Advanced Programming 2
Recitation 3 – Multithreading and Sockets

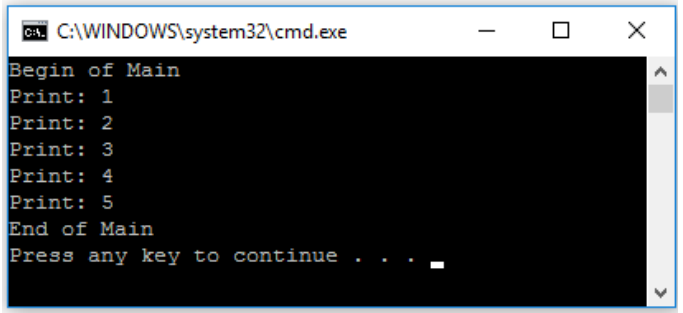
Roi Yehoshua
2017

Multithreading

Threads in C# (The Old Way)

- Creating an instance of the Thread class creates a new managed thtead
- The Thread class has constructors that take a ThreadStart delegate or a ParameterizedThreadStart delegate
- The delegate wraps the method that is invoked by the new thread when you call the Start method

```
static void Print()
{
    for (int i = 1; i <= 5; i++)
    {
        Console.WriteLine("Print: {0}", i);
        Thread.Sleep(1000);
    }
}
static void Main(string[] args)
{
    Thread thread = new Thread(new ThreadStart(Print));

    Console.WriteLine("Begin of Main");
    thread.Start();
    thread.Join();
    Console.WriteLine("End of Main");
}
```



Thread Pool

- Thread pool is where a number of threads are created to perform a number of tasks, which are usually organized in a queue
- Typically, there are many more tasks than threads
- Advantages:
  - Creating and destroying threads is time consuming. Reusing an existing thread saves that time.
  - Control the number of threads running - helps avoid memory running out and resource thrashing
- When not to use thread pool
  - You require a thread to have a particular priority.
  - You have large number of tasks that cause the threads to block for long periods of time, thus preventing other tasks from starting

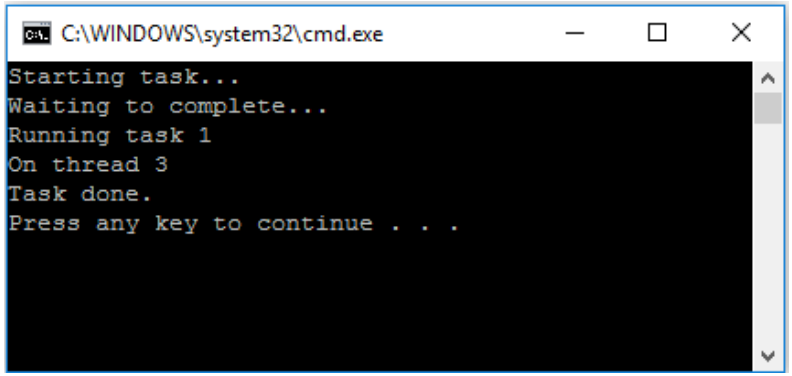
The Task Parallel Library (TPL)

- .NET 4 introduces the **`Task`** class (and friends)
  - This class is used to create and manage tasks
- A task is a logical unit of work that is superficially similar to a work item queued to the thread pool
- Advantages over the raw thread pool
  - There is an object to work with
  - Can wait for a task to finish
  - Can return a result
  - Can combine tasks in interesting ways
  - Can customize the way tasks are scheduled

Creating a Task With No Result

- Create an instance of **Task**
  - Pass a delegate to be called when the task is started
  - **Action** or **Action<object>**
- Call the **Start** method
- Can wait for task to complete with **Wait** method

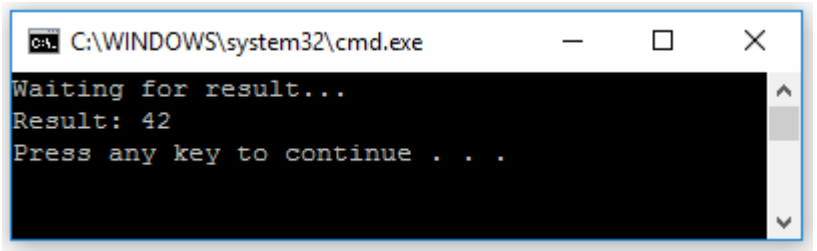
```
Task t = new Task(() => {
    Console.WriteLine("Running task {0}", Task.CurrentId);
    Console.WriteLine("On thread {0}",
Thread.CurrentThread.ManagedThreadId);
    Thread.Sleep(1000);
});
Console.WriteLine("Starting task...");
t.Start();
Console.WriteLine("Waiting to complete...");
t.Wait();
Console.WriteLine("Task done.");
```



Creating a Task that Returns a Result

- Create an instance of **Task<T>**
  - Where T is the type of the result
  - C'tor accepts a **Func<T>** or **Func<object,T>**
- To get the result back, use the **Result** property
  - Blocks if the task is not complete

```
Task<int> t = new Task<int>(() => {
    Thread.Sleep(1000);
    return 42;
});
t.Start();
Console.WriteLine("Waiting for result...");
Console.WriteLine("Result: {0}", t.Result);
```



Sockets

Basic TCP Server in C#

The using statement ensures that Dispose is called on the object even if an exception occurs

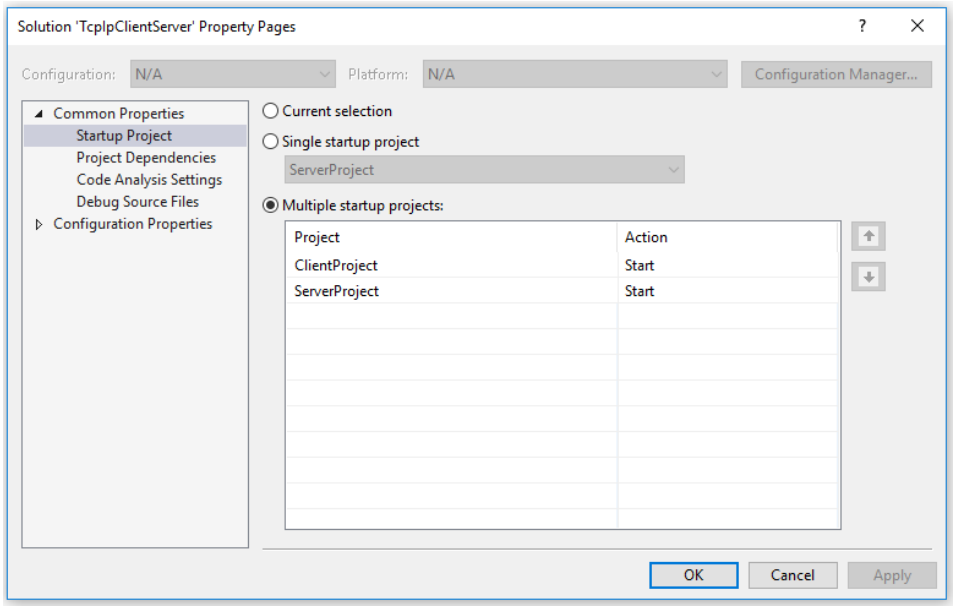
```
IPEndPoint ep = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 8000);
TcpListener listener = new TcpListener(ep);
listener.Start();

Console.WriteLine("Waiting for client connections...");
TcpClient client = listener.AcceptTcpClient();
Console.WriteLine("Client connected");

using (NetworkStream stream = client.GetStream())
using (BinaryReader reader = new BinaryReader(stream))
using (BinaryWriter writer = new BinaryWriter(stream))
{
    Console.WriteLine("Waiting for a number");
    int num = reader.ReadInt32();
    Console.WriteLine("Number accepted");
    num *= 2;
    writer.Write(num);
}

client.Close();
listener.Stop();
```

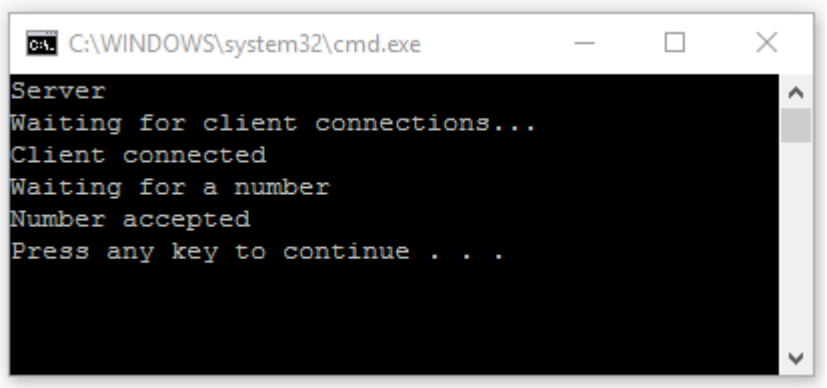
Basic TCP Client in C#

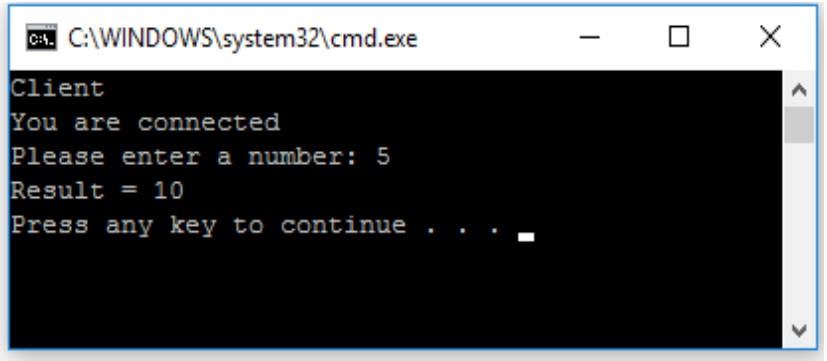
```
IPEndPoint ep = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 8000);
TcpClient client = new TcpClient();
client.Connect(ep);
Console.WriteLine("You are connected");

using (NetworkStream stream = client.GetStream())
using (BinaryReader reader = new BinaryReader(stream))
using (BinaryWriter writer = new BinaryWriter(stream))
{
    // Send data to server
    Console.Write("Please enter a number: ");
    int num = int.Parse(Console.ReadLine());
    writer.Write(num);

    // Get result from server
    int result = reader.ReadInt32();
    Console.WriteLine("Result = {0}", result);
}
client.Close();
```

Basic TCP Client in C#

- You can run both server and client project simultaneously by choosing Multiple Startup Projects in the solution properties







TCP Server that Handles Multiple Clients

```
class Server
{
    private int port;
    private TcpListener listener;
    private IClientHandler ch;

    public Server(int port, IClientHandler ch)
    {
        this.port = port;
        this.ch = ch;
    }

    public void Start()
    {
        IPEndPoint ep = new
IPEndPoint(IPAddress.Parse("127.0.0.1"), port);
        listener = new TcpListener(ep);

        listener.Start();
        Console.WriteLine("Waiting for connections...");
```

```
        Task task = new Task(() => {
            while (true)
            {
                try {
                    TcpClient client = listener.AcceptTcpClient();
                    Console.WriteLine("Got new connection");
                    ch.HandleClient(client);
                }
                catch (SocketException) {
                    break;
                }
            }
            Console.WriteLine("Server stopped");
        });
        task.Start();
    }

    public void Stop()
    {
        listener.Stop();
    }
}
```

ClientHandler

```
public interface IClientHandler
{
    void HandleClient(TcpClient client);
}
```

```
class ClientHandler : IClientHandler
{
    public void HandleClient(TcpClient client)
    {
        new Task(() =>
        {
            using (NetworkStream stream = client.GetStream())
            using (StreamReader reader = new StreamReader(stream))
            using (StreamWriter writer = new StreamWriter(stream))
            {
                string commandLine = reader.ReadLine();
                Console.WriteLine("Got command: {0}", commandLine);
                string result = ExecuteCommand(commandLine, client);
                writer.Write(result);
            }
            client.Close();
        }).Start();
    }
}
```

JSON

JSON

- Lightweight data-interchange format
  - Compared to XML
- Simple format
  - Easy for humans to read and write
  - Easy for machines to parse and generate
- JSON is a text format
  - Programming language independent
  - Conventions familiar to programmers of the C-family of languages, including C# and JavaScript

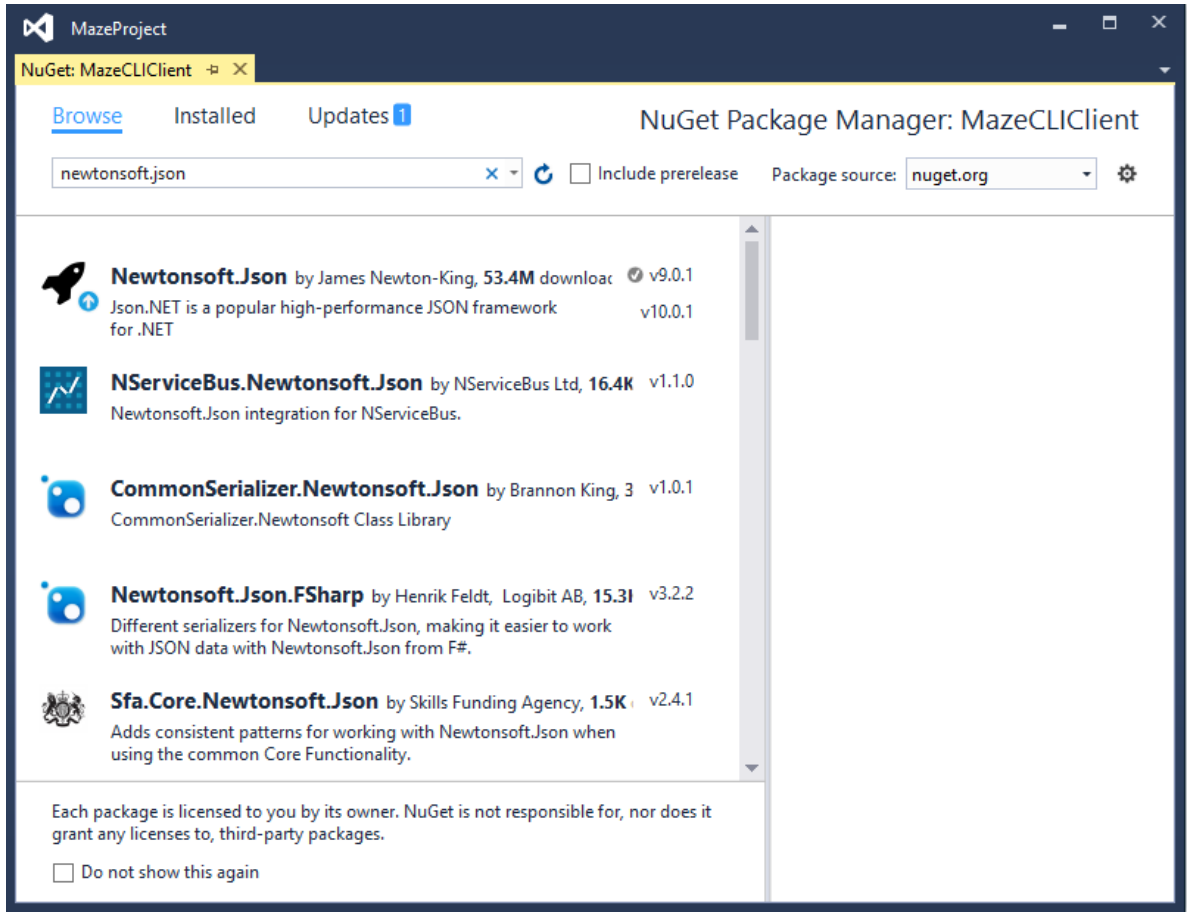

```
{ "users":[
            {
                "firstName":"Ray",
                "lastName":"Villalobos",
                "joined": {
                    "month":"January",
                    "day":12,
                    "year":2012
                }
            },
            {
                "firstName":"John",
                "lastName":"Jones",
                "joined": {
                    "month":"April",
                    "day":28,
                    "year":2010
                }
            }
    ]}
```

JSON .Net

- A popular high-performance JSON framework for .NET
- Create, parse, query and modify JSON using Json.NET's JObject, JArray and JValue objects
- Download from http://www.newtonsoft.com/json
  - or via NuGet Package manager

NuGet Package Manager

- NuGet is the package manager for the Microsoft development platform including .NET.
- The NuGet Gallery (nuget.org) is the central package repository used by all package authors and consumers.
- You can install new NuGet packages in your project via the project properties



Converting from object to JSON and vice versa

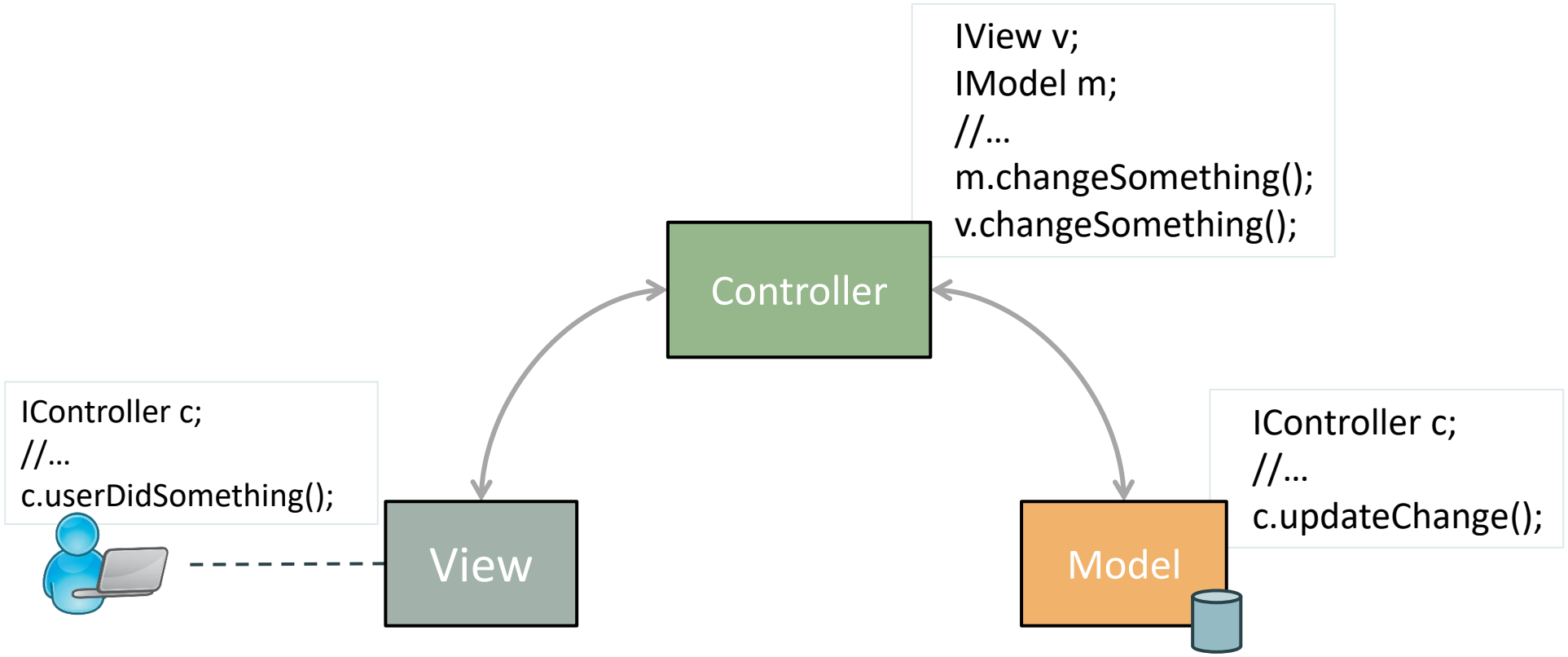
```
public class Maze
{
    public string Name { get; set; }
    public int Rows { get; private set; }
    public int Cols { get; private set; }
    public Position InitialPos { get; set; }
    public Position GoalPos { get; set; }
    private CellType[,] cells;

    public string ToJSON()
    {
        JObject mazeObj = new JObject();
        mazeObj["Name"] = Name;
        mazeObj["Rows"] = Rows;
        mazeObj["Cols"] = Cols;

        JObject startObj = new JObject();
        startObj["Row"] = InitialPos.Row;
        startObj["Col"] = InitialPos.Col;
        mazeObj["Start"] = startObj;
        ...
        return mazeObj.ToString();
    }
```

```
public static Maze FromJSON(string str)
{
    Maze maze = new Maze();

    JObject mazeObj = JObject.Parse(str);
    maze.Name = (string)mazeObj["Name"];
    maze.Rows = (int)mazeObj["Rows"];
    maze.Cols = (int)mazeObj["Cols"];

    maze.InitialPos = new
Position((int)mazeObj["Start"]["Row"],
(int)mazeObj["Start"]["Col"]);
    ...
    return maze;
}
```

MVC

MVC

- The separation of the **View** and the **Model** layers with a **Controller** layer

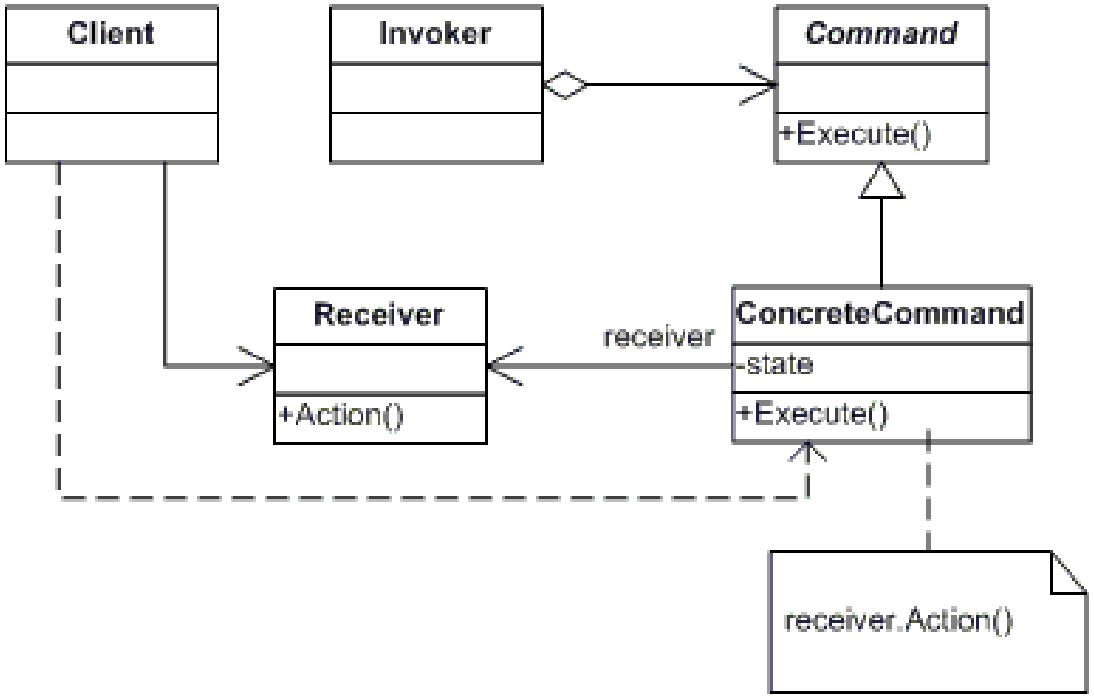


MVC In Server Implementation

- View – the client handler
  - When a new message arrives - the view will ask the controller to handle it
- Controller - handles the request from the view
  - Uses the Model to perform data tasks and operation.
- Model – runs the algorithms and performs data related operations
  - Notifies the controller when a task is done

Command Design Pattern

- Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations



Command Example

- Command interface:

```
interface ICommand
{
    string Execute(string[] args, TcpClient client = null);
}
```

Some commands need the TcpClient object to send messages back to the client

- Sample command:

```
class GenerateMazeCommand : ICommand
{
    private IModel model;
    public GenerateMazeCommand(IModel model)
    {
        this.model = model;
    }
    public string Execute(string[] args, TcpClient client)
    {
        string name = args[0];
        int rows = int.Parse(args[1]);
        int cols = int.Parse(args[2]);

        Maze maze = model.GenerateMaze(name, rows, cols);
        return maze.ToJSON();
    }
}
```

Controller Example

Holds the hash map of the commands

Split the command line to the command name and arguments

```
class Controller
{
    private Dictionary<string, ICommand> commands;
    private IModel model;

    public Controller()
    {
        model = new Model();
        commands = new Dictionary<string, ICommand>();
        commands.Add("generate", new GenerateMazeCommand(model));
        // more commands...
    }

    public string ExecuteCommand(string commandLine, TcpClient client)
    {
        string[] arr = commandLine.Split(' ');
        string commandKey = arr[0];
        if (!commands.ContainsKey(commandKey))
            return "Command not found";
        string[] args = arr.Skip(1).ToArray();
        ICommand command = commands[commandKey];
        return command.Execute(args, client);
    }
}
```