iar cererea este pusă într-o coadă (corespunzătoare evenimentelor de la tastatură, mouse, etc) și se va executa ulterior.

Exemplu

```
//1. definirea unei metode pentru actualizarea unui ListBox
private void updateListBox(ListBox listBox, IList<String> newData){
    listBox.DataSource = null;
    listBox.DataSource = newData;
 }
```

```
//2. definirea unui  delegate care va fi apelat de GUI Thread
public delegate void UpdateListBoxCallback(ListBox list, IList<String>
data);


//3. în celălalt thread se transmite metoda care actualizeaza ListBox:
list.Invoke(new UpdateListBoxCallback(this.updateListBox), new Object[]
{list, data});
or
list.BeginInvoke(new UpdateListBoxCallback(this.updateListBox), new
Object[]{list, data});
```

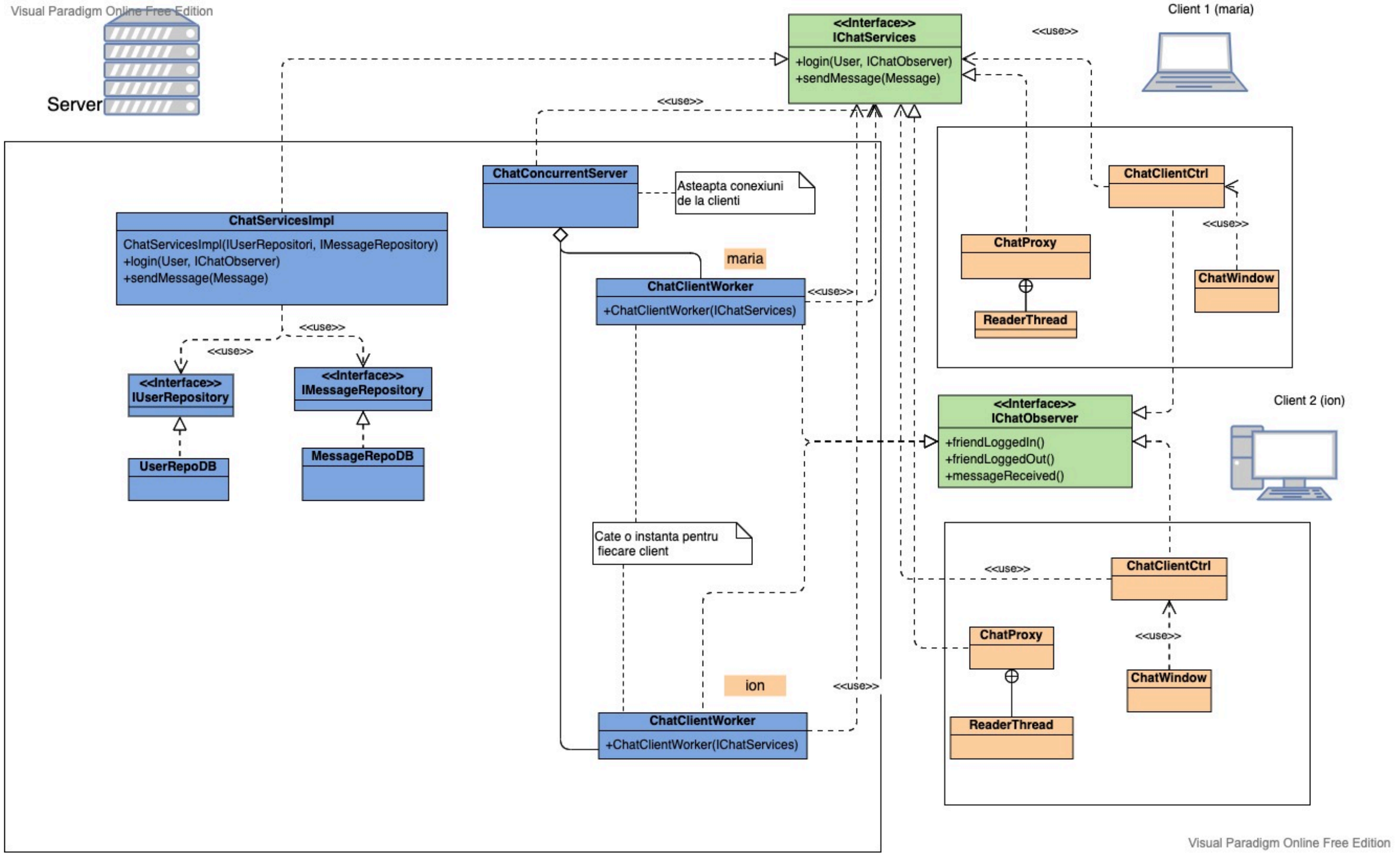
Exemplu C#

- O aplicație simplă client/server:
  - Serverul așteaptă conexiuni.
  - Clientul se conectează la server și îi trimite un text.
  - Serverul returnează textul scris cu litere mari, la care adaugă data și ora la care a fost primit textul.

Exemplu C#

- Mini-chat C#

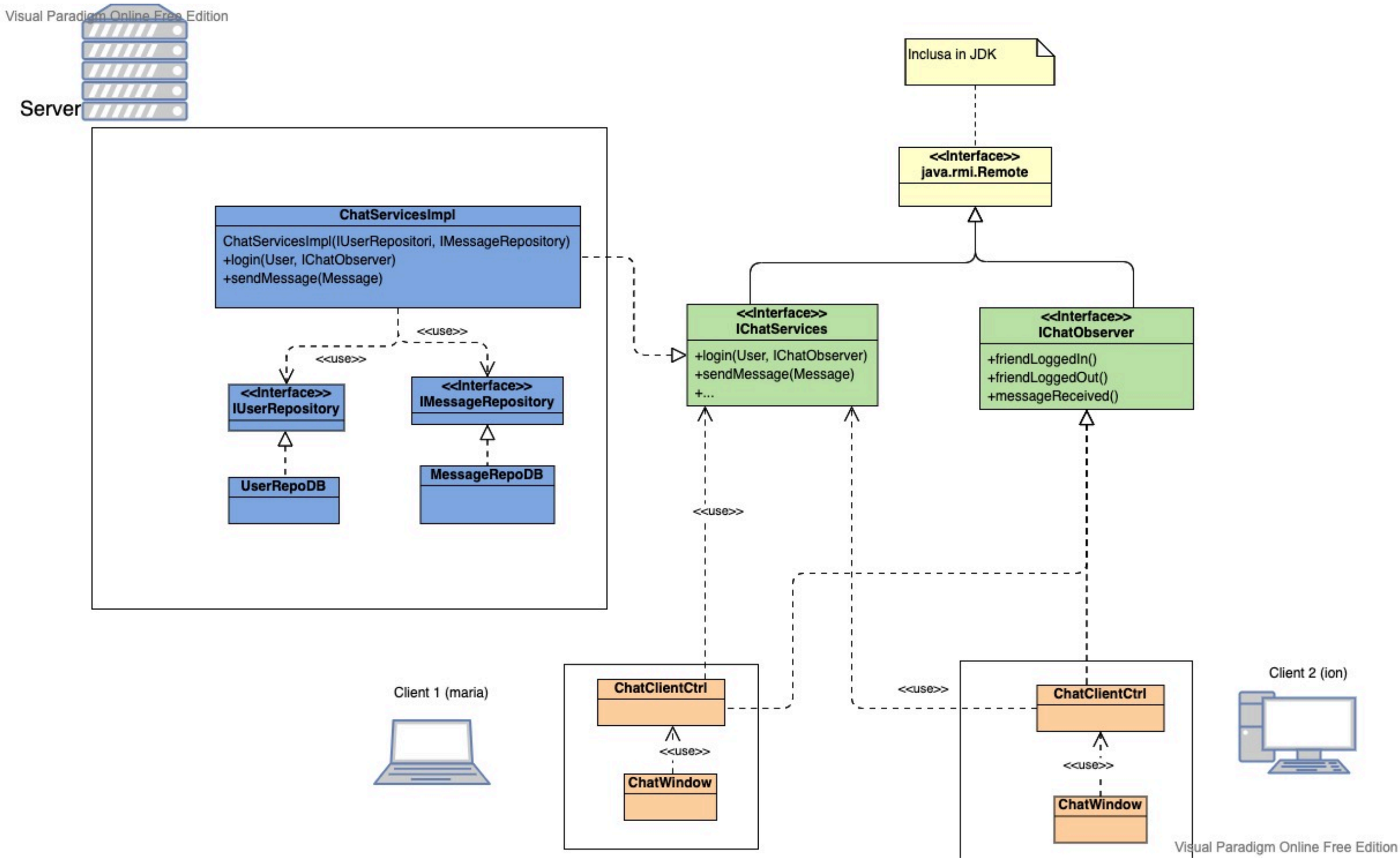
Networking


Visual Paradigm Online Free Edition
Server
<<Interface>>
IChatServices
+login(User, IChatObserver)
+sendMessage(Message)
<<use>>
Client 1 (maria)
ChatConcurrentServer
Asteapta conexiuni
de la clienti
ChatServicesImpl
ChatServicesImpl(IUserRepositori, IMessageRepository)
+login(User, IChatObserver)
+sendMessage(Message)
maria
ChatClientWorker
+ChatClientWorker(IChatServices)
ChatClientCtrl
ChatProxy
ReaderThread
ChatWindow
<<Interface>>
IUserRepository
<<Interface>>
IMessageRepository
UserRepoDB
MessageRepoDB
<<Interface>>
IChatObserver
+friendLoggedIn()
+friendLoggedOut()
+messageReceived()
Client 2 (ion)
Cate o instanta pentru
fiecare client
ion
ChatClientWorker
+ChatClientWorker(IChatServices)
ChatClientCtrl
ChatProxy
ChatWindow
ReaderThread
Visual Paradigm Online Free Edition

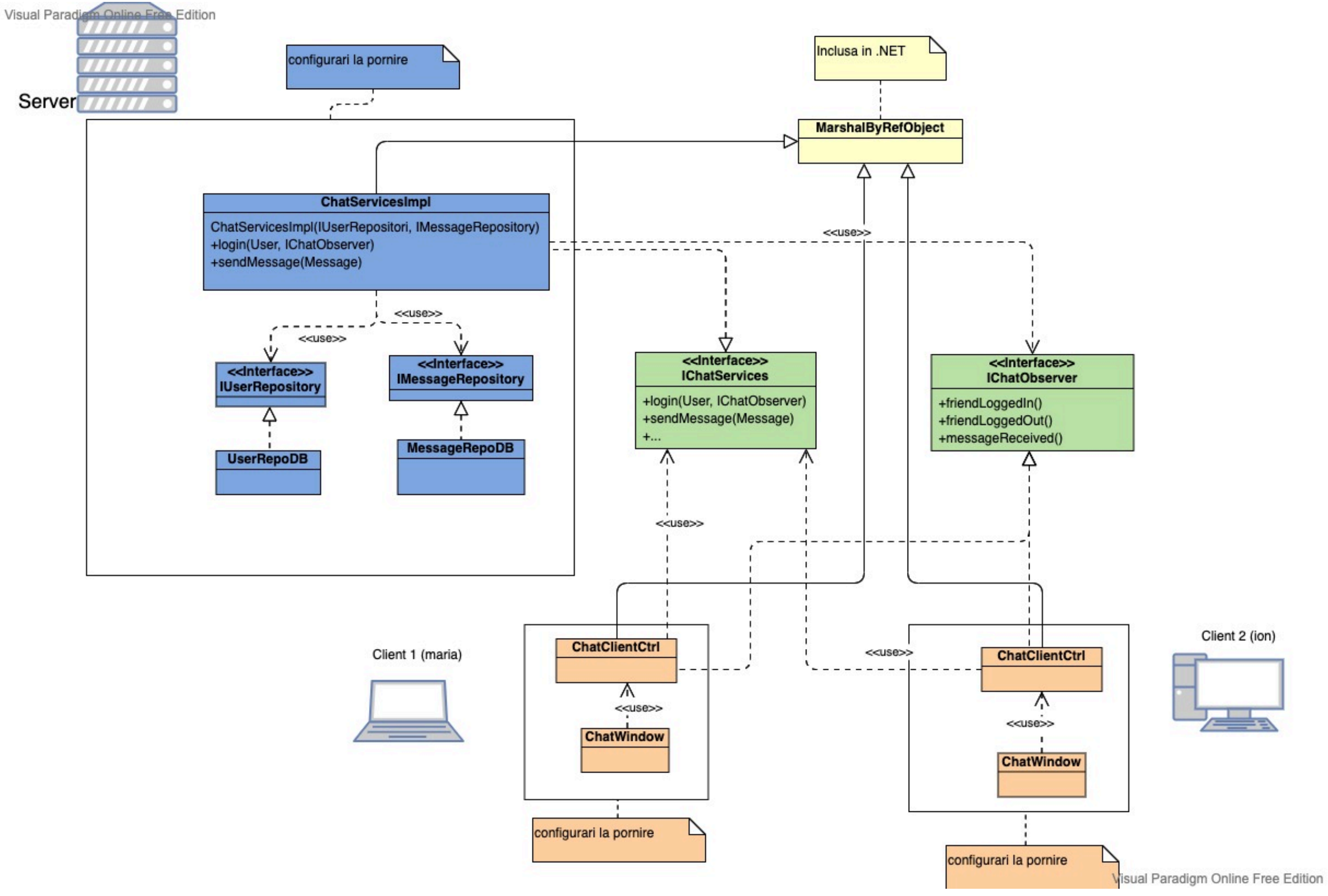
Remote Procedure Call

- ***Apelul procedurilor la distanță*** (eng. *Remote procedure call - RPC*) este o ***tehnologie de comunicare între procese*** care permite unei aplicații să inițieze execuția unei subrutine sau a unei proceduri în alt spațiu de adrese, ***fără ca programatorul să scrie explicit codul corespunzător interacțiunii dintre procese***.
- Programatorul scrie aproximativ același cod indiferent dacă apelează o rutină locală (din același proces) sau una la distanță (din alt proces, în alt spațiu de adrese).
- Când se folosește paradigma orientată pe obiecte RPC - apeluri la distanță sau apelul metodelor la distanță.
- RPC este adesea folosit pentru implementarea aplicațiilor client-server.
- Un apel la distanță este inițiat de client prin trimiterea unei cereri către un server la distanță pentru execuția unei proceduri cu parametrii dați. Serverul trimite un răspuns clientului, iar aplicația își continuă execuția.
- Cât timp serverul procesează cererea clientului, execuția clientului este blocată (așteaptă până când serverul a terminat procesarea cererii).

Java RMI


Visual Paradigm Online Free Edition
Server
ChatServicesImpl
ChatServicesImpl(IUserRepositori, IMessageRepository)
+login(User, IChatObserver)
+sendMessage(Message)
<<use>>
<<use>>
<<Interface>>
IUserRepository
<<Interface>>
IMessageRepository
UserRepoDB
MessageRepoDB
Inclusa in JDK
<<Interface>>
java.rmi.Remote
<<Interface>>
IChatServices
+login(User, IChatObserver)
+sendMessage(Message)
+...
<<Interface>>
IChatObserver
+friendLoggedIn()
+friendLoggedOut()
+messageReceived()
<<use>>
Client 1 (maria)
ChatClientCtrl
<<use>>
ChatWindow
<<use>>
ChatClientCtrl
<<use>>
ChatWindow
Client 2 (ion)
Visual Paradigm Online Free Edition

NET. Remoting



Remote Procedure Call
Java RMI, NET. Remoting

- **Avantaje**
  - Dezvoltarea ușoară a aplicațiilor client-server
  - Ascunderea detaliilor de implementare pentru comunicarea client-server.
- **Dezavantaje**
  - Aplicațiile se pot dezvolta într-un singur limbaj (Java/C#/etc)
  - Dacă apar versiuni noi pentru interfețe (servicii), trebuie modificați toți clienții.

Aplicații distribuite cross-platform

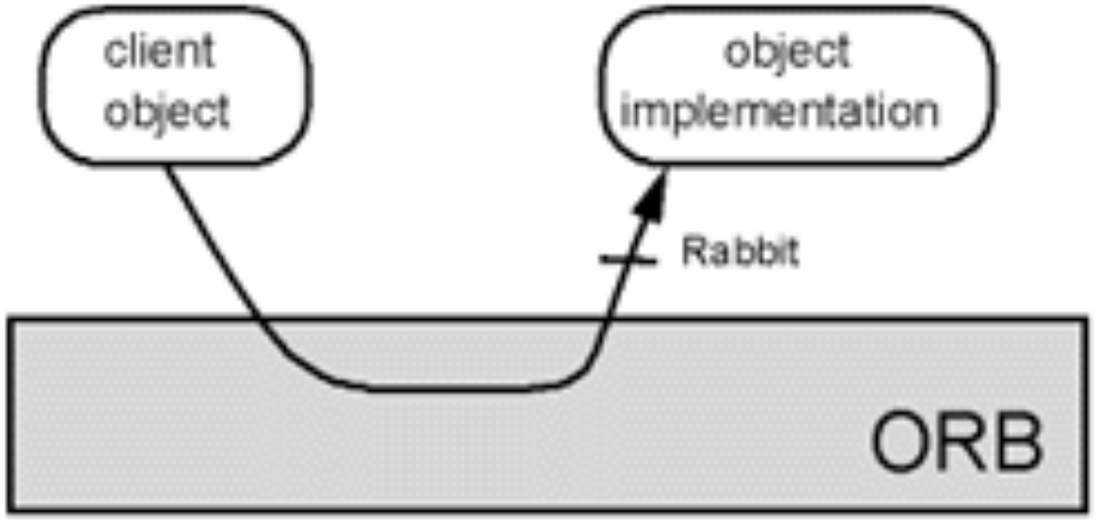
- CORBA
- Protobuf
- gRPC
- Thrift

CORBA

- A fost propusă de OMG (Object Management Group) înainte de anul 2000.
- CORBA (Common Object Request Broker Architecture) este o arhitectură standard pentru sisteme distribuite orientate pe obiect.
- Produsele CORBA oferă un framework pentru dezvoltarea și execuția aplicațiilor distribuite.
- Permite unei colecții de obiecte distribuite ***heterogene*** să interacționeze.
- CORBA definește arhitectura obiectelor distribuite.
- Paradigma de bază CORBA este de cerere a unor servicii oferite de un obiect distribuit (remote).
- Serviciile oferite de un obiect sunt specificate de interfața acestuia.
- ***Interfețele*** sunt definite folosind limbajul Interface Definition Language (IDL) definit de OMG.
- Obiectele distribuite (remote) sunt identificate prin referințe, de tipul interfețelor IDL.

Arhitectura CORBA

- Un client păstrează o referință către un obiect distribuit (remote).
- Un proces numit Object Request Broker (ORB), trimite cererea obiectului și returnează rezultatul primit clientului.




```
interface Rabbit {
        :
    AnotherObject jump(
        in long how_high);
        :
};
```

Interfețe IDL

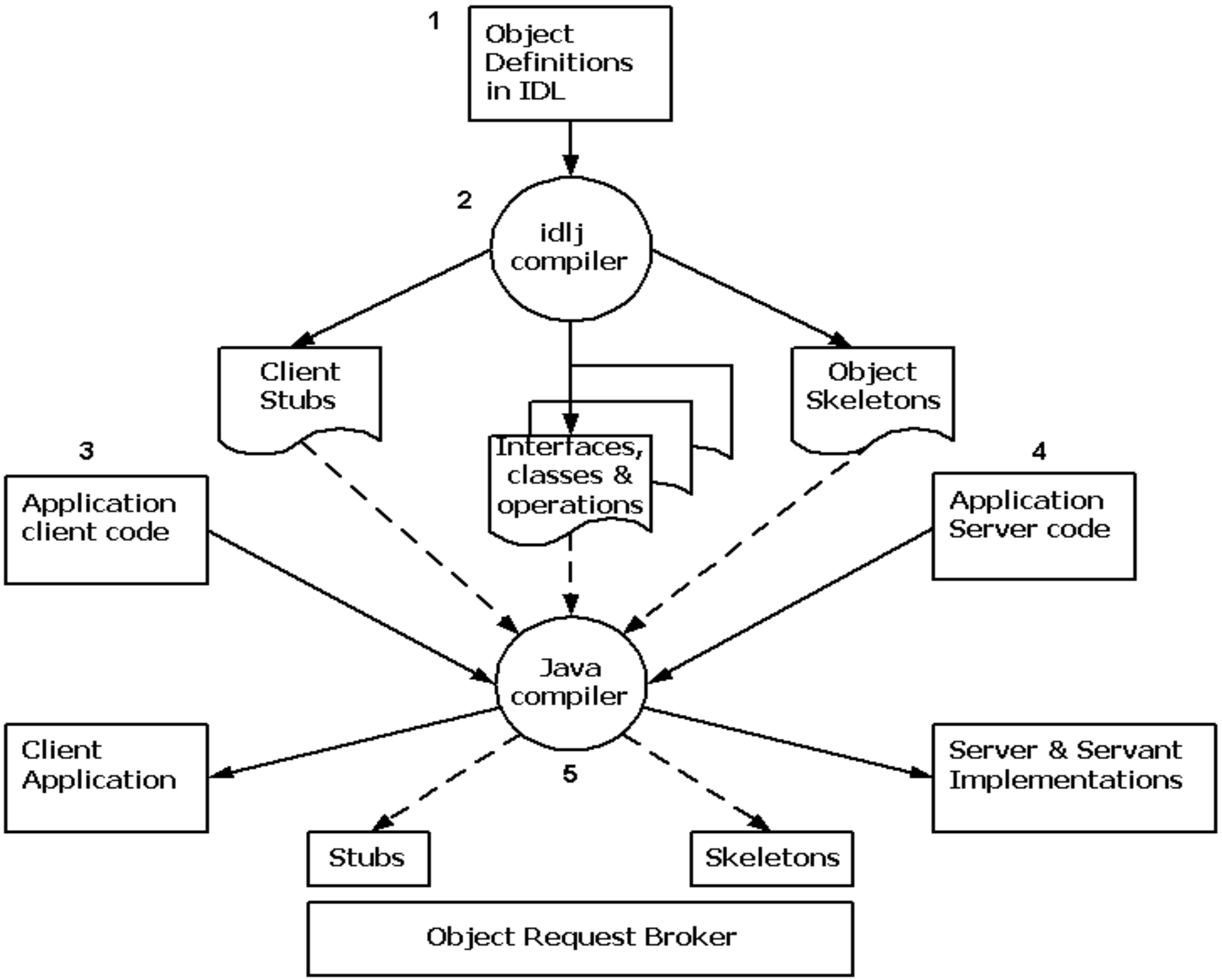
- Limbajul *Interface Definition Language* (IDL) propus de OMG permite specificarea interfețelor obiectelor remote.
- **Interfața** unui obiect remote **indică metodele** ce pot fi apelate la distanță, dar nu cum sunt implementate aceste metode.
- În IDL nu se pot declara starea unui obiect si algoritmi.
- Implementarea unui obiect CORBA este furnizată într-un limbaj de programare (Java, C++, etc). O interfață specifică contractul dintre codul care folosește obiectul și codul care implementează obiectul. Clienții depind doar de interfață.
- Interfețele IDL nu depind de un anumit limbaj de programare. IDL definește transformări (eng. *language bindings*) pentru diferite limbaje de programare.
- Se permite astfel ca pentru obiectul remote să fie ales cel mai potrivit limbaj de programare, și, de asemenea, permite clienților să aleagă cel mai potrivit limbaj (care poate diferi de cel folosit pentru implementarea obiectului remote).
- Transformări pentru C, C++, Java, Ada, Smalltalk, etc.

Exemplu IDL

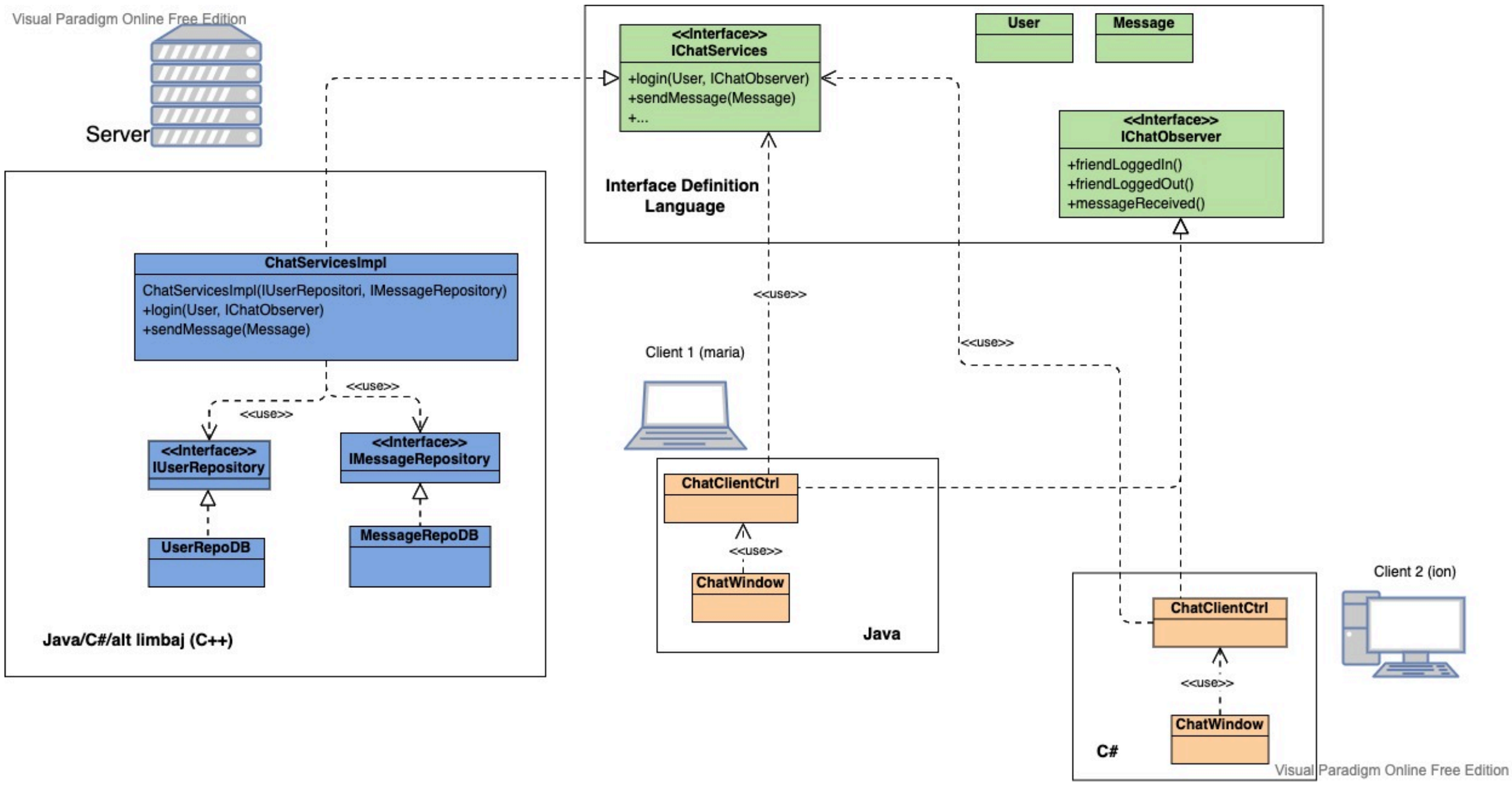
```
module BookShop {
  struct Book {
      double price;
      String title;
      String author
  };
  exception Unknown{};
  interface Library {
      Book getBook(in string title) raises(Unknown);
      readonly attribute string name;


  };
  interface BookFactory {
    Book create_book( in string title, in string author, in double
   price);
  };
};
```

CORBA și Java



CORBA și Java/C#/C++



Protocol Buffers (Protobuf)

- Reprezintă o modalitate independentă de limbaj și platformă de a serializa structuri de date folosite în protocoale de comunicație, stocarea datelor, etc.
- Sunt flexible, eficiente și au un mecanism automat de serializare a datelor structurate (similar cu XML), dar mecanismul este mai rapid, mai simplu și mai ușor de folosit.
- Se definește o singura dată modul de structurare a datelor (folosind un IDL), iar apoi se folosește un compilator special (*protoc*) care generează cod ce permite scrierea/citirea structurilor de date în/din o varietate de streamuri de date și folosind diferite limbaje de programare.
- Este permisă modificarea structurii datelor fără a provoca apariția erorilor în programele dezvoltate folosind vechea structură de date.
- Specificarea structurii datelor se definește folosind tipuri de mesaje protocol buffers și sunt salvate în fișiere **`.proto`**.
- Fiecare mesaj protocol buffer este o înregistrare logică mică de informații, conținând o serie de perechi cheie-valoare.

Protocol Buffers vs XML

- Protocol buffers au avantaje asupra XML în privința serializării datelor:
  - sunt mai simple
  - sunt de 3 până la de 10 ori mai mici ca și dimensiune
  - sunt de 20 până la de 100 de ori mai rapide
  - sunt mai puțin neclare (ambigue)
  - clasele generate automat pentru accesarea datelor sunt mai ușor de folosit în limbajul de programare
- Protocol buffers nu sunt totdeauna o soluție mai bună decât XML:
  - Protocol buffers nu sunt potrivite pentru modelarea unui text ce folosește markup (ex. HTML), deoarece nu permite ușor imbricarea structurii datelor cu text.
  - XML este ușor de citit de oameni (eng. *human-readable*) și de editat (eng. *human-editable*);
  - Protocol buffers nu pot fi citite de oameni și editate
  - XML se descrie pe sine.
  - Un protocol buffer are sens doar dacă avem acces și la definiția mesajului (fișierul .proto).

Protocol Buffers vs XML

```
//XML

<person>

    <name>John Doe</name>

    <email>jdoe@example.com</email>

</person>
```

```
//Reprezentarea textuala a unui  Protobuf (nu octeții serializați)

person {

  name: "John Doe"

  email: "jdoe@example.com"

}
```

Protocol Buffers vs XML

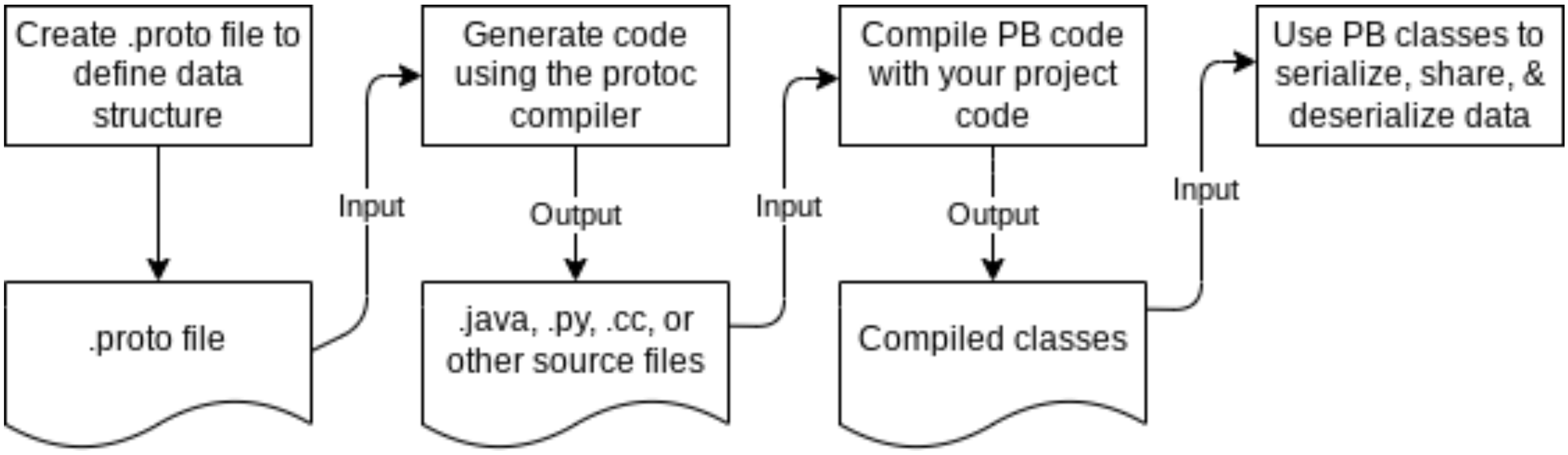
//Parsarea unui XML

```
<person>
    <name>John Doe</name>
    <email>jdoe@example.com</email>
  </person>
//Java
 person.getElementsByTagName("name").item(0).getNodeValue()
 person.getElementsByTagName("email").item(0).getNodeValue();
```

//Parsarea unui mesaj Protobuf

```
person {
  name: "John Doe"
  email: "jdoe@example.com"
}
System.out.println("Name: " + person.getName());
System.out.println("E-mail: "+ person.getEmail());
```

Protocol Buffers Workflow*



*https://developers.google.com/protocol-buffers/docs/overview

Protocol Buffers vers. 2 – Exemplu

```
syntax="proto2"
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;
  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }
  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }
  repeated PhoneNumber phone = 4;
}
```

Definirea unui nou tip de mesaj - vers. 3

- Tipurile mesajelor sunt salvate într-un fișier `.proto`

```
syntax="proto3";
message SearchRequest {
  string query = 1;
  int32 page_number = 2;
  int32 result_per_page = 3;
}
```

- Fiecare câmp are un nume, tip și un tag asociat.
- ***Tipul*** poate fi:
  - scalar (double, float, int32, int64, bool, string, bytes, etc).
  - tip compus (enumerare)
  - tipul unui alt mesaj

Definirea unei enumerări - vers. 3

- ***Enumerare***: un câmp de tip enumerare poate avea ca și valoare doar una dintre constantele specificate.
- Fiecare enumerare trebuie să definească o constantă care are asociată valoarea zero ca primă constantă din enumerare.
- Aceasta valoare va fi folosită ca și valoarea implicită.
- Două constante pot avea aceeași valoare (dacă se setează opțiunea allow_alias).

```
enum Corpus {
     option allow_alias = true;
    UNIVERSAL = 0;
    WEB = 1;
    IMAGES = 2;
    LOCAL = 3;
    NEWS = 4;
    PRODUCTS = 5;
    VIDEO = 5;
  }
  Corpus corpus = 4;
```

Reguli pentru specificarea câmpurilor - vers. 3

- Un câmp poate fi:
  - *singular*: într-un mesaj bine format acest câmp poate să apară cel mult o dată (zero sau unu).
  - **`repeated`**: acest câmp poate să apară de ori câte ori (inclusiv zero) într-un mesaj bine format. Ordinea valorilor care se repetă se va păstra.
- Dacă un mesaj nu conține nici o valoare pentru un câmp singular, la parsare câmpul va primi valoarea implicită corespunzătoare tipului său.

```
message Result {
  string url = 1;
  string title = 2;
  repeated string snippets = 3;
}
```

Atribuirea tagurilor

- Fiecare câmp din definiția unui mesaj are un tag numeric unic asociat.
- Aceste taguri sunt folosite pentru identificarea câmpurilor în formatul binar și nu ar trebui schimbate după începerea folosirii lui.
- Tagurile cu valori între 1 și 15 ocupă un octet (incluzând numărul de identificare și tipul câmpului).
- Tagurile cu valori între 16 - 2047 ocupă 2 octeți.
- Tagurile cu valori între 1 și 15 ar trebui folosite pentru câmpurile folosite cele mai des din mesaj.
- Tagurile pot avea valori între 1 și 536,870,911.
- Numerele între 19000 și 19999 sunt rezervate pentru implementarea Protocol Buffers - compilatorul protoc va genera o eroare dacă se încearcă folosirea lor.

Tipuri multiple de mesaje

- Mai multe tipuri de mesaje pot fi definite într-un singur fișier .proto.

```
message SearchRequest {
  string query = 1;
  int32 page_number = 2;
  int32 result_per_page = 3;
}


message SearchResponse {
 ...
}
```

Câmpuri rezervate

- Dacă definiția tipului unui mesaj se modifică ulterior prin ștergerea sau comentarea unui câmp, utilizatorii pot refolosi tag-ul asociat câmpului respectiv.
- Pot apărea erori când se folosesc mesaje salvate cu versiuni mai vechi ale aceluiași tip de mesaj.
- Se poate specifica că anumite taguri sau nume de câmpuri sunt rezervate și nu mai pot fi refolosite ulterior, folosind instrucțiunea **`reserved`**.
- În aceeași instrucțiune **`reserved`** nu pot fi folosite taguri și nume de câmpuri.

```
message Foo {
    int32 foo = 2;  // câmpul foo

  reserved 2, 15, 9 to 11;
  reserved "foo", "bar";
}
```

Fișiere .proto multiple

- Într-un fișier .proto se pot folosi definițiile din alte fișiere .proto prin importarea lor.
- Importarea definițiilor din alt fișier **.proto** se face cu instrucțiunea import la începutul fișierului (după instrucțiunea syntax):

```
import “myproject/other_protos.proto“;
```

- Compilatorul caută fișierele importate în lista de directoare specificată la linia de comandă ca și parametru al compilatorului folosind opțiunea **-I sau --proto_path**.
- Dacă nu a fost specificată această opțiune, compilatorul va căuta in directorul în care a fost rulat compilatorul.

Tipuri Nested

- Se pot defini tipuri de mesaje în interiorul altor tipuri de mesaje:

```
message SearchResponse {
  message Result {
     string url = 1;
     string title = 2;
     repeated string snippets = 3;
  }
  repeated Result result = 1;
}
```

- Tipul nested se poate folosi din exterior respectând formatul **`Parent.Type:`**

```
message SomeOtherMessage {
   SearchResponse.Result result = 1;
}
```

Tipul Any

- Tipul Any este folosit pentru a defini câmpuri pentru care nu știm/avem fișierul .proto corespunzător.
- Un câmp de tip Any va conține un mesaj necunoscut serializat într-un șir de octeți și un URL folosit ca și identificator global unic care va indica către definiția tipului mesajului.
- Pentru a folosi tipul Any trebuie importat fișierul *google/protobuf/any.proto*.

```
import "google/protobuf/any.proto";

message ErrorStatus {
  string message = 1;
  repeated google.protobuf.Any details = 2;
}
```

URL implicit asociat unui mesaj definit într-un fișier .proto are formatul

```
type.googleapis.com/packagename.messagename
```

Opțiunea oneof

- Este folosită când valoarea unui câmp poate fi cel mult una dintr-o mulțime finită de valori de tipuri diferite. Folosirea opțiunii reduce dimensiunea mesajului.
- Câmpurile Oneof sunt câmpuri normale, exceptând faptul că partajează aceeași zonă de memorie și cel mult un câmp poate avea asociată o valoare la un moment dat.
- Setarea valorii unuia dintre câmpurile marcate *oneof* duce automat la ștergerea valorii celorlalte câmpuri.
- Se poate determina care câmp din mulțimea specificată oneof are asociată o valoare (dacă există un astfel de câmp). (Dependent de limbaj)
- Nu se pot folosi câmpuri repeated în mulțimea câmpurilor specificate cu oneof.

```
message SampleMessage {
  oneof test_oneof {
    string name = 4;
    SubMessage sub_message = 9;
  }
}
```

Tipul Map

- Se pot defini dicționare ca și câmp al unui nou tip de mesaj.

```
map<key_type, value_type> map_field = N;
```

- **`key_type`** poate fi orice tip *integral* (orice tip scalar exceptând tipurile reale și octeți) sau tipul string. **`value_type`** poate fi orice tip.
- Observații:
  - Câmpurile Map nu pot fi *repeated*.
  - Ordinea serializarii valorilor dintr-un dicționar este nespecificată, nu ne putem baza pe o anumită ordine a elementelor din dicționar.
  - Dacă la parsare/deserializare există mai multe chei având aceeași valoare se păstrează ultima valoare întâlnită.

```
map<string, Project> projects = 3;
```