Systems for Design and Implementation

2015-2016

Course 2

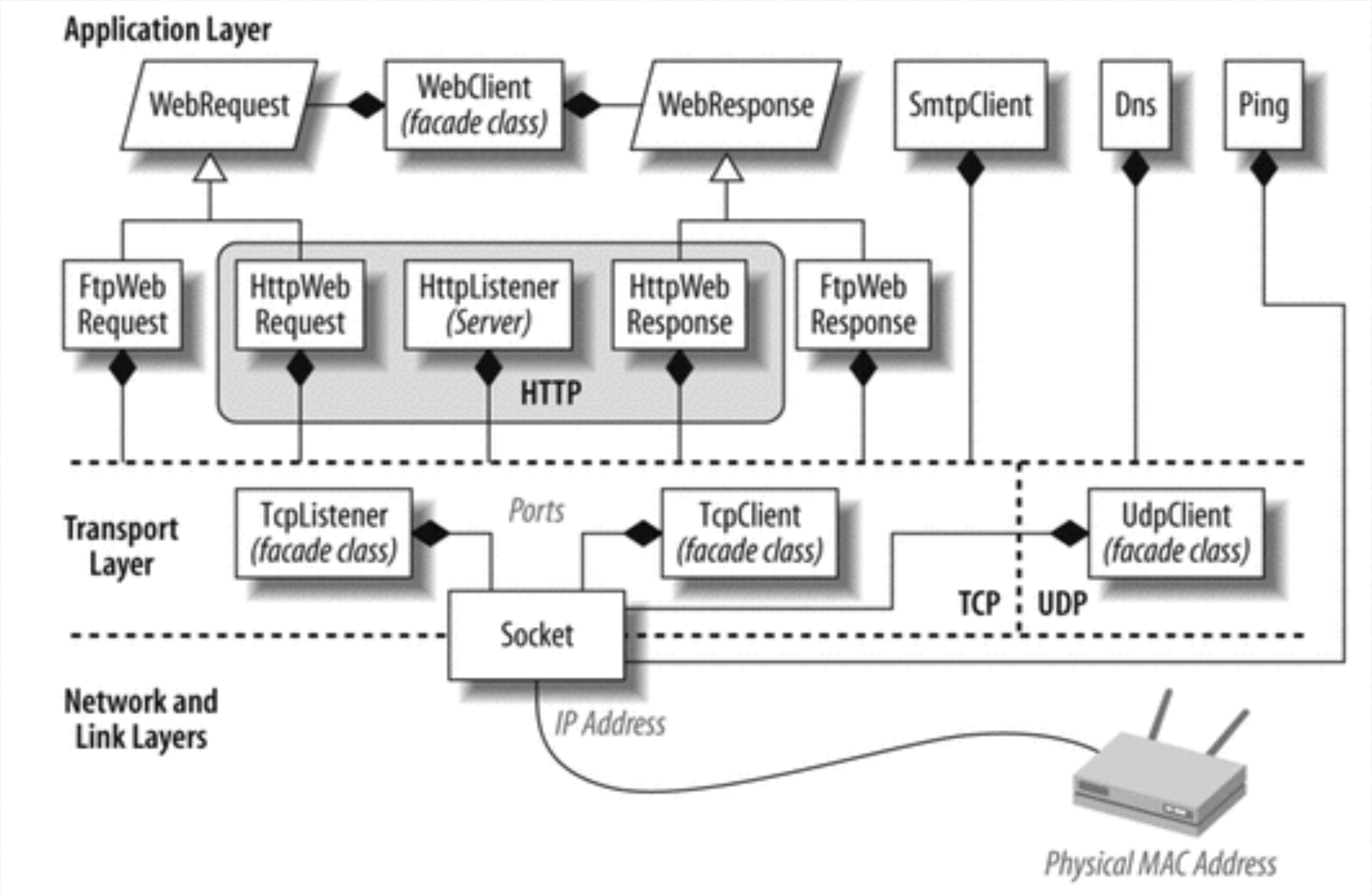
Contents

- Networking in C#
- Threading in C#
- Networking in Java
- Threading in Java
- Proxy pattern

Networking in C#

- The .NET framework contains a variety of classes for communicating via standard network protocols, such as HTTP, TCP/IP, and FTP.
- They are contained in the **`System.Net.*`** namespaces.
  - A **`WebClient`** façade class for simple download/upload operations via HTTP or FTP.
  - **`WebRequest`** and **`WebResponse`** classes for more control over client-side HTTP or FTP operations.
  - **`HttpListener`** for writing an HTTP server.
  - **`SmtpClient`** for constructing and sending mail messages via SMTP.
  - **`Dns`** for converting between domain names and addresses.
  - **`TcpClient`**, **`UdpClient`**, **`TcpListener`**, and **`Socket`** classes for direct access to the transport and network layers.

Networking in C#



Networking in C#

- The `IPAddress` class in the `System.Net` namespace represents an address in either protocols IPv4 (32 bits) or IPv6 (128 bits).
- It has a constructor accepting a byte array, and a static `Parse` method accepting a correctly formatted string:

```
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
```

- An IP address and port combination is represented in the .NET Framework by the `IPEndPoint` class:

```
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
```

- Port numbers: 1 – 65535.
- The TCP and UDP ports from 49152 to 65535 are officially unassigned.

Networking in C#

- TCP and UDP constitute the transport layer protocols on top of which most Internet—and local area network—services are built.
- TCP is connection-oriented and includes reliability mechanisms.
- UDP is connectionless, has a lower overhead, and supports broadcasting.

- For TCP network communication: `TcpClient` and `TcpListener` facade classes, or the feature-rich `Socket` class.
- For UDP network communication: `UdpClient`, `TcpListener`

System.Net.Sockets Namespace

- The **`TcpClient`**, **`TcpListener`**, and **`UdpClient`** classes encapsulate the details of creating TCP and UDP connections to the Internet.
- **`Socket`** implements the Berkeley sockets interface.
- **`SocketException`** the exception that is thrown when a socket error occurs.
- **`NetworkStream`** provides the underlying stream of data for network access.

TcpListener

- A simple 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` requires the local IP address on which to listen (i.e., the computer has two network cards). The `IPAddress.Any` can be used to tell it to listen on all (or the only) local IP addresses.
- `AcceptTcpClient` blocks until a client request is received.

TcpClient

- A simple Tcp client:

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

- `TcpClient` immediately establishes a connection upon construction to a server at the given IP or domain name address and port.
- The constructor blocks until a connection is established.

NetworkStream

- `NetworkStream` provides two-way communication, for both transmitting and receiving bytes of data when a socket connection was established.
- Methods:
  - `Read`
  - `Close`
  - `Write`
  - `Seek`
  - `Flush`
- Properties:
  - `CanRead, CanWrite`
  - `Socket`
  - `DataAvailable`
  - `Length`

Threading in C#

- **`System.Threading`** namespace provides classes and interfaces that enable multithreaded programming:
  - **`Thread`** class.
  - **`ThreadStart`**, **`ParameterizedThreadStart`** delegates.
  - **`Monitor`**, **`Mutex`**, **`Semaphore`** classes for synchronization.
- Delegates: represent the method that executes on a Thread.

```
public delegate void ThreadStart();

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

- **`Thread`** class: creates and controls a thread, sets its priority, and gets its status.

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

Threading in 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
```

Thread Synchronization

- The synchronization constructs can be divided into:
  - *Exclusive locking*: allow just one thread to perform some activity or execute a section of code at a time. Their primary purpose is to let threads access shared writing state without interfering with one other.
    - `lock`, `Mutex`, and `SpinLock`.
  - *Nonexclusive locking*: lets you limit concurrency.
    - `Semaphore` and `ReaderWriterLock`.
  - *Signaling*: they allow a thread to block until receiving one or more notifications from other thread(s).
    - `ManualResetEvent`, `AutoResetEvent`, `CountdownEvent`, and `Barrier`.

Thread Synchronization –Locking

- `lock` statement:

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

- The `lock` statement acquires a lock on any reference-type instance.
- Only one thread can lock the synchronizing object at a time, and any contending threads are blocked until the lock is released. If more than one thread contends the lock, they are queued on a “ready queue” and granted the lock on a first-come, first-served basis.
- If another thread has already acquired the lock, the thread does not continue until the other thread releases its lock on that instance.

Thread Synchronization

```
class LockTest {
  static void Main( ) {
      LockTest lt = new LockTest ( );
      Thread t = new Thread(new ThreadStart(lt.run));
      t.Start( );
      lt.run( );
  }
  void run() {
      lock(this)
      for (char c='a'; c<='z'; c++) Console.Write(c);
  }
}

//abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz
```

Thread Synchronization

- The `System.Threading.Monitor` class provides an implementation of the monitor mechanism that allows the use of any reference-type instance as a monitor.
- The `Enter` method obtains a lock on an object. If the object is already held by another thread, `Enter` waits until the lock is released or the thread is interrupted by a `ThreadInterruptedException`.
- The `Exit` method releases the lock from a specified object.
- Every call to `Enter` for a given object on a thread should be matched with a call to `Exit` for the same object on the same thread.

Thread Synchronization

- The `TryEnter` methods are similar to the `Enter` method but do not require a lock on the object to proceed. These methods return true if the lock is obtained and false if it is not, optionally passing in a timeout parameter that specifies the maximum time to wait for the other threads to release the lock.

- The `lock` statement is a syntactic shortcut for calling the `Enter` and `Exit` methods of the `Monitor` class:

```
Monitor.Enter(lockerObj);
try {
     ...
}finally {
     Monitor.Exit(lockerObj);
}
```

Thread Synchronization - Signaling

- Event wait handles are the simplest of the signaling constructs:
  - `EventWaitHandle`- represents a thread synchronization event. Typically, one or more threads block on an `EventWaitHandle` until an unblocked thread calls the Set method, releasing one or more of the blocked threads.
  - `AutoResetEvent`, `ManualResetEvent`
  - `CountdownEvent` (Framework 4.0)
- `AutoResetEvent` notifies a waiting thread that an event has occurred (only one thread at a time)
  - `Set()` - release a waiting thread
  - `WaitOne()` - thread waits for a signal

Remarks:

1. If `Set` is called when no thread is waiting, the handle stays open for as long as it takes until some thread calls `WaitOne`.
2. Calling `Set` repeatedly when no one is waiting, does not allow more than one thread to execute when calling `WaitOne`.

Thread Synchronization - Signaling

- **`ManualResetEvent`**, is like **`AutoResetEvent`**, but it notifies all waiting threads that an event has occurred.
- Creating wait event handlers:
  - Constructors:

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

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

  - EventWaitHandle class

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

- Disposing wait handles:
  - Call its `Close` method to release the operating system resource.
  - You can drop all references to the wait handle and allow the garbage collector take care of the disposal.

Signaling Example

```
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");
    }
}
```

Thread Synchronization - Signaling

- The **`Monitor`** class provides a low-level signaling construct via the static methods **`Wait`** and **`Pulse`** (and **`PulseAll`**). The principle is that you write the signaling logic yourself using custom flags and fields (enclosed in **`lock`** statements)
- **`Monitor.Wait`** methods release the lock on an object and block the current thread until it reacquires the lock.
- **`Monitor.Pulse`** notifies a thread in the waiting queue of a change in the locked object's state. If multiple threads are waiting on the same monitor, **`Pulse`** activates only the first in the queue.
- **`Monitor.PulseAll`** notifies all waiting threads of a change in the object's state.

Thread Synchronization

```
class MonitorTest {
    static void Main( ) {
        MonitorTest mt = new MonitorTest( );
        Thread t = new Thread(new ThreadStart(mt.Go));
        t.Start();
        mt.Go( );
    }
    void Go( ) {
        for ( char c='a'; c<='z'; c++)
            lock(this) {
                Console.Write(c);
                Monitor.Pulse(this);
                Monitor.Wait(this);
            }
    }
}
// aabbccddeeffgghhiijjkkllmmnnooppqqrrssttuuvvwwxxyyzz
```

Barrier class

- The Barrier class implements a thread execution barrier, allowing many threads (*n*) to meet at a point in time.
- The class is very fast and efficient, and is built upon **`Wait`**, **`Pulse`**, and **`spinlocks`**.
- Class usage:
  1. Instantiate it, specifying how many threads should partake in the rendezvous
  2. Have each thread call **`SignalAndWait`** when it wants to rendezvous.

- You can also specify a post-phase action when constructing it, as a delegate that runs after **`SignalAndWait`** has been called *n* times, but before the threads are unblocked.

Barrier class - example

```
class BarrierTest {
  static Barrier _barrier = new Barrier (3);
  static void Main( ) {
     new Thread (letters).Start();
     new Thread (letters).Start();
     new Thread (letters).Start();
  }
  static void letters( ) {
     for ( char c='a'; c<='z'; c++){
        Console.Write(c);
       _barrier.SignalAndWait();
     }
  }
}
//Output
aaabbbcccdddeeefffggghhhiiijjjkkklllmmmnnnooopppqqqrrrssstttuuuvvvwwwxxxyyyzzz
```

Thread class

- Members:
  - `Abort:` raises a `ThreadAbortException` in the thread on which it is invoked, to begin the process of terminating the thread. Calling this method usually terminates the thread.
  - `Join`. Overloaded. Blocks the calling thread until a thread terminates.
  - `Sleep`. Overloaded. Blocks the current thread for the specified number of milliseconds.
  - `Name` Gets or sets the name of the thread.
  - `Priority` Gets or sets a value indicating the scheduling priority of a thread.
  - Etc.

Tasks

- A thread is a low-level tool for creating concurrency, but it has limitations:
  - You can pass data into a thread, but you cannot easily get a "return value" back from a thread that you `Join`. If the operation throws an exception, catching and propagating that exception is not easy to handle.
  - You cannot tell a thread to start something else when it has finished (instead you must `Join` it, blocking your own thread).
- A `Task` is higher-level abstraction that represents a concurrent operation that may or may not be backed by a thread.
  - Tasks are compositional (you can chain them together through the use of continuations).
  - They can use the thread pool to lessen startup latency.
- The `Task` types were introduced in Framework 4.0 as part of the parallel programming library.
- `System.Threading.Tasks` namespace.

Starting a Task

- In Framework 4.5 you start a `Task` backed by a thread with the static method `Task.Run` to which you pass in an `Action` delegate:

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

- In Framework 4.0, you start it by calling `Task.Factory.StartNew:`

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

- Tasks use pooled threads by default, which are background threads.
  - When the main thread ends, so do any tasks that you create.
  - You must block the main thread after starting the task to insure completion.
- Calling Task.Run this way is similar to starting a thread (without implicit thread pooling):

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

- `Task.Run` returns a `Task` object that can be used to monitor its progress.
- There is no need to call the `Start` method.

Returning the value

- `Task` has a generic subclass called `Task<TResult>` that allows a task to emit a return value.
- `Task<TResult>` can be obtained by calling `Task.Run` with a `Func<TResult>` delegate (or a compatible lambda expression).
- The result can be obtained later by using the `Result` property.
- If the task is not finished, accessing this property will block the current thread until the task finishes:

```
Task<int> task = Task.Run(()=>{int x=23; return 2*x; });
int result = task.Result;    // Blocks if not already finished
 Console.WriteLine (result);     // 3
```

Exceptions and Tasks

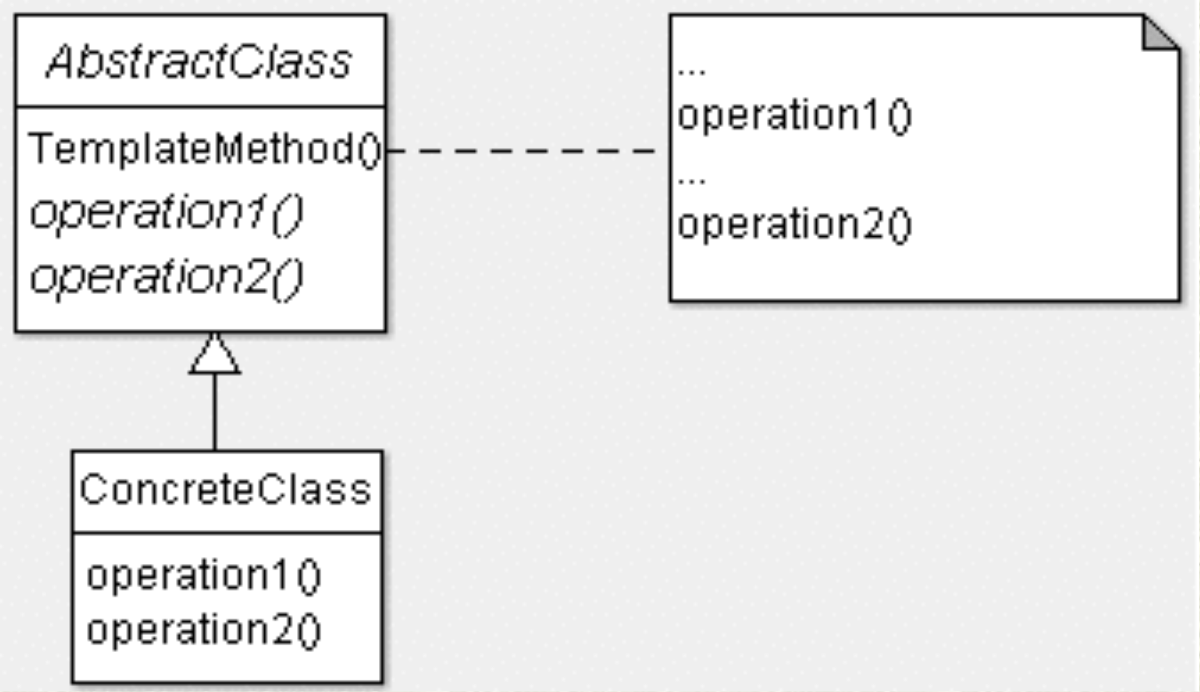
- Tasks propagate exceptions (unlike threads).
- If the code in the task throws an unhandled exception, that exception is automatically rethrown to whoever calls `Wait()` or accesses the `Result` property of a `Task<TResult>`:
- The CLR wraps the exception in an AggregateException:

```
// 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;
    }
```

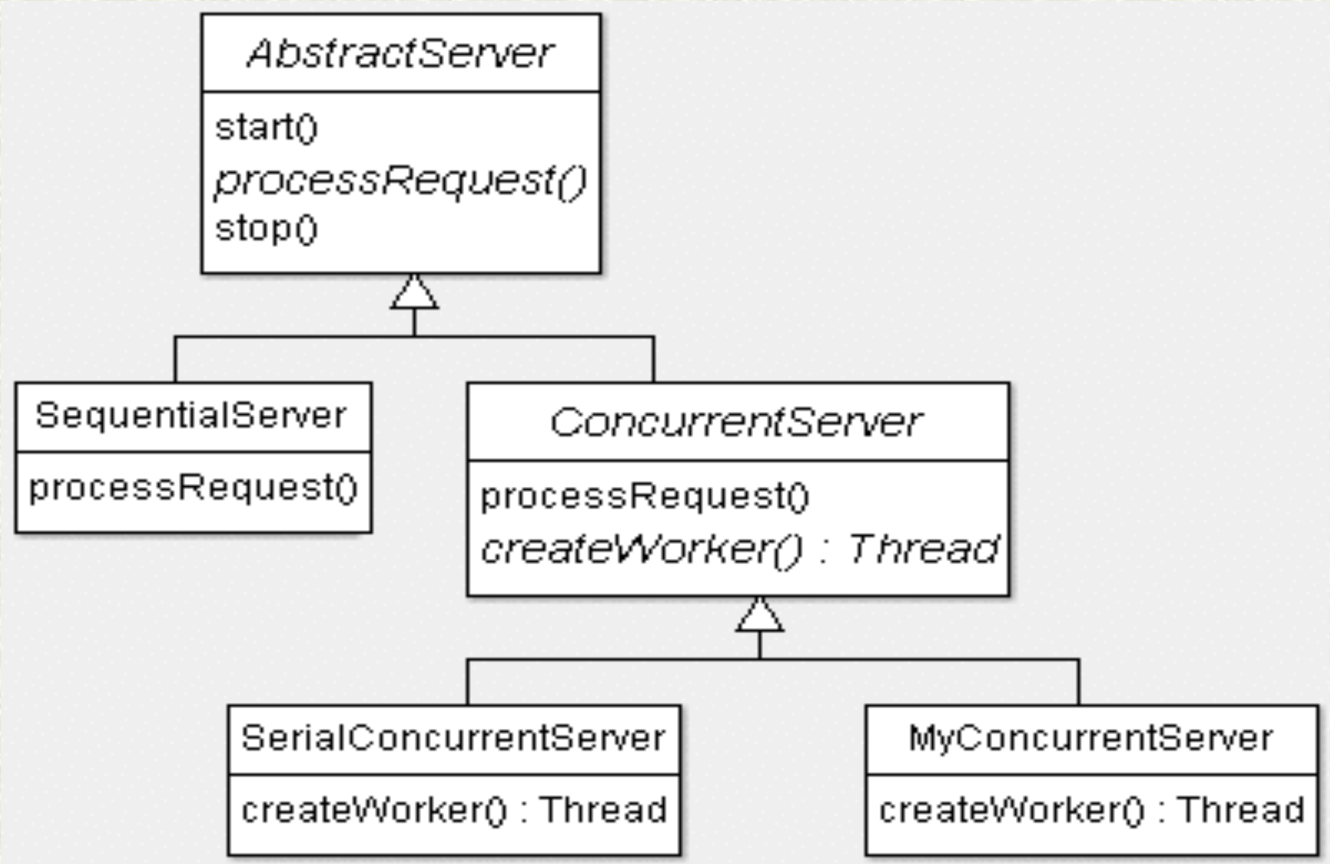
Example C#

- A simple client/server application:
  - The server waits for clients
  - The client connects to the server, and sends a message.
  - The server returns the message, written in uppercase, to which it has appended the time when it was received.

Template Pattern



Server Template



GUI Update

- The objects that make up a rich client are based primarily on Control in the case of Windows Forms.
- These objects have thread affinity, which means that only the thread that instantiates them can subsequently access their members.
- Violating this causes either unpredictable behavior, or an exception to be thrown.
- If you want to call a member on object X created on another thread Y, you must marshal the request to thread Y. You can do this by calling `Invoke` or `BeginInvoke` on the control.

GUI Update

- **`Invoke`** and **`BeginInvoke`** both accept a delegate, which references the method on the target control that you want to run.
  - **`Invoke`** works synchronously: the caller blocks until the marshal is complete.
  - **`BeginInvoke`** works asynchronously: the caller returns immediately and the marshaled request is queued up (using the same message queue that handles keyboard, mouse, and timer events).

GUI Update Example

```
//1. define a method for updating a ListBox
private void updateListBox(ListBox listBox, IList<String> newData){
    listBox.DataSource = null;
    listBox.DataSource = newData;
 }

//2. define a delegate to be called back by the GUI Thread
public delegate void UpdateListBoxCallback(ListBox list, IList<String>
data);

//3. in the other thread call like this:
list.Invoke(new UpdateListBoxCallback(this.updateListBox), new
Object[]{list, data});
or
list.BeginInvoke(new UpdateListBoxCallback(this.updateListBox), new
Object[]{list, data});
```

Networking in Java

- `java.net` package contains classes for TCP or UDP communication over the Internet.
- TCP: `Socket` and `ServerSocket` classes.
- UDP: `DatagramPacket`, `DatagramSocket`, and `MulticastSocket` classes.
- `InetAddress`: represents an Internet Protocol (IP) address. It has two subclasses:
  - `Inet4Address`: represents an IPv4 address (32 bits).
  - `Inet6Address`: represents an IPv6 address (128 bits).

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

- `InetSocketAddress (derived from SocketAddress abstract class)`: represents an IP address and a port number:

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

Networking in Java

- **`ServerSocket`** class represents a server socket that runs on the server and listens for incoming TCP connections.
- Constructors:

```
   public ServerSocket(int port) throws BindException, IOException
public ServerSocket(int port, int queueLength) throws BindException,
                                   IOException
public ServerSocket(int port,int queueLen, InetAddress bindAddress)
                                            throws 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
public void bind(SocketAddress endpoint, int queueLength) throws IOException

public Socket accept( ) throws IOException //blocks and waits for clients
//closes the server
public void close() throws IOException
```

Networking in 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 in Java

- `java.net.Socket` represents the class for performing client-side TCP operations.

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

- Methods:

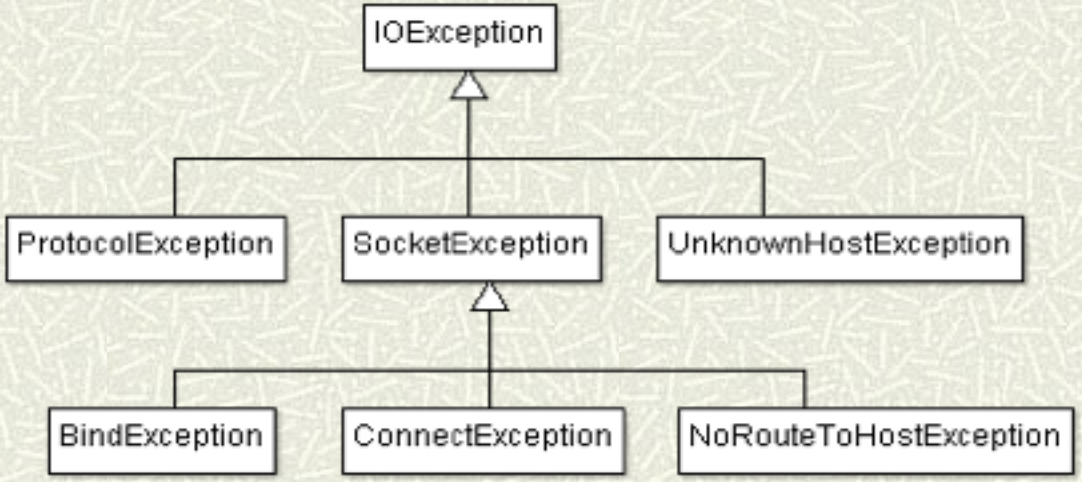
```
public int getPort()
public InputStream getInputStream() throws IOException
public OutputStream getOutputStream() throws IOException
public void close() throws IOException
```

Networking in Java

```
try{
   Socket connection=new Socket("172.30.106.5", 54321);
   //processing code

   connection.close();
}catch(UnknownHostException e){
   //…
}catch(IOException  e){
   //…
}
```

SocketExceptions



Example Java

- A simple client/server application:
  - The server waits for clients
  - The client connects to the server, and sends a message.
  - The server returns the message, written in uppercase to which it has appended the time when it was received.

Threading in Java

- In Java there are two ways to define a thread:
  - Subclass the `Thread` class (from `java.lang` package) and override the run method
  - Implement the `Runnable` interface (from `java.lang` package) and implement the `run` method.
- To create a thread, we use the `Thread` class

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

- To start a thread, we use the `start` method from `Thread` class:

```
public void start()
```

- Other methods:

```
public static void sleep(long millis)  throws InterruptedException;
public static void yield();   //temp. pause to allow others to execute
```

Threading in Java

```
//Subclass
public class WorkerC extends Thread{
    private Object data;
    public WorkerC(Object data){ this.data=data; }
    @Override
    public void run() {
        for(int i=0;i<26;i++)   System.out.printf("%s ", data);
}}
//Interface implementation
public class WorkerI implements Runnable{
    public void run() {
        for(int i=0; i<26;i++)
            System.out.printf("%d ",i);
    }
}
```

Threading in Java

```
//Starting the threads
public class StartThreads {
    public static void main(String[] args) {
        Thread t1=new WorkerC("a");
        Thread t2=new Thread(new WorkerI());
        t1.start();
        t2.start();
    }
}

//a 0 a 1 a a a a a a a a a a a a a a a a a a a a a a a a 2 3
   4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
```

Thread Synchronization

- `synchronized` statement

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

- An entire method can be synchronized:

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

- Yielding: a thread gives up control allowing other thread to execute:

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

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

Thread Synchronization

```
public class SynThreadEx implements Runnable{
    public void run() {
        synchronized (this){
            for(char c='a';c<='z';c++)
                System.out.print(c);
        }
    }
}
public class SynchronizedTest {
    public static void main(String[] args) {
        SynThreadEx tw=new SynThreadEx();
        Thread thread=new Thread(tw);
        thread.start();
        tw.run();
    }
}
//abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz
```

Threading in Java

- Sleeping: `sleep(long milliseconds)    //overloaded`
- Joining threads: `join(… ) //overloaded`
- Waiting on an object: `wait(… )`
- Notification (inherited from `Object` class): `notify( ) notifyAll( )`

Remark:

- `wait` and `notify`/`notifyAll` must be called on the same object that was used for synchronization

```
dou
ble[] array = new double[10000];
for (int i = 0; i < array.length; i++) {array[i] = Math.random();}
SortThread t = new SortThread(array);
t.start( );
try {
t.join( );
 System.out.println("Minimum: " + array[0]);
 System.out.println("Maximum: " + array[array.length-1]);
} catch (InterruptedException ex) {/*...*/}
```

Threading in Java

```
public class NotifyTest {
    public static void main(String[] args) {
        Worker worker=new Worker();
        Thread tw=new Thread(worker);
        tw.start();
        worker.run();
    }
    static  class Worker implements Runnable{
        public void run() {
            for(char c='a';c<='z';c++){
                synchronized (this){
                    System.out.print(c);
                    this.notify();
                    try {
                        this.wait();
                    } catch (InterruptedException e) { }
                }}
            synchronized (this){  this.notify();  }
        }
    }
}
```

Java Concurrency Utilities

- Java 5 introduced the concurrency utilities - extensible framework of high-performance threading utilities such as thread pools and blocking queues:
  - `java.util.concurrent`: Utility types that are often used in concurrent programming (i.e., executors)
  - `java.util.concurrent.atomic`: Utility classes that support lock-free thread-safe programming on single variables.
  - `java.util.concurrent.locks`: Utility types that lock and wait on conditions. Locking and waiting via these types is more performant and flexible than using `synchronized` or `wait`/`notify` mechanisms.

Java Tasks

- A Java task is an object whose class implements the **`java.lang.Runnable`** interface (a runnable task) or the **`java.util.concurrent.Callable`** interface (a callable task).

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

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

- **`Callable's call()`** method returns a value and can throw checked exceptions.

Executing Tasks

- Executor interface - execute Runnable tasks

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

- `ScheduledThreadPoolExecutor, ThreadPoolExecutor`
- Disadvantages:
  - It focuses exclusively on Runnable. As the `run()` method does not return a value, it is not easy for a runnable task to return a value to its caller.
  - It does not provide a way to track the progress of runnable tasks that are executing, cancel an executing runnable task, or determine when the runnable task finishes execution.
  - It cannot execute a collection of runnable tasks.
  - It does not provide a way for an application to shut down an executor.

ExecutorService

- `java.util.concurrent.ExecutorService` interface addresses the limitations of an `Executor`, and its implementation is typically a 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);
//other methods
}
```

- `ScheduledThreadPoolExecutor, ThreadPoolExecutor`
- Remark: The executor must be shut down after it completes; otherwise, the application might not end.

Future Interface

- A Future represents the result of an asynchronous computation.
- The result is known as a future because it typically will not be available until some moment in the future.
- It provides methods for canceling a task, for returning a task's value, and for determining whether or not the task has finished.

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

Executors

- The `Executors` utility class declares several class methods that return instances of various `ExecutorService` (and other kind of executors) implementations:
  - `newFixedThreadPool(int nThreads): ExecutorService`
  - `newSingleThreadExecutor(): ExecutorService`
  - `newCachedThreadPool():ExecutorService`
  - `newWorkStealingPool(): ExecutorService`

Example

```
ExecutorService executor = Executors.newSingleThreadExecutor();

Future<String[]> taskString = executor.submit(
                      ()-> new String[]{"Ana","are","mere"});
Future<Person> taskPerson=executor.submit(
                      ()->new Person("Ana",20));
try {
    String[] entries = taskString.get();
    Person p=taskPerson.get();

  } catch (InterruptedException|ExecutionException e) {
    e.printStackTrace();
  }
executor.shutdown();
```

Advanced Synchronization

- java.util.concurrent package
  - Countdown latch - causes one or more threads to wait at a “gate” until another thread opens this gate, at which point these other threads can continue. (`CountDownLatch`)
  - Cyclic barrier lets a set of threads wait for each other to reach a common barrier point. The barrier is cyclic because it can be reused after the waiting threads are released. (`CyclicBarrier`)
  - Semaphore maintains a set of permits for restricting the number of threads that can access a limited resource. A thread attempting to acquire a permit when no permits are available blocks until some other thread releases a permit. (`Semaphore`)
  - Phasers, Exchangers

CyclicBarrier Example

```
public class BarrierTest {
// static CyclicBarrier barrier=new CyclicBarrier(3,()->System.out.println());
 static CyclicBarrier barrier=new CyclicBarrier(3);

private static void letters(){
    for(char c='a'; c<='z';c++){
        System.out.print(c);
        try {
            barrier.await();
        } catch (InterruptedException|BrokenBarrierException e) {
            e.printStackTrace();
        }
    }
}

public static void main(String[] args) {
    ExecutorService executor= Executors.newFixedThreadPool(3);
    executor.execute(()->letters());
    executor.execute(()->letters());
    executor.execute(()->letters());

    executor.shutdown();

}
}
```

Concurrent Collections

- It contains utility classes useful in concurrent programming.
- Starting with version 1.5
- **`BlockingQueue`** interface :
  - It is a queue that additionally supports operations that wait for the queue to become non-empty when retrieving an element, and wait for space to become available in the queue when storing an element.
  - BlockingQueue implementations are designed to be used primarily for producer-consumer queues.
  - **`ArrayBlockingQueue`**, **`LinkedBlockingQueue`**, **`PriorityBlockingQueue`**, etc.
- **`BlockingDeque`** interface:
  - It extends **`BlockingQueue`** to support both FIFO and LIFO (stack-based) operations.
  - **`LinkedBlockingDeque`**
- **`ConcurrentMap`** interface
  - It is a subinterface of **`java.util.Map`** that declares additional indivisible **`putIfAbsent(), remove(), and replace()`** methods.
  - **`ConcurrentHashMap, ConcurrentSkipListMap`**

BlockingQueue example

- Simple Producer-Consumer without BlockingQueue

```
//both threads have a reference to messages
//initialization
private List<String> messages=new ArrayList<String>();
//Producer
messages.add(message);
try {
   Thread.sleep(1000);
} catch (InterruptedException e) { e.printStackTrace(); }
synchronized (messages){
     messages.notify();
}
//Consumer
synchronized (messages){
   messages.wait();
}
String message = messages.remove(0);
```

BlockingQueue example

- Simple Producer-Consumer with BlockingQueue

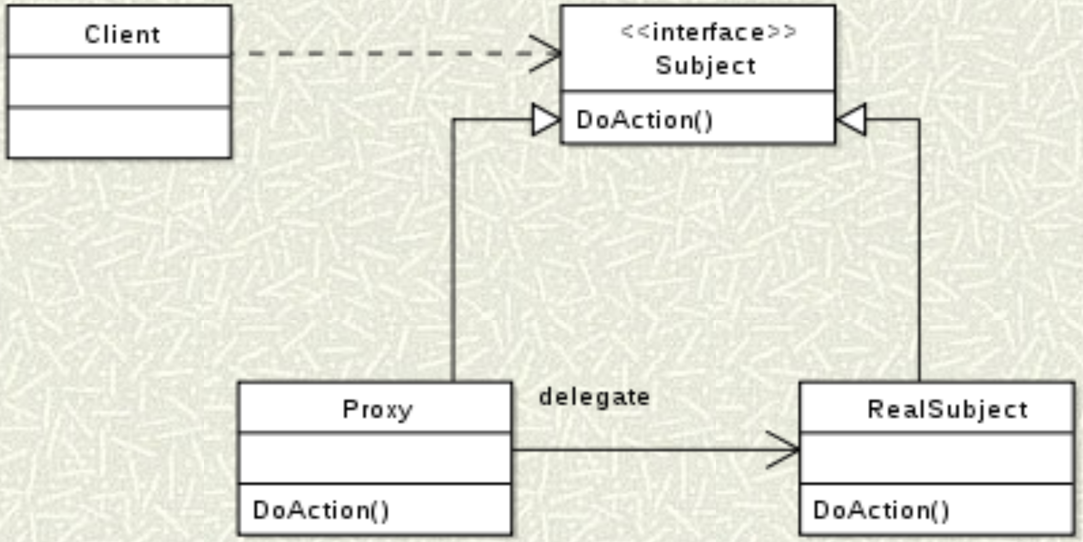
```
//both threads have a reference to messages
//initialization
private BlockingQueue<String> messages=new
   LinkedBlockingQueue<String>();


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


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

Proxy Design Pattern

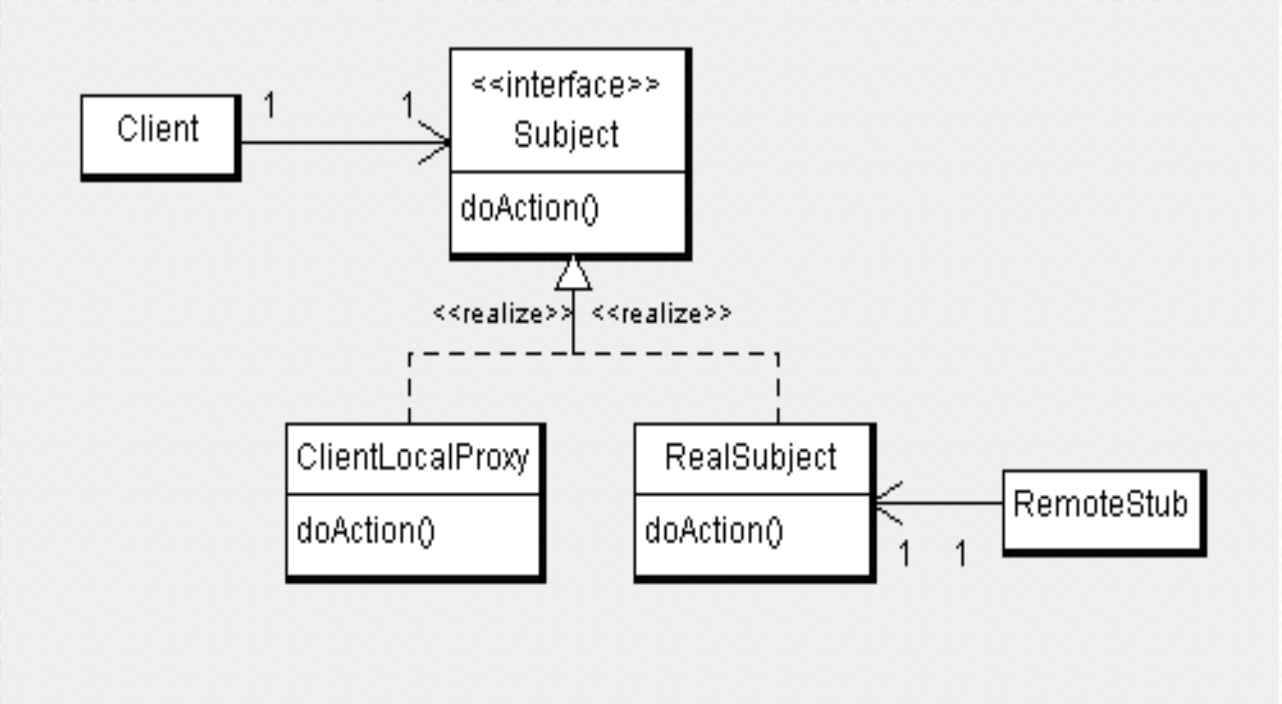
- A proxy, in its most general form, is a class functioning as an interface to something else.
- The proxy could interface to anything: a network connection, a large object in memory, a file, or some other resource that is expensive or impossible to duplicate.



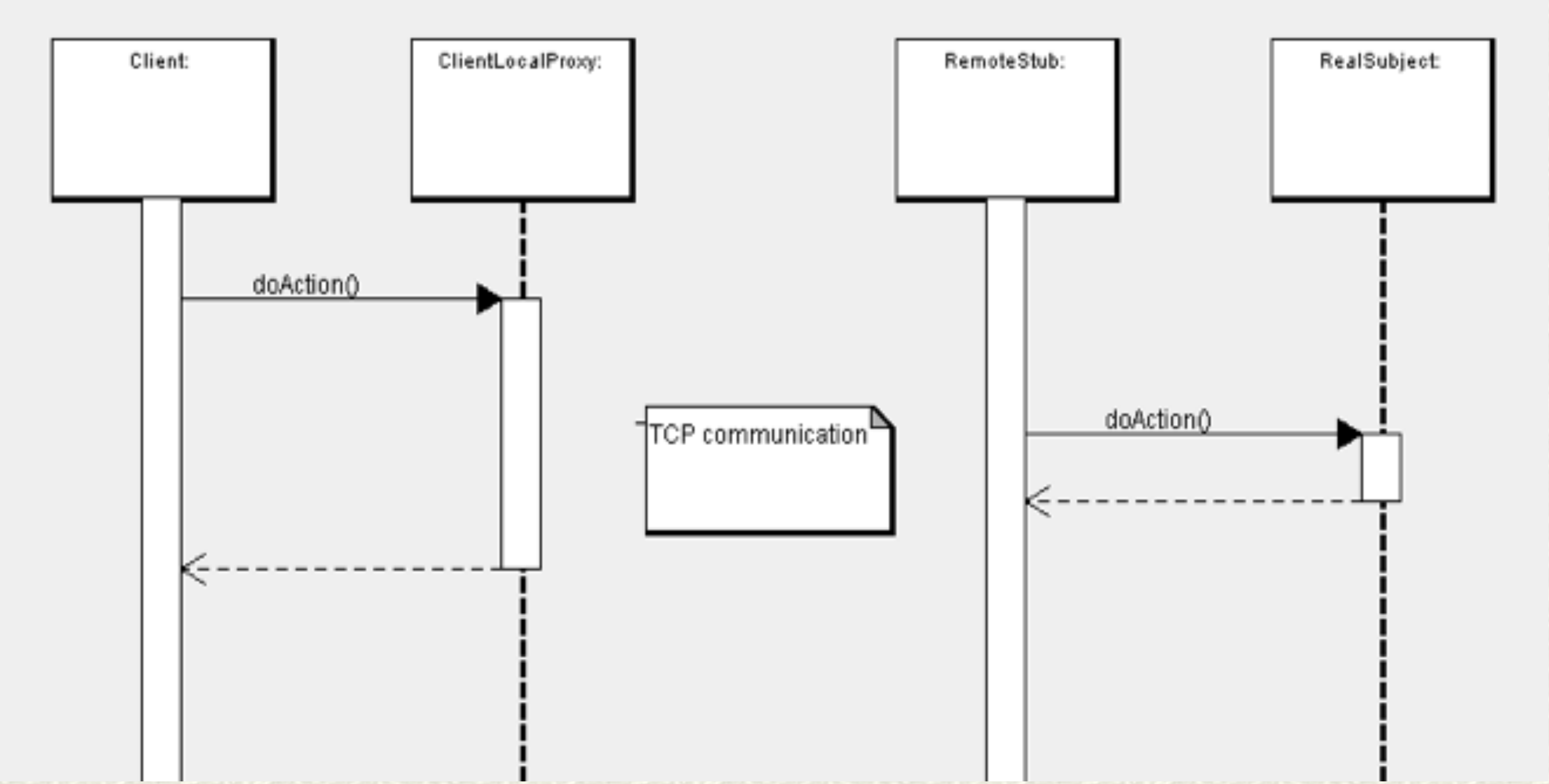
Remote Proxy Design Pattern

- Remote Proxy provides a local representative for an object in a different address space.
- Remote proxies are responsible for encoding a request and its arguments and for sending the encoded request to the real subject, or another proxy in a different name space.
- The client thinks that it talks with the remote object, but there is the proxy between them.
- The Proxy translates the client’s queries in remote calls, gets the results of the query from remote object and forwards them to the client.

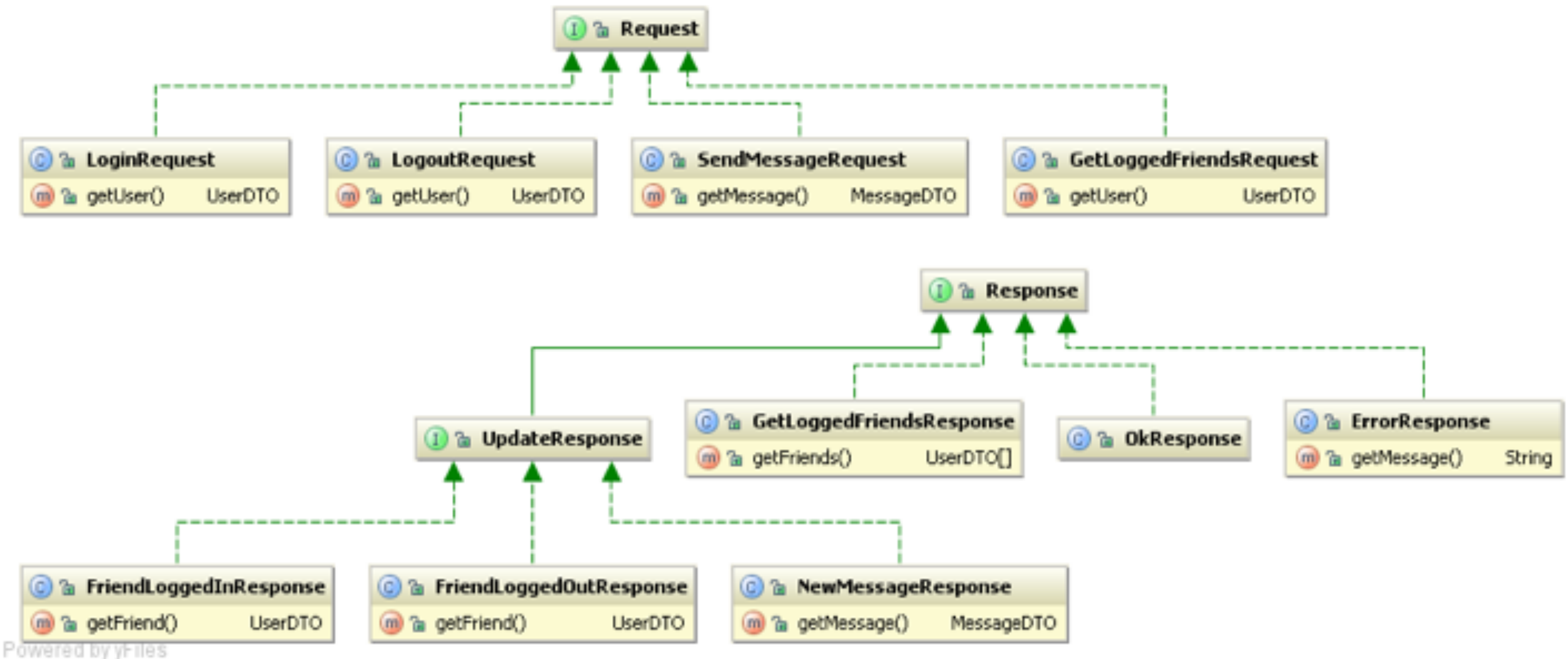
Remote Proxy Design Pattern



Remote Proxy Design Pattern



Chat Case Study
Object Protocol



Chat Case Study
Rpc Protocol