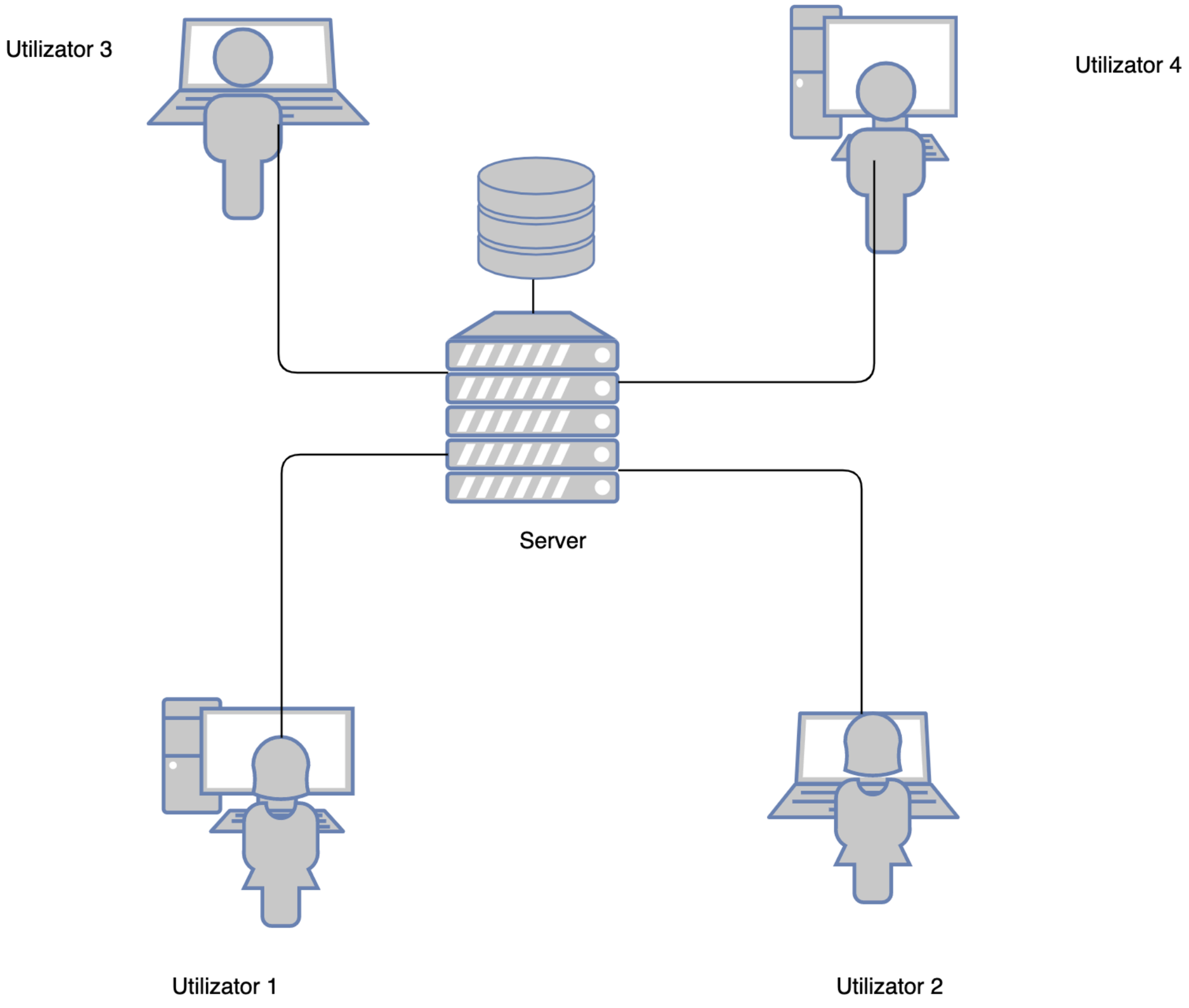
Medii de proiectare și programare

2023-2024
Curs 5

Conținut curs 5

- Aplicații client-server
  - Șablonul Proxy
  - Data transfer object
- Networking și threading în Java
  - Exemplu Mini-Chat
- Networking și threading în C# (curs 6)

Aplicații client-server


Utilizator 3
Utilizator 4
Server
Utilizator 1
Utilizator 2

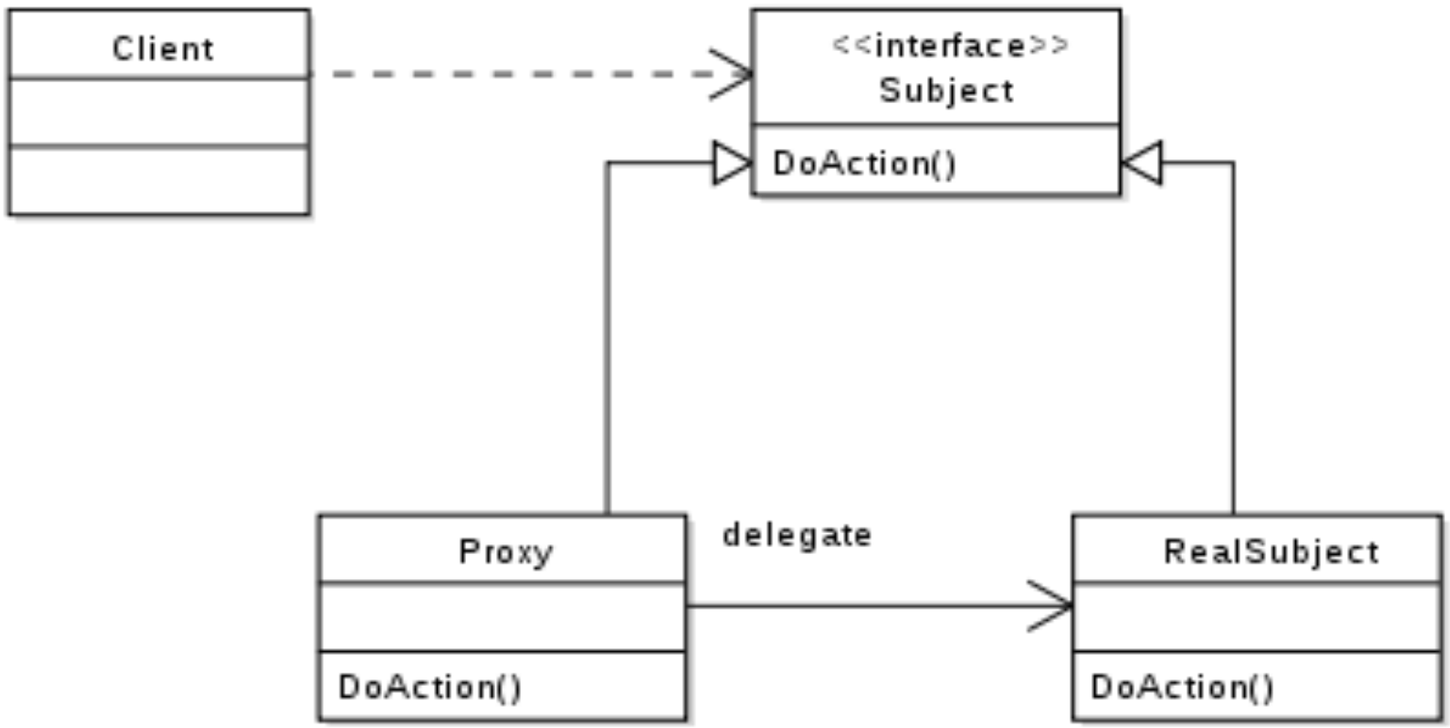
Mini-chat

- Proiectați și implementați o aplicație client-server pentru un mini-chat având următoarele funcționalități:
  - *Login.* După autentificarea cu succes, o nouă fereastră se deschide în care sunt afișați toți prietenii *online* ai utilizatorului și o listă cu mesajele trimise/primite de utilizator. De asemenea, toți prietenii online văd în lista lor că utilizatorul este *online*.
  - *Trimiterea unui mesaj*. Un utilizator poate trimite un mesaj text unui prieten care este online. După trimiterea mesajului, prietenul vede automat mesajul în fereastra lui.
  - *Logout.* Toți prietenii online ai utilizatorului văd în lista lor că utilizatorul nu mai este *online*.

Exemplu Java

Șablonul Proxy

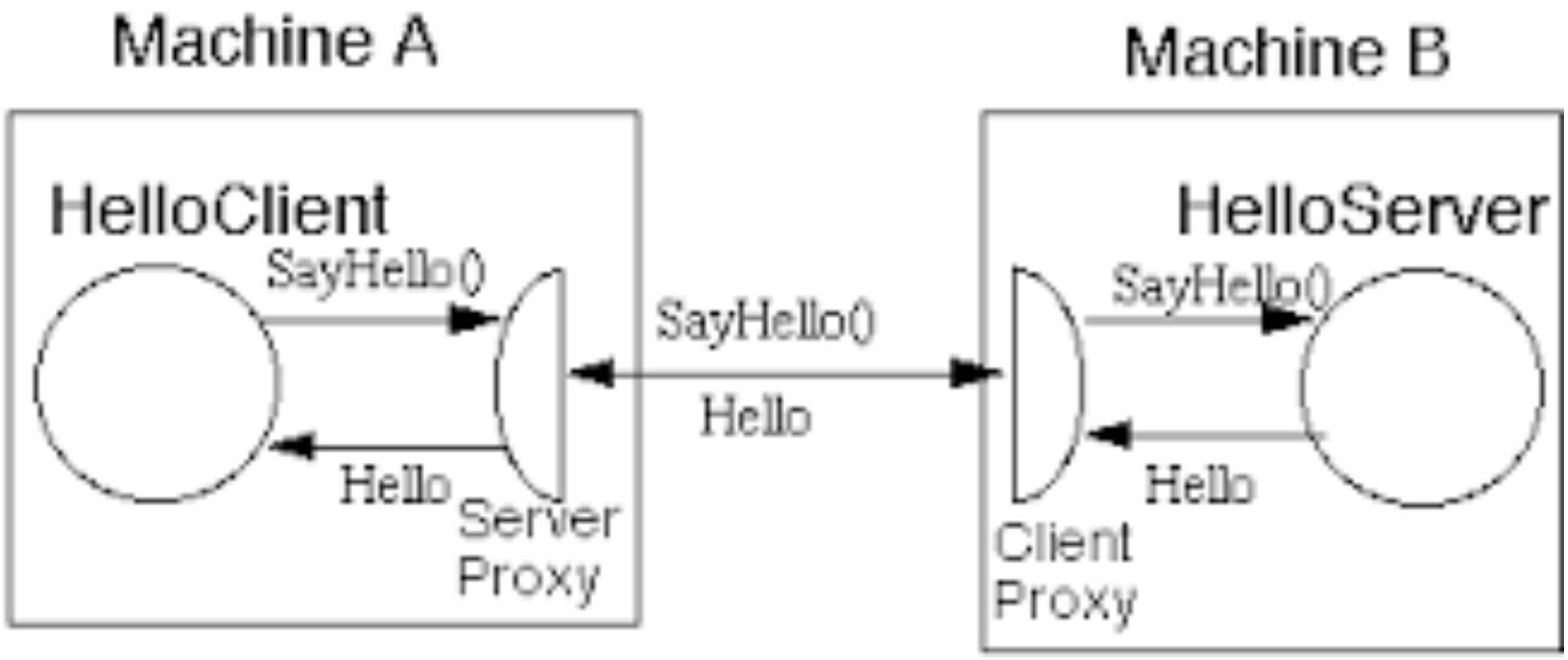
- Asigură pentru un obiect existent, un surogat sau un înlocuitor în scopul controlării accesului la acesta.
- Înlocuitorul poate fi:
  - *Proxy la distanță* (eng. *remote proxy*) - obiect în alt spațiu de adresă,
  - *Proxy virtual* (eng. *virtual proxy*) - un obiect mare din memorie,
  - *Proxy de protecție* - controlează accesul la obiectul original,
  - etc.



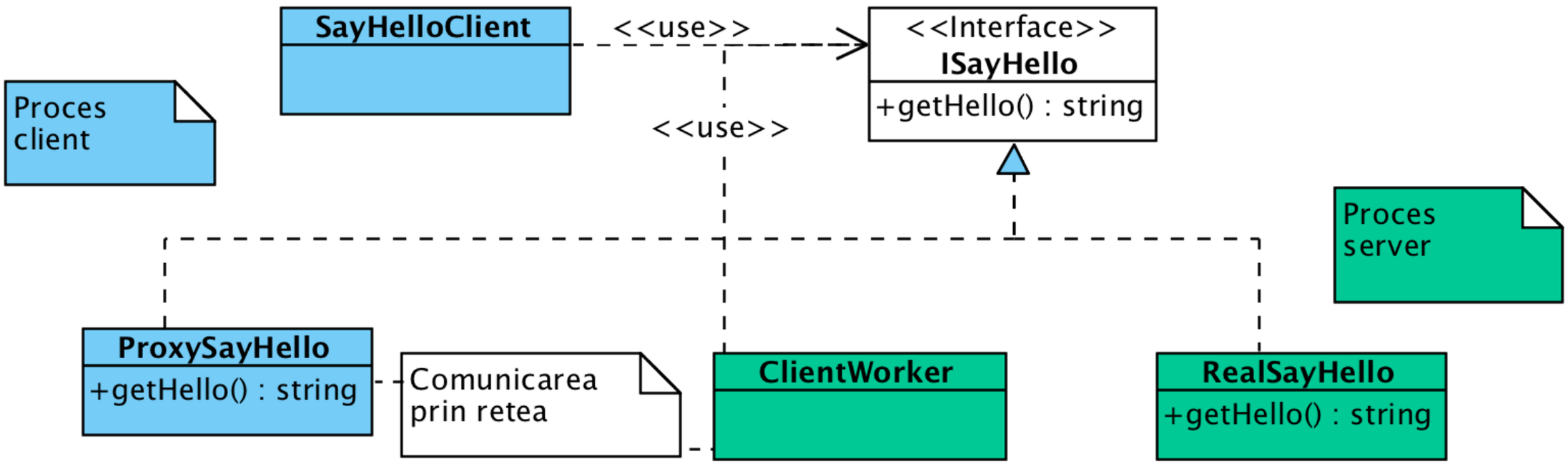
Șablonul *Remote Proxy*

- ***Remote Proxy*** oferă un înlocuitor local pentru un obiect aflat în alt spațiu de adresă/memorie.
- Înlocuitorul este responsabil cu codificarea unei cereri, a parametrilor și trimiterea lor către obiectul real aflat într-un spațiu de adresă diferit.
- *Clientul* cererii crede că comunică cu obiectul real, dar este un proxy între ei.
- Proxy-ul transformă cererile clientului în cereri la distanță, obține rezultatul cererii și îl transmite clientului.

Șablonul Remote Proxy



Șablonul Remote Proxy

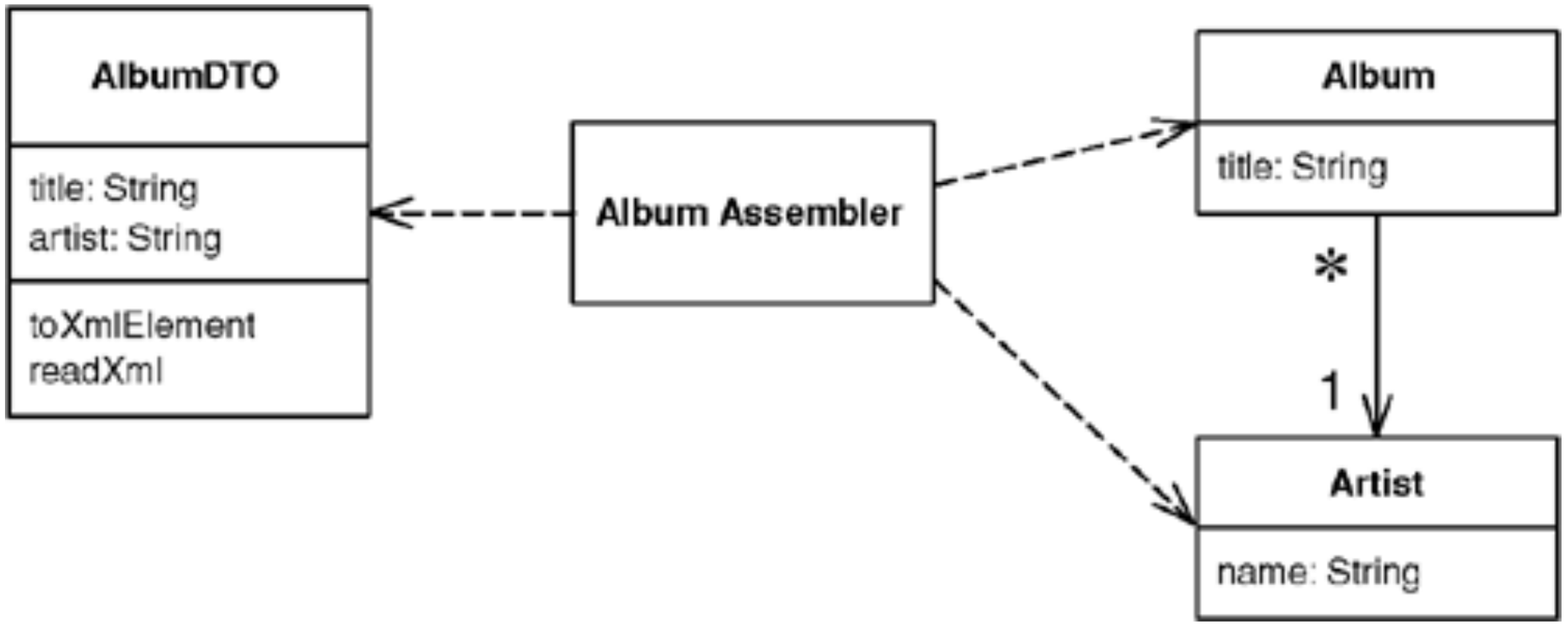


Șablonul *Data Transfer Object* (DTO)

- Un ***DTO*** este *un obiect care conține informația ce trebuie transmisă între unul sau mai multe procese pentru a reduce numărul de apeluri* (dintre procese).
- Fiecare apel de metodă remote este costisitor, de aceea numărul de apeluri ar trebui redus și mai multă informație ar trebui transmisă la un apel.
- O soluție posibilă este de a folosi mai mulți parametri:
  - dificil de programat
  - în unele cazuri nu este posibil (ex., în Java o metodă poate returna o singură valoare).
- Soluția: crearea unui DTO (*Data Transfer Object*) care păstrează toată informația necesară unui apel. De obicei obiectul este serializabil (binar, XML, etc.) pentru a putea fi transmis prin rețea.
- Un alt obiect este responsabil cu conversia datelor din model într-un DTO și invers.

Șablonul Data Transfer Object

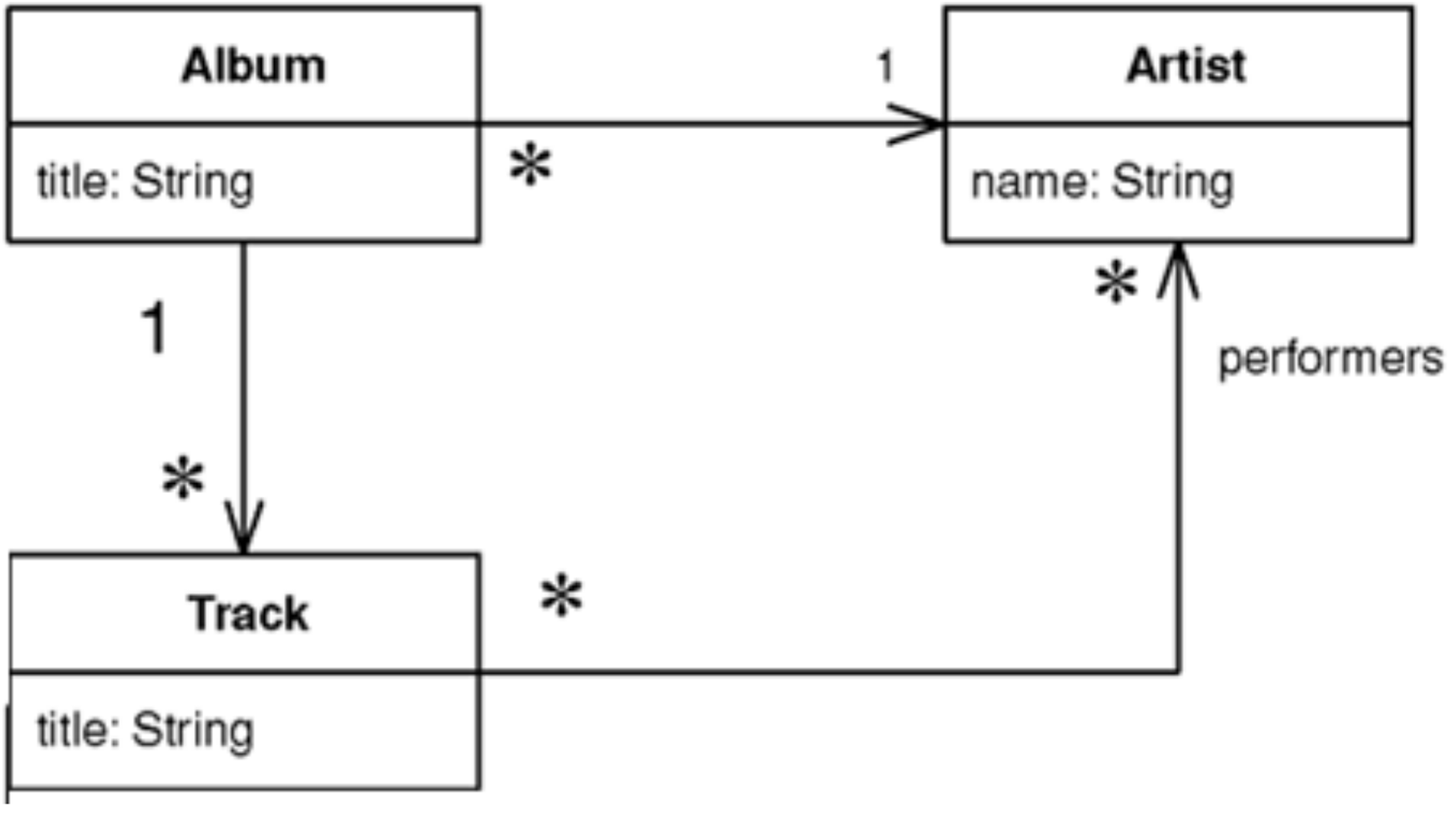
- Un DTO conține, de obicei, *multe atribute și metode de tip get/set pentru acestea*.
- Când un obiect remote are nevoie de date, cere DTO-ul corespunzător. DTO poate să conțină mai multă informație decât este necesară la acel apel, dar ar trebui să conțină toată informația de care va avea nevoie obiectul remote o perioadă.
- Un DTO conține de obicei informație provenind de la mai multe obiecte din model.

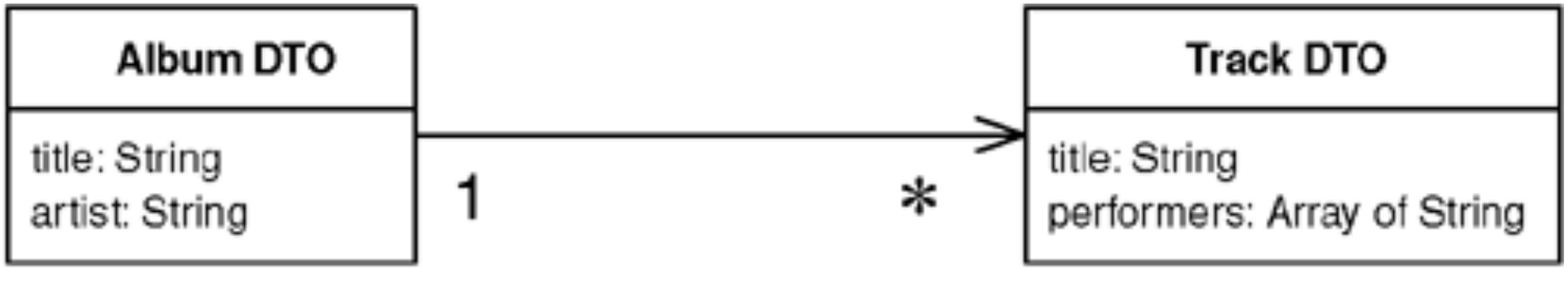


Șablonul Data Transfer Object

- Un DTO ar trebui folosit ori de câte ori este necesară transmiterea mai multor date între două procese într-un singur apel de metodă.
- *Alternative*:
  - De a folosi metode de tip set/get cu mai mulți parametrii transmiși prin referință.
    - Multe limbaje (ex. Java) permit returnarea unei singure valori.
    - Alternativa poate fi folosită pentru actualizări (metode de tip set), dar nu poate fi folosită pentru a obține date (metode de tip get).
- Folosirea unei reprezentări sub formă de string.
  - Totul va fi cuplat cu reprezentarea sub formă de string (poate fi costisitoare).

Data Transfer Object - Exemplu





Data Transfer Object - Exemplu

```
class AlbumAssembler{
   public AlbumDTO writeDTO(Album subject) {
      AlbumDTO result = new AlbumDTO();
      result.setTitle(subject.getTitle());
      result.setArtist(subject.getArtist().getName());
      writeTracks(result, subject);
      return result;
   }
   private void writeTracks(AlbumDTO result, Album subject) {
      List<TrackDTO> newTracks = new ArrayList<TrackDTO>();
       for(Track track: subject.getTracks()){
            TrackDTO newDTO = new TrackDTO();
            newDTO.setTitle(track.getTitle());
            writePerformers(newDTO, track);
            newTracks.add(newDTO);
       }
result.setTracks(newTracks.toArray(new TrackDTO[newTracks.size()]));
   }
```

Data Transfer Object - Exemplu

```
private void writePerformers(TrackDTO dto, Track subject) {
        List<String> result = new ArrayList<String>();
         for(Artist artist: subject.getPerformers()){
                result.add(artist.getName());
        }
        dto.setPerformers(result.toArray(new String[result.size()]));
    }
```

Networking în Java

- **`java.net`** - pachetul conține clase pentru comunicarea TCP/UDP prin rețea.
- TCP: **`Socket`** și **`ServerSocket`**.
- UDP: **`DatagramPacket`**, **`DatagramSocket`** și **`MulticastSocket`**.
- Clasa **`InetAddress`** reprezintă o adresă IP:
  - **`Inet4Address`**: pentru adrese IPv4 (32 bits).
  - **`Inet6Address`**: pentru adrese IPv6 (128 bits).

```
InetAddress localHost=InetAddress.getLocalHost();
InetAddress googAdr=InetAddress.getByName("www.google.com");
```

- **`InetSocketAddress`** asociere între o adresă IP și un port:

```
InetSocketAddress(InetAddress addr, int port) ;
InetSocketAddress(String hostname, int port);
```

Networking in Java

- **`ServerSocket`** reprezintă clasa corespunzătoare serverului care așteaptă conexiuni TCP.

- Constructori/Metode:

```
public ServerSocket(int port) throws BindException, IOException
public ServerSocket( ) throws IOException //not bind yet, since Java 1.4

//binds a server to a port
public void bind(SocketAddress endpoint) throws IOException

//blocks and waits for clients
public Socket accept( ) throws IOException

//closes the server
public void close() throws IOException
```

Networking în Java

```
ServerSocket server=null;
try{
    server=new ServerSocket(5555);
    while(keepProcessing){
         Socket client=server.accept();
         //processing code
    }
}catch(IOException ex){
    //…
}finally{
     if(server!=null){
        try{
           server.close();
        }catch(IOException ex){…}
    }
}
```

Networking în Java

- **`java.net.Socket`** deschide o conexiune TCP din partea clientului.

```
public Socket(String host, int port) throws UnknownHostException,
                                            IOException
public Socket(InetAddress host, int port) throws IOException
```

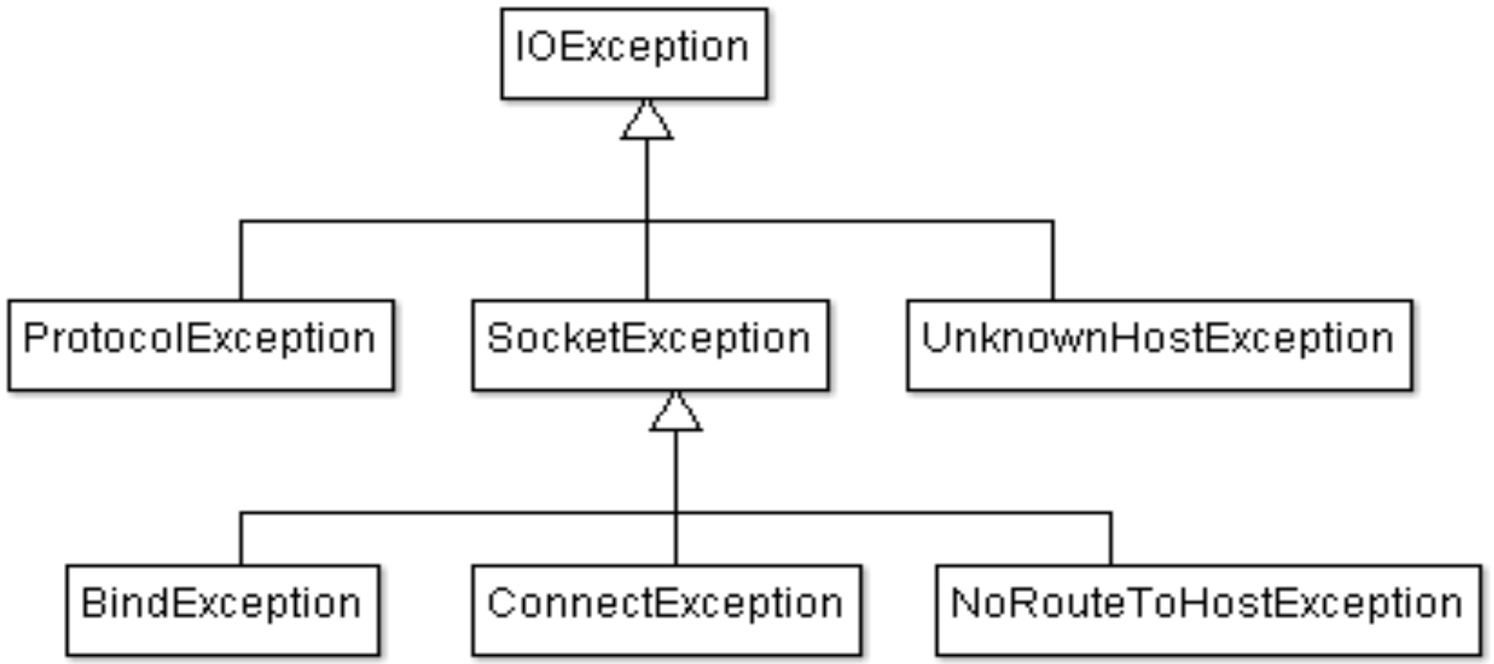
- Metode:

```
public int getPort()
public InputStream getInputStream() throws IOException
public OutputStream getOutputStream() throws IOException
public void close() throws IOException
```

Networking în Java

```
try (Socket connection=new Socket(“172.30.106.5”, 5555)){
    //processing code
}catch(UnknownHostException e){
   //…
}catch(IOException  e){
   //…
}
```

Excepții


IOException
ProtocolException
SocketException
UnknownHostException
BindException
ConnectException
NoRouteToHostException

Threading în Java

- *Două modalități de definire a unui thread*:
  - Extinderea clasei **`Thread`** și redefinirea metodei **`run`**.
  - Implementarea interfeței **`Runnable`** și definirea metodei **`run`**.

- Crearea unui thread se face prin intermediul clasei **`Thread`**

```
public Thread()
public Thread(Runnable target)
```

- Pornirea execuției unui thread se face prin apelul metodei **`start`** din clasa **`Thread`**:

```
public void start()
```

Sincronizarea threadurilor

- Instrucțiunea **synchronized**

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

- Sincronizarea unei metode:

```
public synchronized void methodA();
```

- Yielding: *un thread renunța la CPU alocat și permite execuția altui thread*:

```
public static void yield( );
```

```
public void run( ) {
  while (true) {
    // Time and CPU consuming thread's work…,
    Thread.yield( );
  }
}
```

Utilități Java Concurrency

- Java 5 a introdus utilități pentru concurență - un framework extensibil care permite crearea containerelor de thread-uri și cozi sincronizate (eng. *blocking queues*):

  - **`java.util.concurrent`**: Tipuri utile în programarea concurentă (ex. executors)

  - **`java.util.concurrent.atomic`**: Programare concurentă avansată

  - **`java.util.concurrent.locks`**: Mecanisme de blocare avansate, mai performante decât notify/wait.

Taskuri Java

- Un obiect *task* Java este un obiect a cărui clasa implementează interfața **`java.lang.Runnable`** (*taskuri runnable*) sau interfața **`java.util.concurrent.Callable`** (*taskuri callable*).

```
public interface Runnable{
    void run()
}

public interface Callable<V>{
     V call() throws Exception
}
```

- Metoda **`call()`** poate *returna o valoare și poate arunca excepții* (checked).

Execuția taskurilor Java

- Interfața Executor - execuția taskurilor runnable:

```
public interface Executor{
  void execute(Runnable command)
}
```

- **`ScheduledThreadPoolExecutor, ThreadPoolExecutor`**
- *Dezavantaje:*
  - Se axează doar pe **`Runnable`**. Metoda **`run()`** nu returnează nici o valoare. Este dificilă returnarea unei valori ca și rezultat al execuției taskului.
  - Nu oferă posibilitatea monitorizării progresului execuției unui task runnable care se execută (se execută încă?, anulat? execuția s-a încheiat?)
  - Nu poate executa mai multe taskuri.
  - Nu oferă posibilitatea opririi unui executor.

ExecutorService

- **`java.util.concurrent.ExecutorService`** soluția pentru problemele apărute la interfața **`Executor`**.
- Este implementat folosind un container de threaduri (eng. *thread pool*).

```
public interface ExecutorService extends Executor {

  void shutdown();

  List<Runnable> shutdownNow();

  <T> Future<T> submit(Callable<T> task);

  <T> Future<T> submit(Runnable task,  T result);

  <T> List<Future<T>> invokeAll(Collection<? extends  Callable<T>> tasks);

  <T> T invokeAny(Collection<? extends Callable<T>> tasks);

//alte metode

}
```

- `ScheduledThreadPoolExecutor, ThreadPoolExecutor`

Executorul trebuie oprit după terminarea execuției, altfel aplicația nu își va încheia execuția.

Interfața Future

- Un obiect de tip ***Future*** reprezintă rezultatul unui calcul asincron.
- Rezultatul este numit *future* pentru că de obicei nu va fi disponibil decât la un moment în viitor.
- Are metode pentru: anularea execuției unui task, obținerea rezultatului execuției, determinarea dacă un task și-a încheiat execuția.

```
public interface Future<V>{
  boolean isCancelled();
  boolean isDone();
  boolean cancel(boolean mayInterruptIfRunning)
  V get() throws InterruptedException, ExecutionException;
  //alte metode …
}
```

Clasa Executors

- Clasa **`Executors`** conține metode statice care returnează obiecte de tip **`ExecutorService`**:

  - **`newFixedThreadPool(int nThreads): ExecutorService`**
  - **`newSingleThreadExecutor(): ExecutorService`**
  - **`newCachedThreadPool():ExecutorService`**
  - **`newWorkStealingPool(): ExecutorService`**

Colecții concurente

- Colecții folositoare în programarea concurentă.
- Începând cu versiunea 1.5
- Interfața **`BlockingQueue`**:
  - ***Coadă*** - conține metode care așteaptă ca coadă să devină nevidă la scoaterea unui element, respectiv așteaptă eliberarea spațiului la adăugarea unui element.
  - Implementările BlockingQueue au fost proiectate și implementate pentru a fi folosite în situații de tip producător-consumator.
  - **`ArrayBlockingQueue`**, **`LinkedBlockingQueue`**, **`PriorityBlockingQueue`**, etc.
- Interfața **`BlockingDeque`**:
  - Extinde **`BlockingQueue`** și oferă suport pentru operații de tip FIFO și LIFO.
  - **`LinkedBlockingDeque`**
- Interfața **`ConcurrentMap:`**
  - Subinterfață a **`java.util.Map`**
  - **`ConcurrentHashMap, ConcurrentSkipListMap.`**

Exemplu BlockingQueue

- Producător-Consumator simplu cu BlockingQueue

```
//ambele threaduri au referință la obiectul messages

//inițializarea
private BlockingQueue<String> messages=new LinkedBlockingQueue<String>();

//Producator
try {
     messages.put(message);
} catch (InterruptedException e) {
     e.printStackTrace();
}

//Consumator
String message = messages.take();
```

Actualizare GUI

- Interfețele grafice (JavaFX, Swing) folosesc obiecte de tip Component.
- Aceste obiecte ***pot fi modificate (actualizate, șterse, etc) doar de threadul care le-a creat***.
- Nerespectarea acestei reguli are rezultate neașteptate sau aruncă excepții.

```
Platform.runLater(new Runnable() {                    //JavaFX
      @Override
       public void run() {
         //codul care modifica informatia de pe interfata grafica
         label.setText("New text …");
       }
 });
```

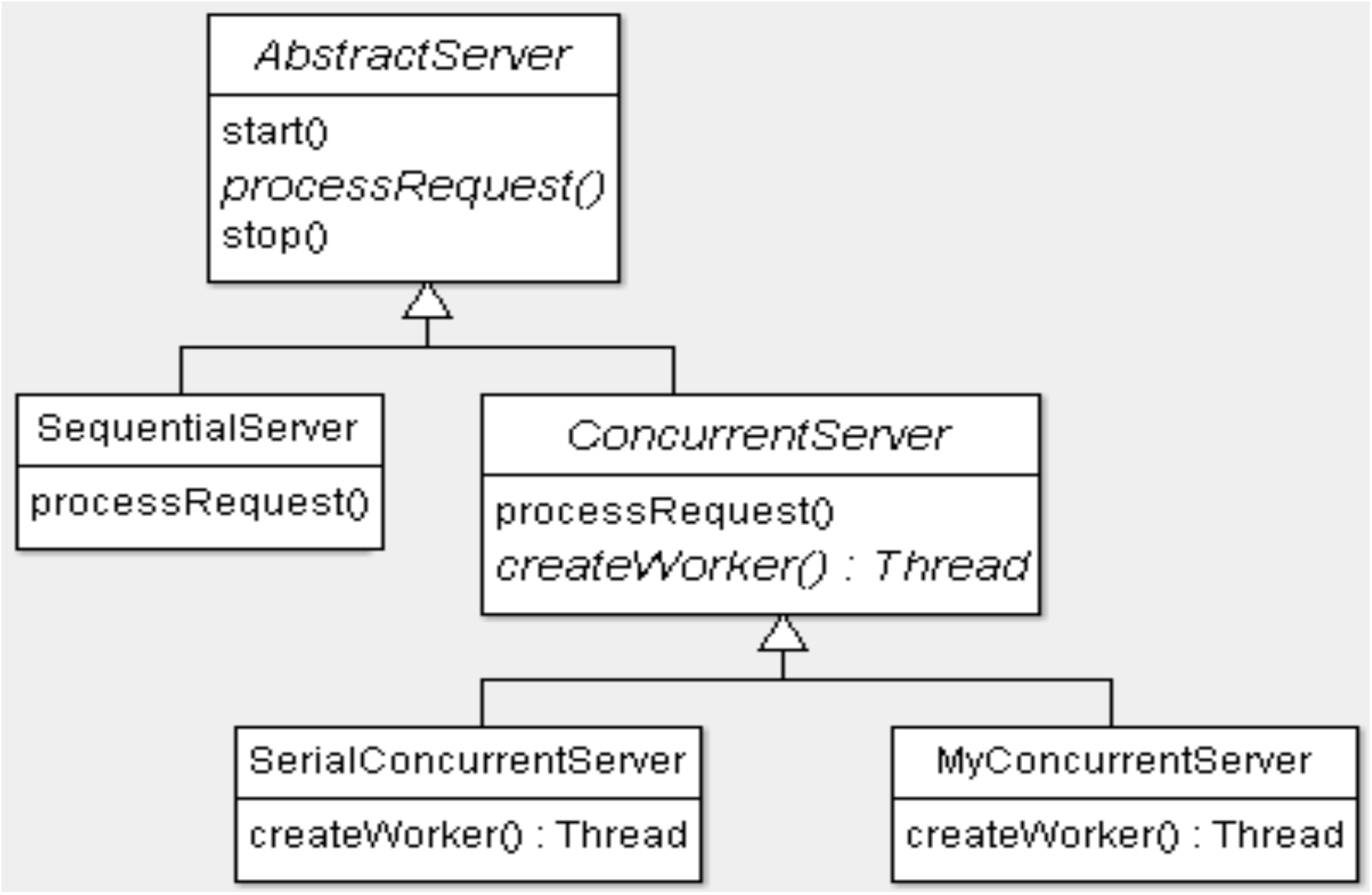
//sau, folosind funcții lambda

```
Platform.runLater(() -> {                        //JavaFX
        //codul care modifica informatia de pe interfata grafica
        label.setText("New text …");
 });
```

Exemplu Java

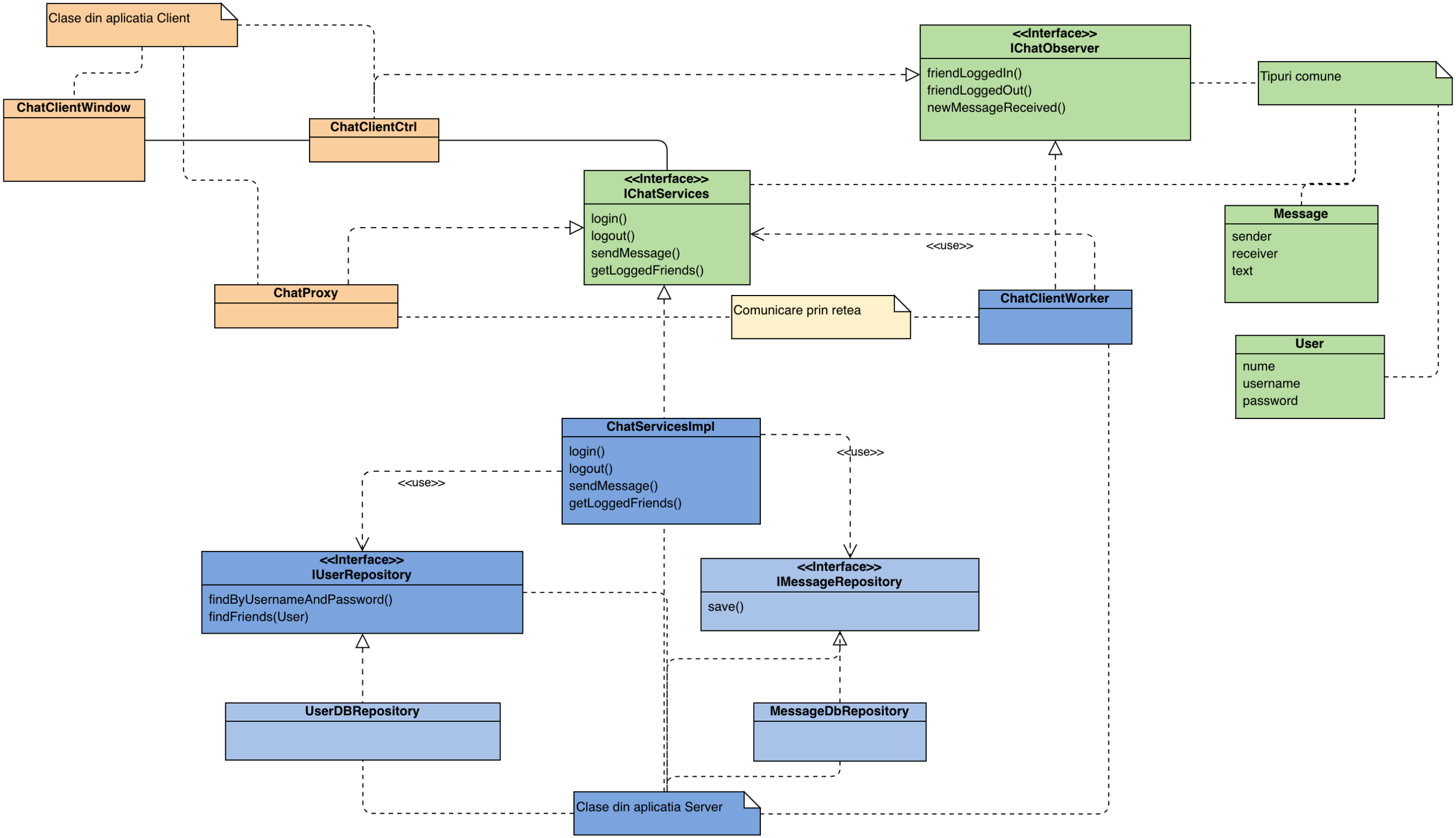
- O aplicație simplă client/server:
  - Serverul așteaptă conexiuni.
  - Clientul se conectează la server si îi trimite un text.
  - Serverul returnează textul scris cu litere mari, la care adaugă data și ora la care a fost primit textul.

Server Template

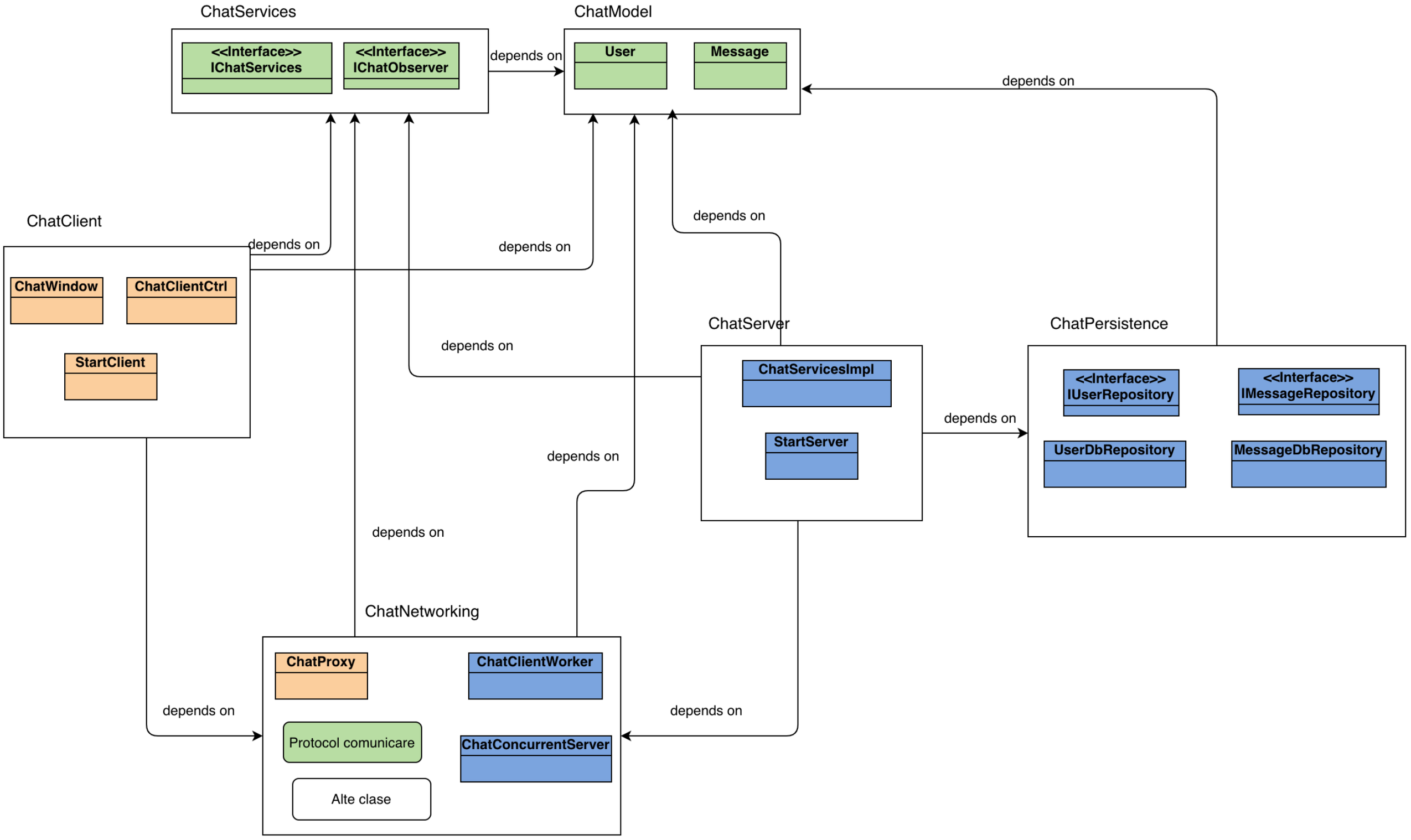


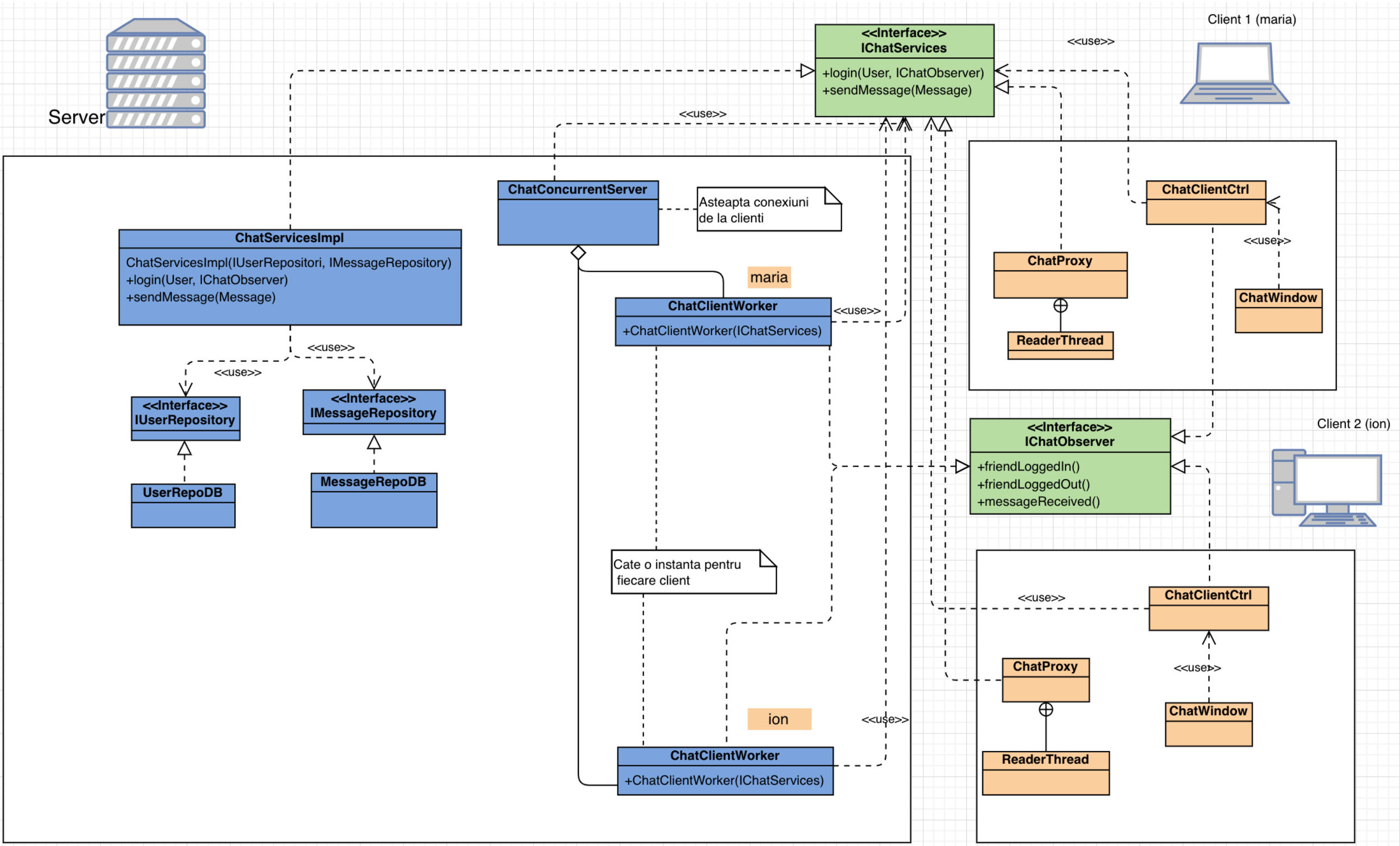
Mini-Chat

- Proiectare (diagrame)

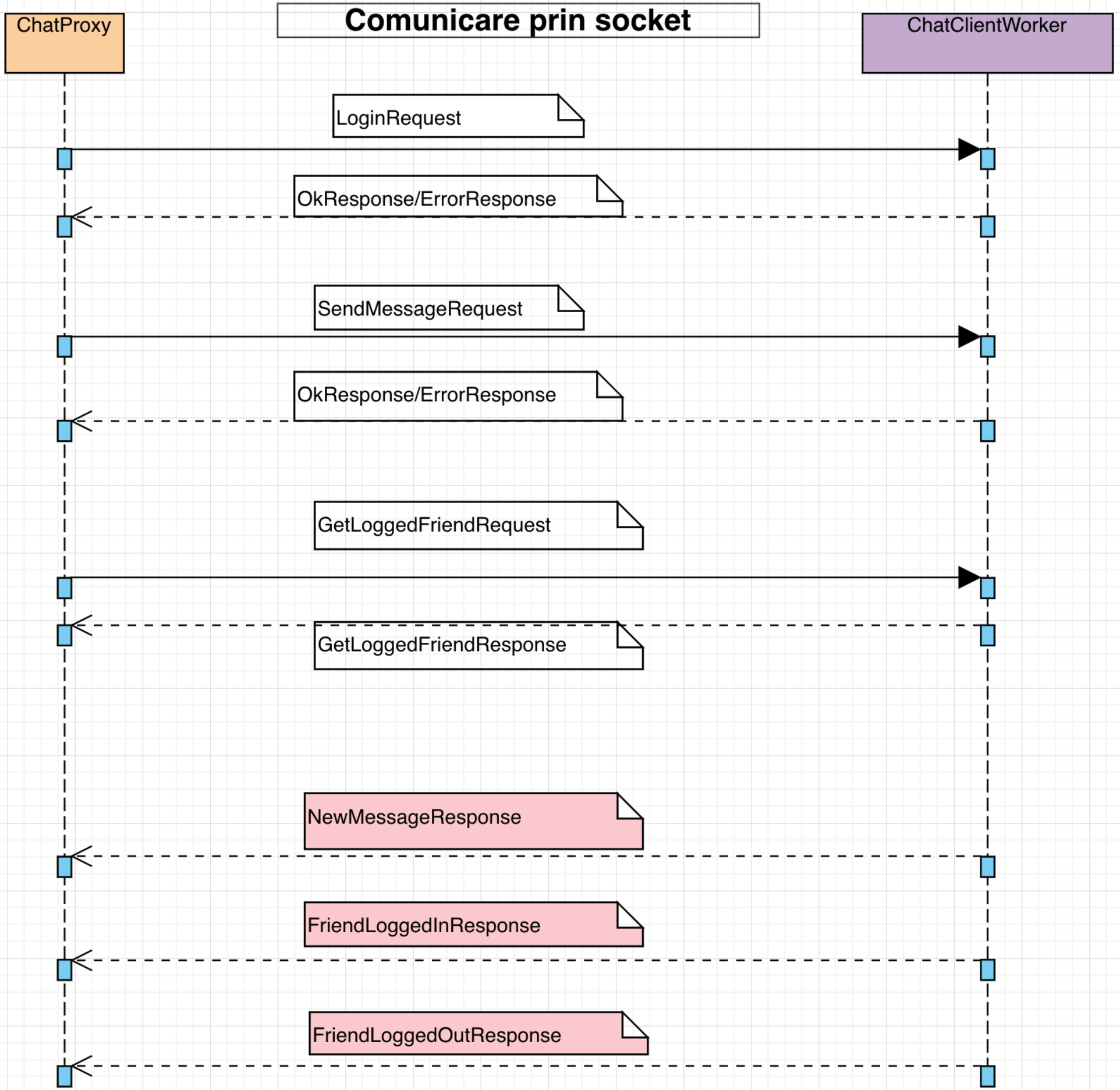

Clase din aplicatia Client
ChatClientWindow
ChatClientCtrl
<<Interface>>
IChatObserver
friendLoggedIn()
friendLoggedOut()
newMessageReceived()
Tipuri comune
<<Interface>>
IChatServices
login()
logout()
sendMessage()
getLoggedFriends()
<<use>>
Message
sender
receiver
text
ChatProxy
Comunicare prin retea
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

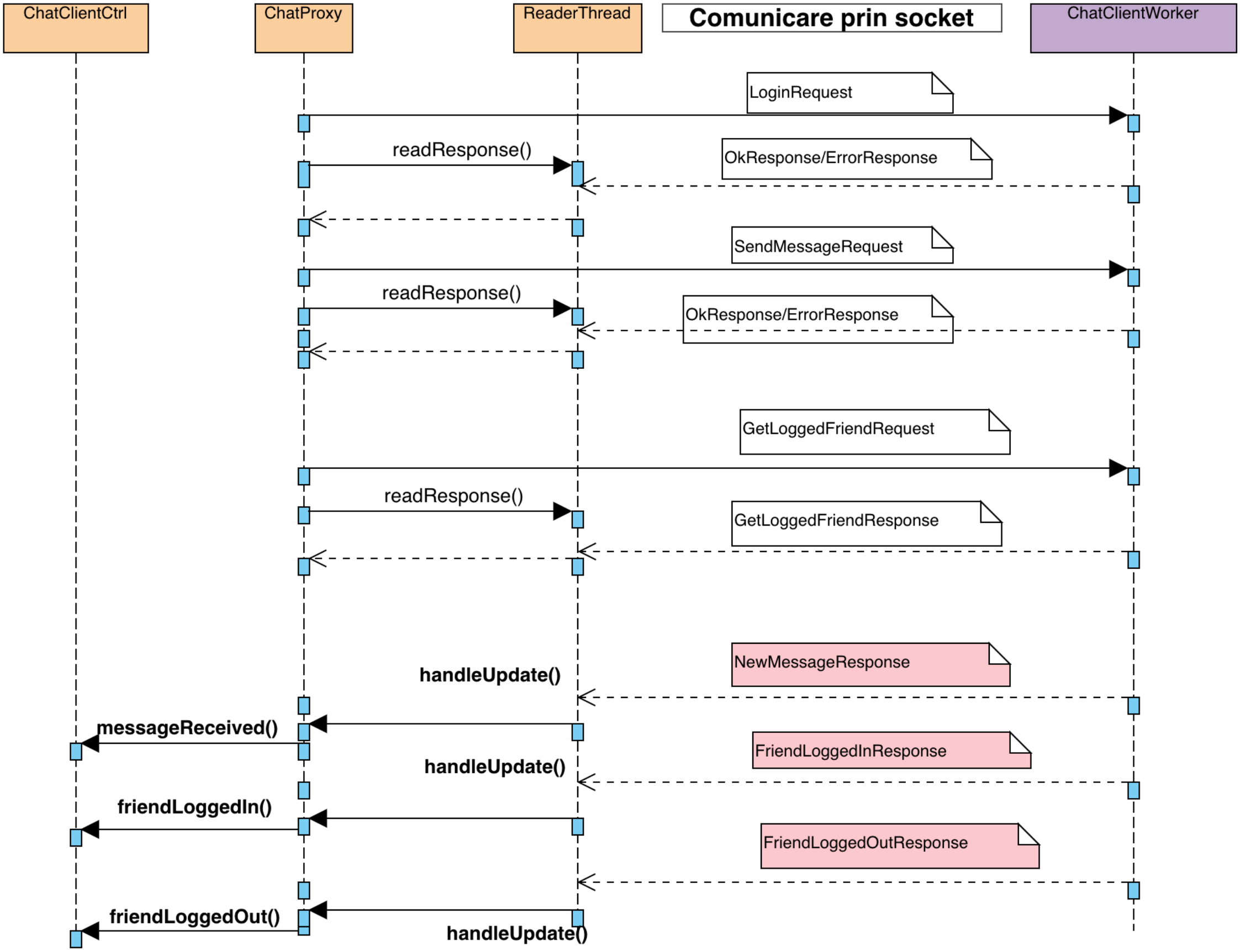
Proiecte


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

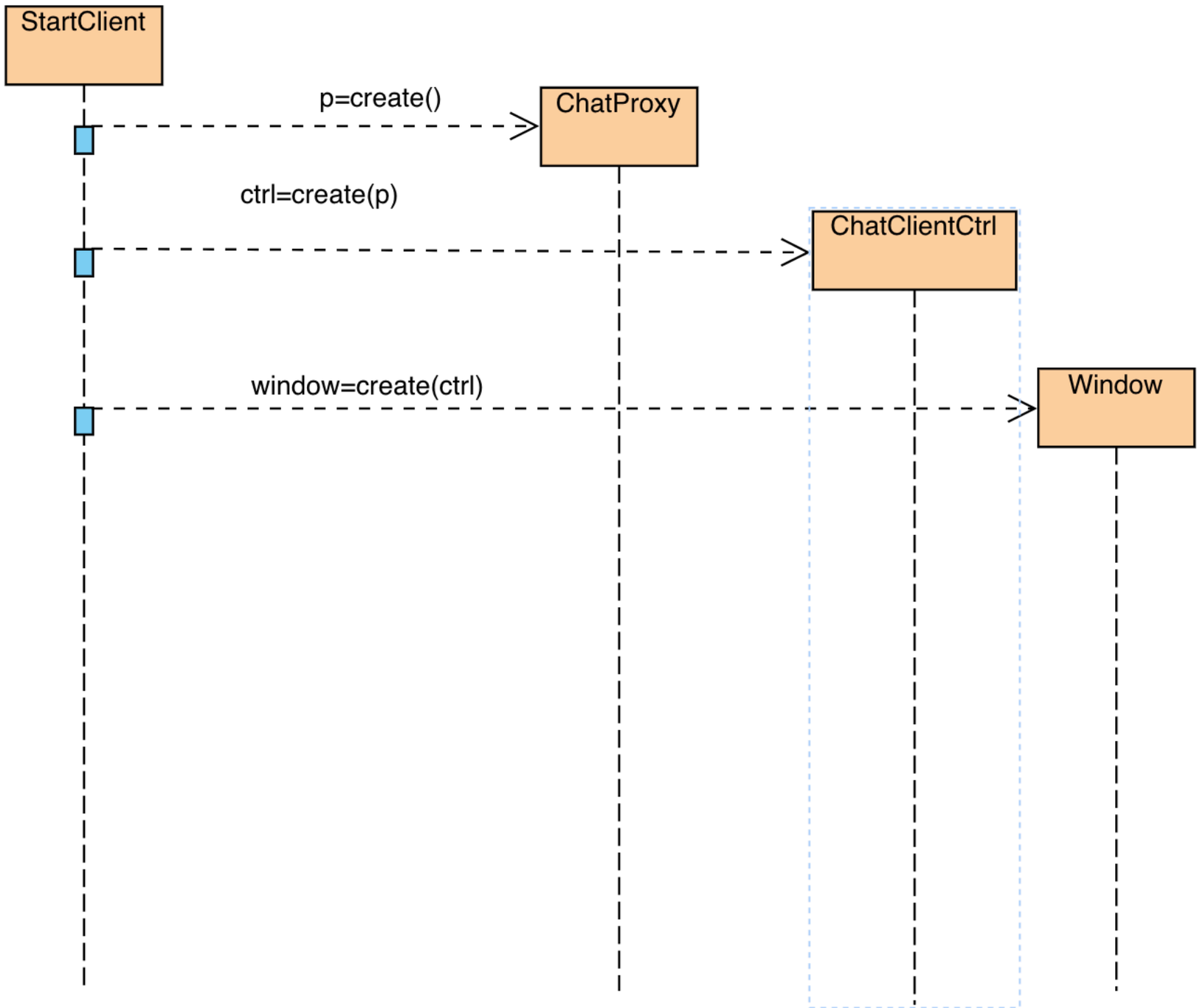

Server
<<Interface>>
IChatServices
+login(User, IChatObserver)
+sendMessage(Message)
<<use>>
Client 1 (maria)
<<use>>
ChatConcurrentServer
Asteapta conexiuni
de la clienti
ChatClientCtrl
ChatServicesImpl
ChatServicesImpl(IUserRepositori, IMessageRepository)
+login(User, IChatObserver)
+sendMessage(Message)
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

Comunicare prin socket




ChatClientCtrl
ChatProxy
ReaderThread
Comunicare prin socket
ChatClientWorker
LoginRequest
readResponse()
OkResponse/ErrorResponse
SendMessageRequest
readResponse()
OkResponse/ErrorResponse
GetLoggedFriendRequest
readResponse()
GetLoggedFriendResponse
handleUpdate()
NewMessageResponse
messageReceived()
FriendLoggedInResponse
handleUpdate()
friendLoggedIn()
FriendLoggedOutResponse
friendLoggedOut()
handleUpdate()

Mini-Chat (Pornire client)



Mini-Chat (Pornire server)