Medii de proiectare și programare

2023-2024
Curs 6

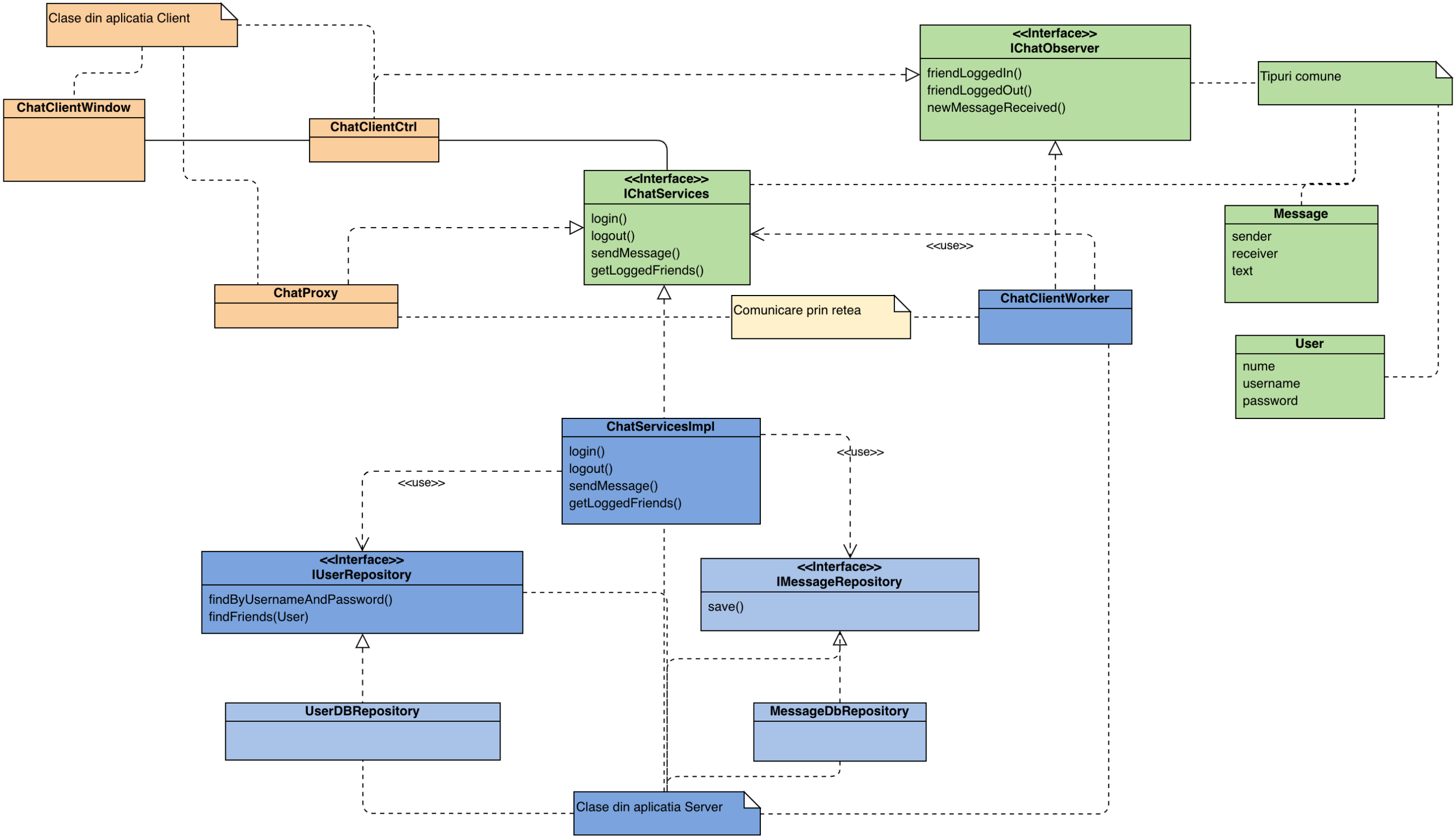
Conținut curs 6

- Exemplu Mini-Chat (networking - Java)
- Networking si threading in C#
- Exemplu Mini-Chat (networking - C#)

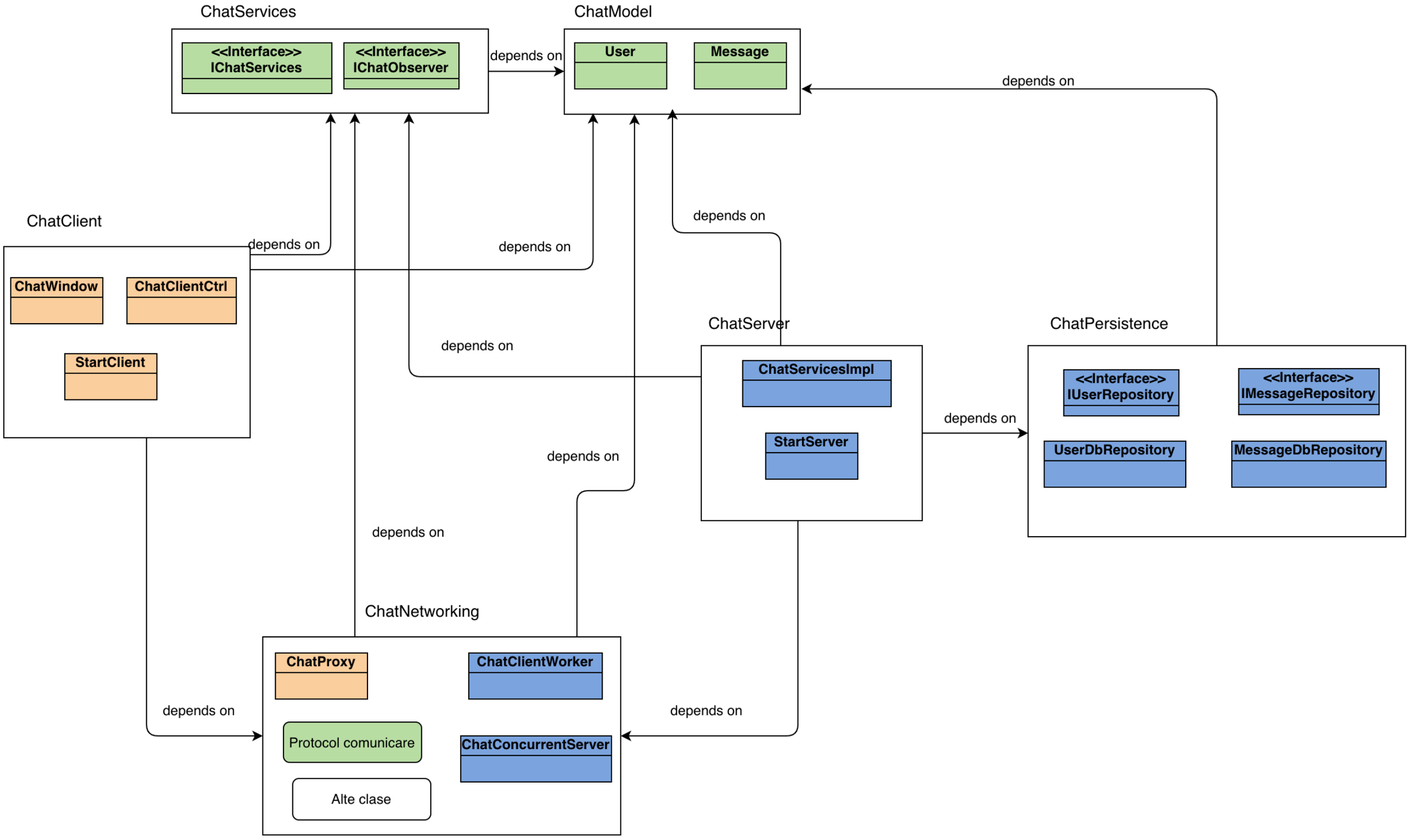
Mini-Chat

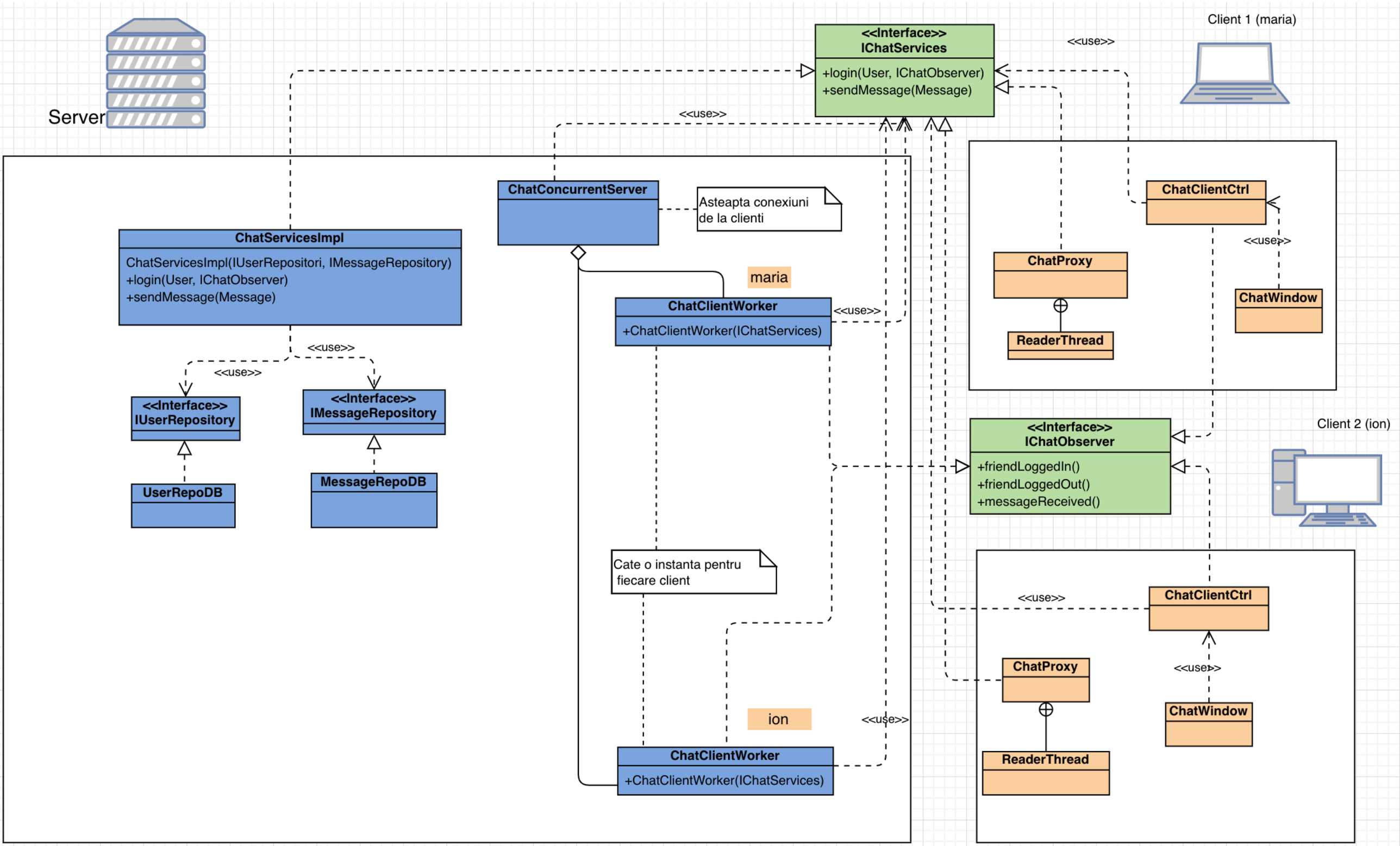
- Proiectare (diagrame)

- Implementare Java

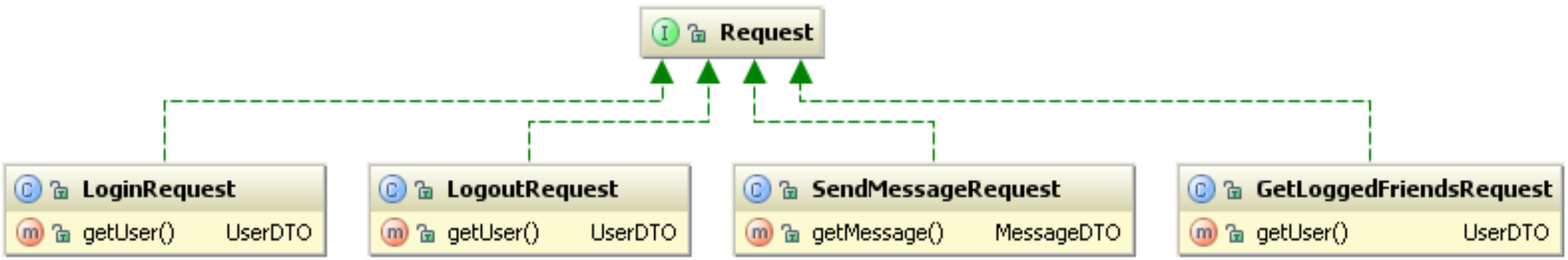

Clase din aplicatia Client
ChatClientWindow
ChatClientCtrl
<<Interface>>
IChatObserver
friendLoggedIn()
friendLoggedOut()
newMessageReceived()
Tipuri comune
<<Interface>>
IChatServices
login()
logout()
sendMessage()
getLoggedFriends()
<<use>>
Message
sender
receiver
text
ChatProxy
Comunicare prin retea
ChatClientWorker
User
nume
username
password
ChatServicesImpl
login()
logout()
sendMessage()
getLoggedFriends()
<<use>>
<<use>>
<<Interface>>
IUserRepository
findByUsernameAndPassword()
findFriends(User)
<<Interface>>
IMessageRepository
save()
UserDBRepository
MessageDbRepository
Clase din aplicatia Server

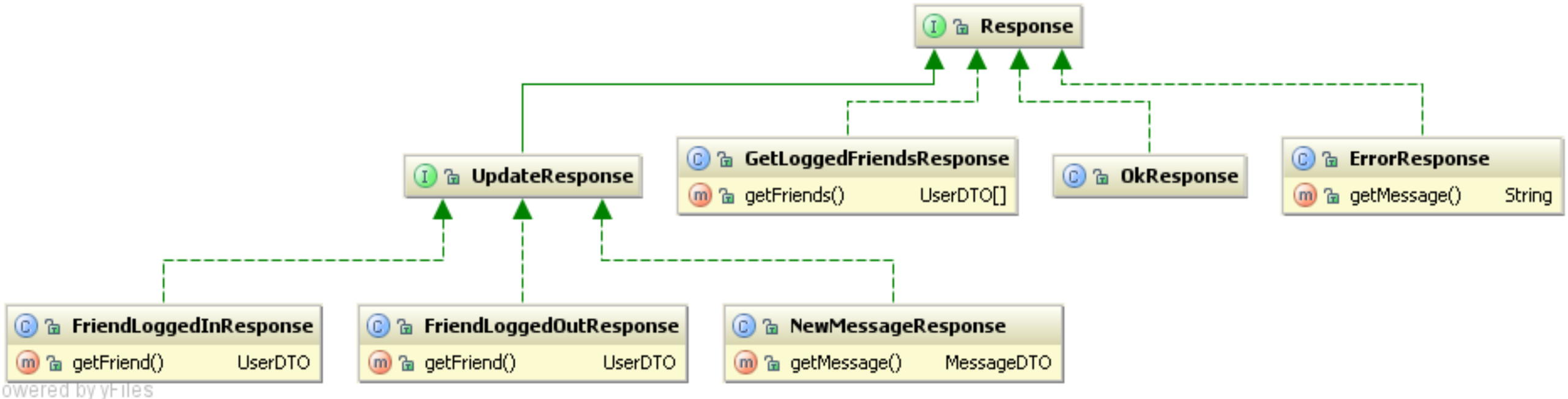
Proiecte


ChatServices
<<Interface>>
IChatServices
<<Interface>>
IChatObserver
depends on
ChatModel
User
Message
depends on
ChatClient
depends on
depends on
depends on
ChatWindow
ChatClientCtrl
StartClient
ChatServer
ChatPersistence
depends on
ChatServicesImpl
StartServer
depends on
<<Interface>>
IUserRepository
<<Interface>>
IMessageRepository
UserDbRepository
MessageDbRepository
depends on
depends on
depends on
ChatNetworking
depends on
ChatProxy
ChatClientWorker
Protocol comunicare
ChatConcurrentServer
Alte clase

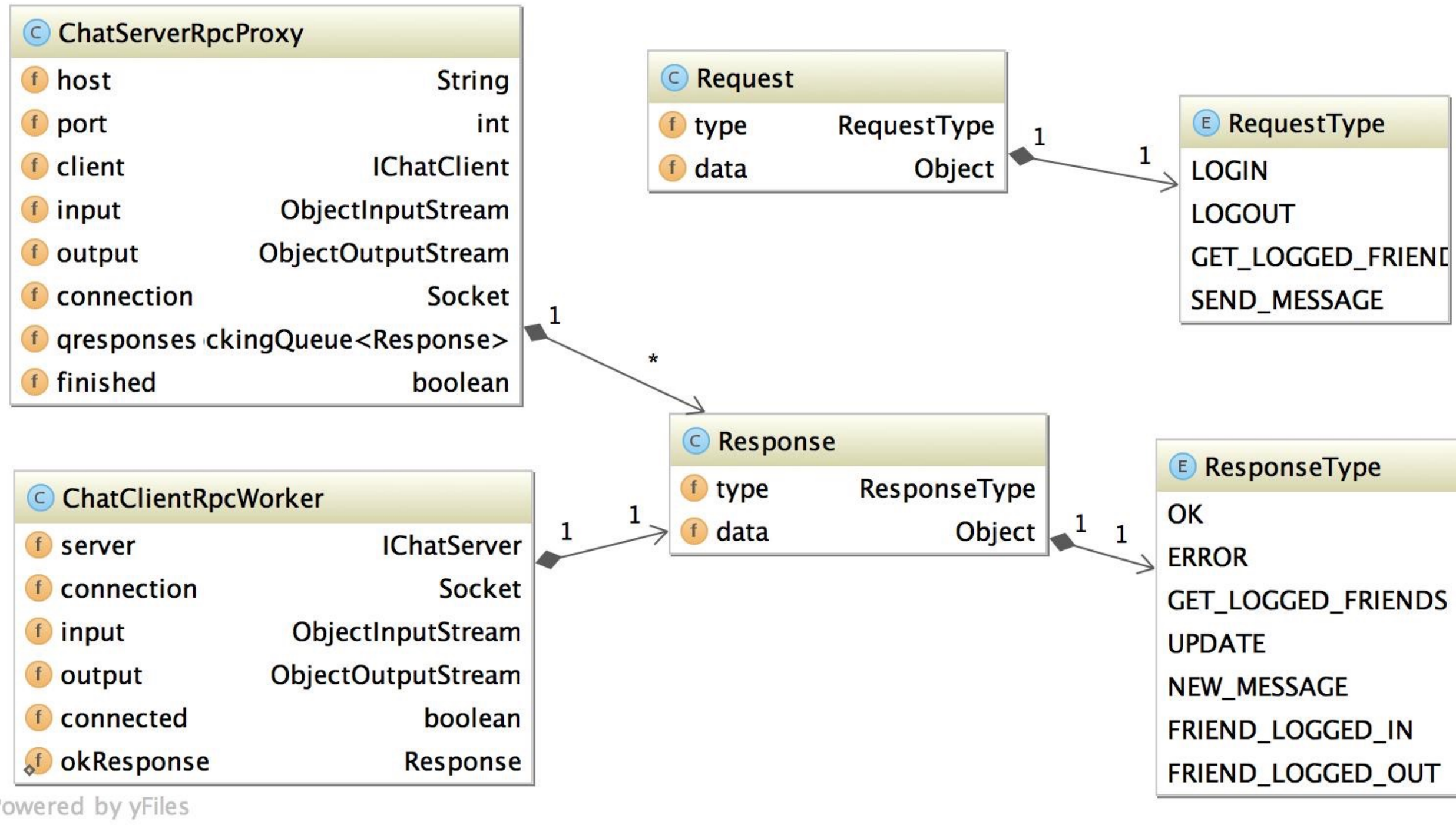

Server
<<Interface>>
IChatServices
+login(User, IChatObserver)
+sendMessage(Message)
<<use>>
Client 1 (maria)
<<use>>
ChatConcurrentServer
Asteapta conexiuni
de la clienti
ChatClientCtrl
ChatServicesImpl
ChatServicesImpl(IUserRepositori, IMessageRepository)
+login(User, IChatObserver)
+sendMessage(Message)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

Mini-chat Object Protocol





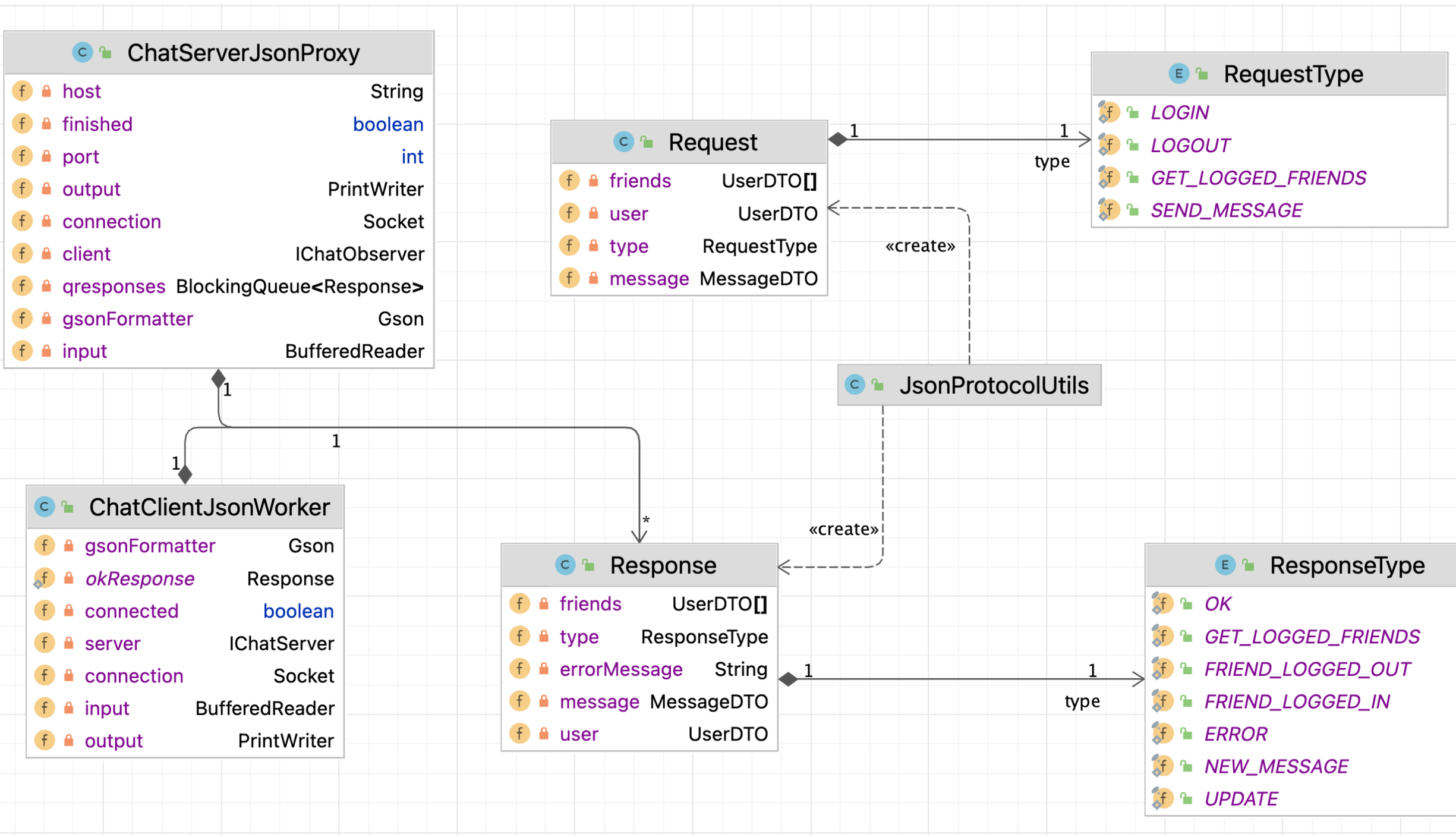
Mini-chat Rpc Protocol



Deserializarea binară - securitate

- Java și C#
- Vulnerabilități:
  - Denial of service (DoS)
  - Controlul accesului
  - Execuție de cod nedorit
- Alternative:
  - Serializare XML, Json, etc. (avantaje/dezavantaje)
- https://learn.microsoft.com/en-us/dotnet/standard/serialization/binaryformatter-security-guide
- https://medium.com/@AlexanderObregon/a-deep-dive-into-java-serialization-e514346ac2b2

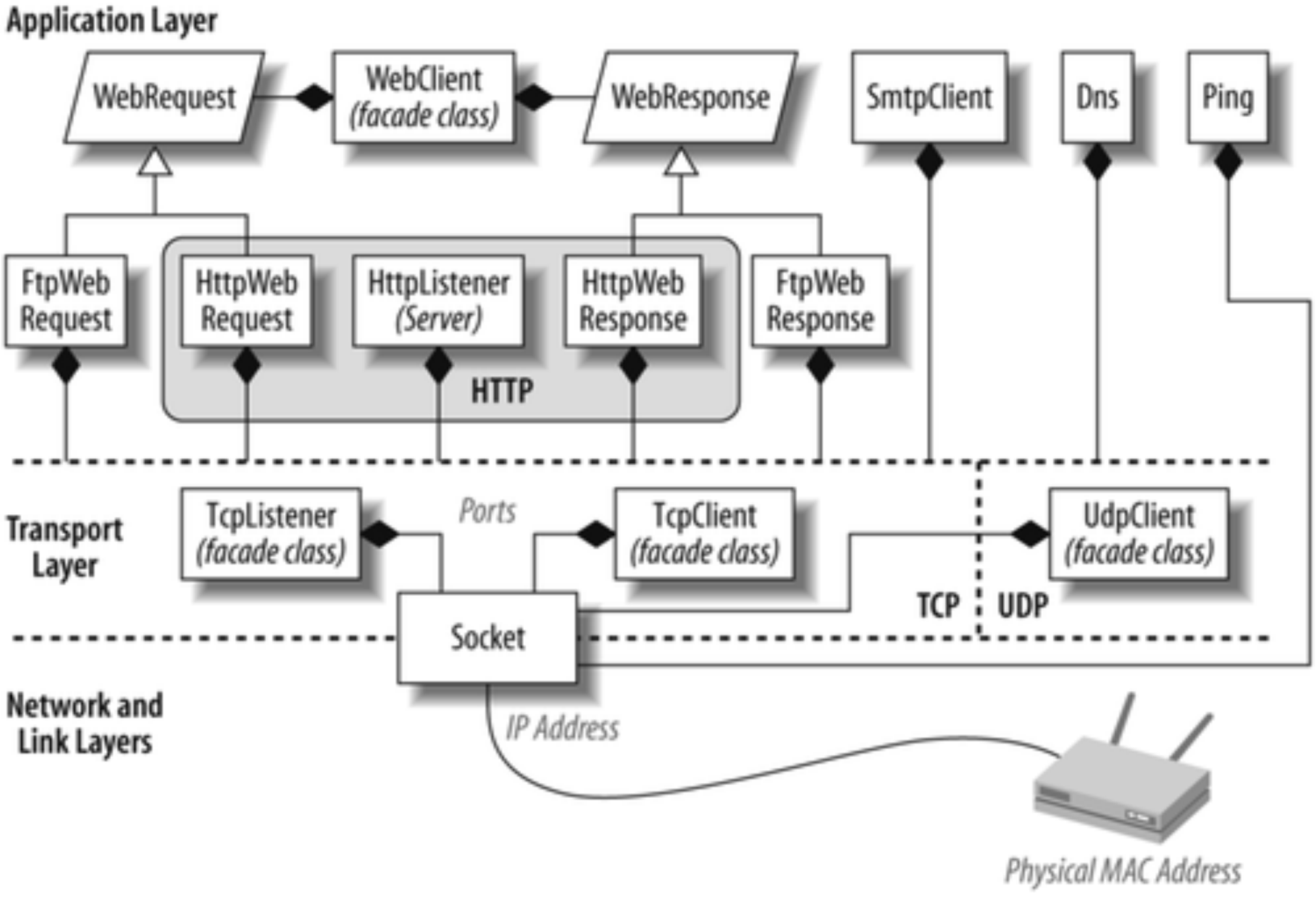
Mini-chat Json Protocol



Networking în C#

- .NET conține clase pentru comunicarea prin rețea folosind protocoale standard cum ar fi HTTP, TCP/IP și FTP.
- Spațiul de nume **`System.Net.*`**:
  - **`WebClient`** façade pentru operații simple de download/upload folosind HTTP sau FTP.
  - **`WebRequest`** și **`WebResponse`** pentru operații HTTP și FTP complexe.
  - **`HttpListener`** pentru implementarea unui HTTP server.
  - **`SmtpClient`** pentru construirea și trimiterea mesajelor folosind SMTP.
  - **`TcpClient`**, **`UdpClient`**, **`TcpListener`** și **`Socket`** pentru acces direct la nivelul rețea.

Networking în C#



Networking în C#

- Clasa **`IPAddress`** din **`System.Net`** reprezintă o adresă IPv4 (32 bits) sau IPv6 (128 bits).

```
IPAddress a1 = new IPAddress (new byte[] { 172, 30, 106, 5 });
IPAddress a2 = IPAddress.Parse ("172.30.106.5");
IPAddress a3 = IPAddress.Parse
   ("[3EA0:FFFF:198A:E4A3:4FF2:54fA:41BC:8D31]"); //IPv6
```

- O asociere între o adresă IP și un port este reprezentată fololosind clasa **`IPEndPoint`**:

```
IPAddress a = IPAddress.Parse ("172.30.106.5");
IPEndPoint ep = new IPEndPoint (a, 55555);              // Port 55555
Console.WriteLine (ep.ToString(  ));           // 172.30.106.5:55555
```

- Porturile: 1 – 65535.
- Porturile dintre 49152 și 65535 nu sunt rezervate oficial.

System.Net.Sockets Namespace

- Clasele **`TcpClient`**, **`TcpListener`** și **`UdpClient`** încapsulează detaliile creării conexiunilor de tip TCP și UDP.
- **`Socket`** implementează interfața Berkeley socket.
- **`SocketException`** excepția aruncată când apare o eroare la comunicarea prin socket.
- **`NetworkStream`** streamul folosit pentru comunicarea prin rețea.

TcpListener

- TCP server:

```
TcpListener listener = new TcpListener (<ip address>, port);
listener.Start();
while (keepProcessingRequests)
    using (TcpClient c = listener.AcceptTcpClient(  ))
        using (NetworkStream n = c.GetStream(  ))     {
        // Read and write to the network stream...
}
listener.Stop ();
```

- **`TcpListener`** necesită adresa IP la care va aștepta conexiunile clienților (dacă calculatorul are două sau mai multe plăci de rețea).
  - **`IPAddress.Any`** ascultă pe toate adresele IP locale (sau singura).
- **`AcceptTcpClient`** blochează execuția până când se conectează un client.

TcpClient

- Client Tcp:

```
using (TcpClient client = new TcpClient (<address>, port))
  using (NetworkStream n = client.GetStream(  ))
  {
   // Read and write to the network stream...
  }
```

- **`TcpClient`** încearcă crearea conexiunii în momentul creării obiectului folosind adresa IP și portul specificate.
- Constructorul blochează execuția pâna la stabilirea conexiunii.

NetworkStream

- **`NetworkStream`** comunicare ***bidirecțională*** pentru transmiterea și recepționarea datelor după stabilirea unei conexiuni.
- Methods:
  - **`Read`**
  - **`Close`**
  - **`Write`**
  - **`Seek`**
  - **`Flush`**
- Properties:
  - **`CanRead, CanWrite`**
  - **`Socket`**
  - **`DataAvailable`**
  - **`Length`**

Threading în C#

- Spatiul de nume **`System.Threading`** clase și interfețe pentru programarea concurentă:
  - Clasa **`Thread`**.
  - Delegate: **`ThreadStart`**, **`ParameterizedThreadStart`**.
  - Sincronizare: **`lock`**, **`Monitor`**, **`Mutex`**, **`Semaphore`**, **`EventHandles`**.

- **Delegates**: reprezintă metoda executată de un thread.

```
public delegate void ThreadStart();

public delegate void ParameterizedThreadStart(Object obj);
```

- Clasa **`Thread`**: crearea unui thread, setarea priorității, obținerea informațiilor despre statusul unui thread.

```
public Thread(ThreadStart start);
public Thread(ParameterizedThreadStart start);
```

Threading în C#

```
class Program   {
   static void Main(string[] args)  {
      Worker worker=new Worker();
      Thread t1=new Thread(new ParameterizedThreadStart(static_run));
      Thread t2=new Thread(new ThreadStart(worker.run));
      t1.Start("a");
      t2.Start();
   }
   static void static_run(Object data) {
      for(int i=0;i<26;i++)  { Console.Write("{0} ",data);   }
   }}
class Worker {
   public void run() {
     for(int i=0;i<26;i++)    Console.Write("{0} ",i);
   }}
//a a a a a a a a a a 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
   19 20 21 22 23 24 25 a a a a a a a a a a a a a a a a
```

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 începand cu Framework 4.0 ca facând 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
```