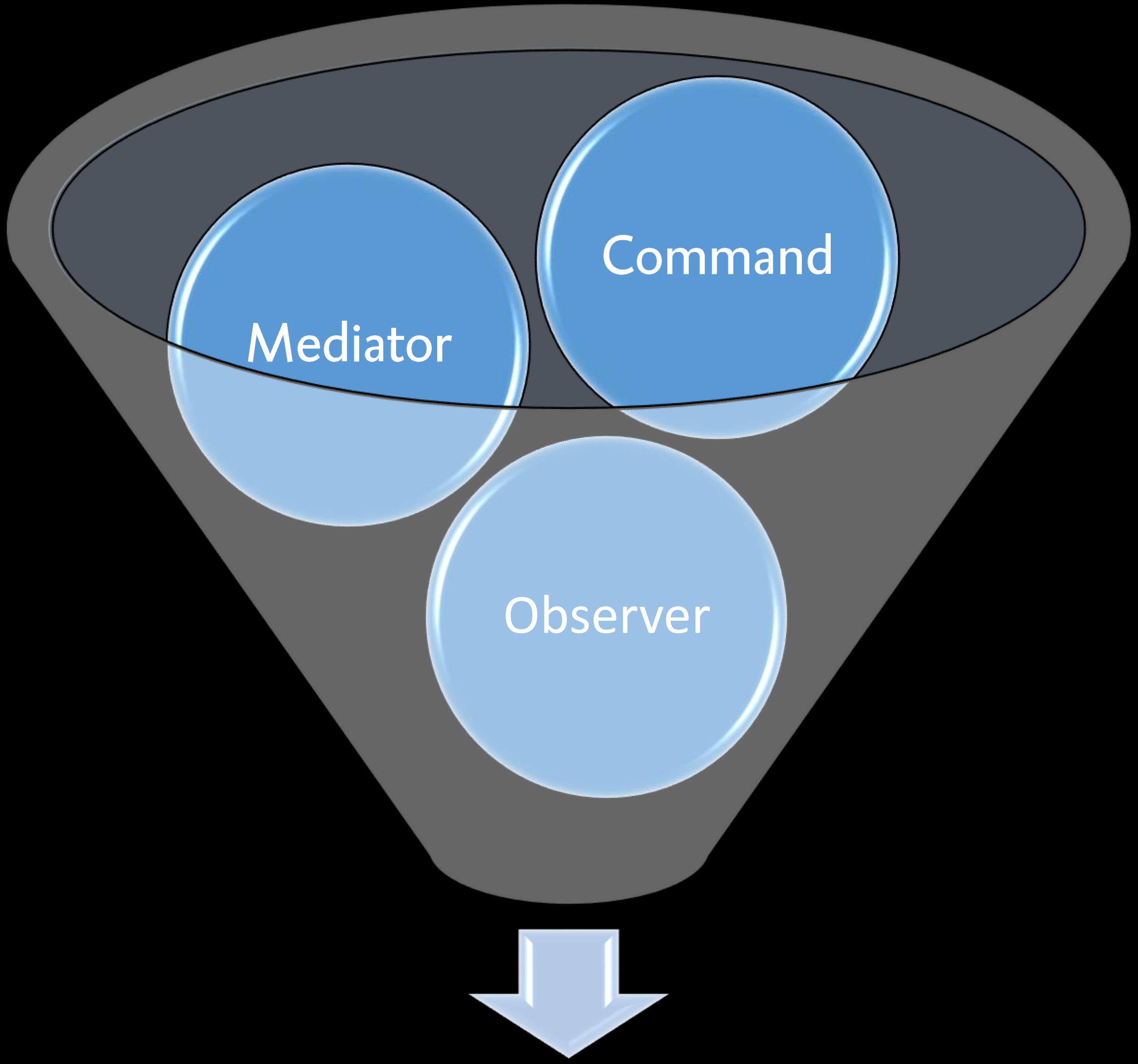
Event Brokers

Dmitri Nesteruk
@dnesteruk



Event Broker

```
class BankAccount
{
  private Guid id = Guid.NewGuid();
  private int balance;

  public void Deposit(int amt)
  {
    this.balance += amt;
  }
}
```

Why Commands?

When you run the code

```
var acc = new Account();
acc.Deposit(1000);
```

There is no record when and what account was created

There is no record of changes to account

Command

An instruction to do something

Typically a mutating operation

Who handles the command?

- Command is self-executing (serializable unit of work)
- The affected object
- A dedicated command processor (this can also handle the creation command)

One command can be processed by many processors simultaneously

```
class DepositCommand
{
  public Guid AccountId;
  public int Amount;
}
class CommandProcessor
{
  public void Process(DepositCommand cmd)
  {
    accounts
      .FindById(cmd.AccountId)
      .Deposit(cmd.Amount);
  }
}
```

Command Interface

A command can also ‘execute itself’

In other words, there’s no command processor, each command does its own thing

This lets us imbue the command with additional responsibility

- Being able to undo itself
- Storing a flag indicating whether it succeeded

```
public abstract class Command
{
  public abstract void Call();
  public abstract void Undo();
  public bool Success;
}
```

We can only
undo command
if it succeeded!

Composite + Command = Macro

Transfer between two accounts is two operations:

- Withdraw from one account
- Deposit to another account

A simple list that we can use to

- `Call()` each command; or
- `Undo()` each command (in reverse order)

If the first operation fails, the second must not be executed (an example of CEP)

DEMO

CompositeCommand.cs

Mediator

A central binding glue that helps objects communicate without being aware of one another

This can happen

- In-process/out of process
- Synchronously/asynchronously

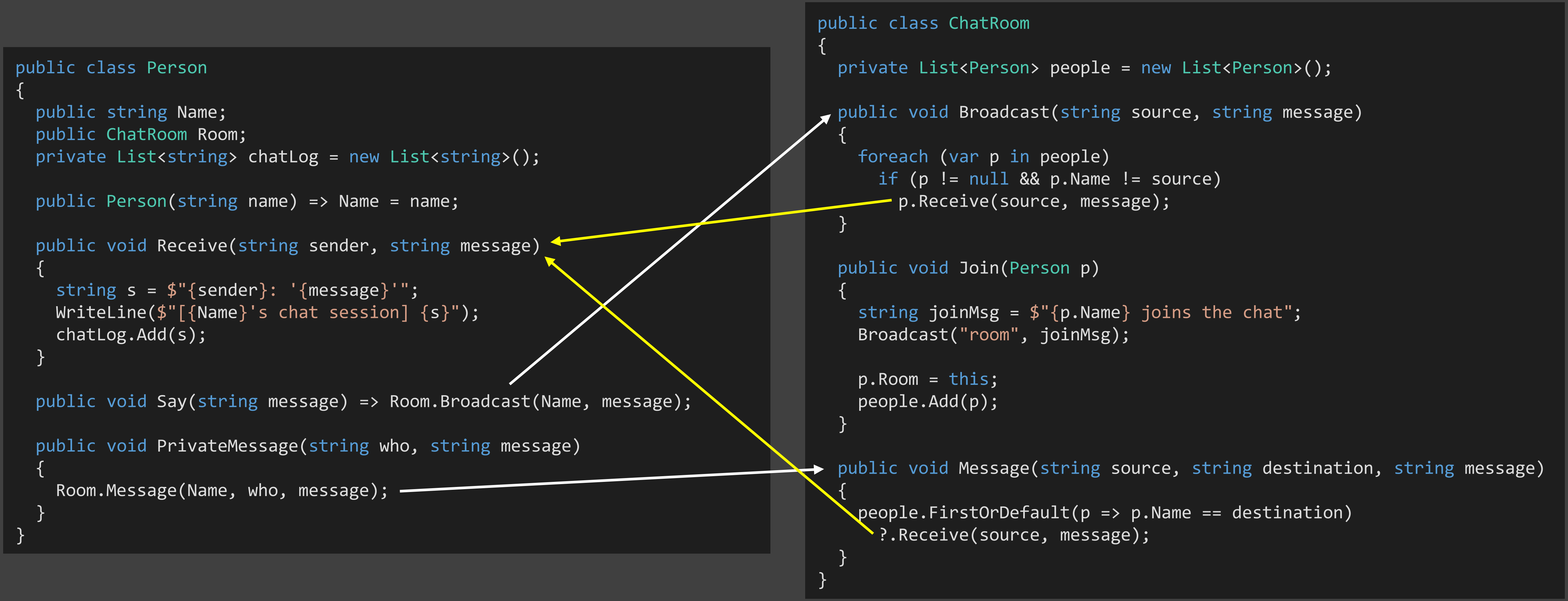
Chat Room

Several people chat with one another

No direct references, but

Each references a ChatRoom

People leaving and entering does not break the system



```csharp
public class Person
{
  public string Name;
  public ChatRoom Room;
  private List<string> chatLog = new List<string>();

  public Person(string name) => Name = name;

  public void Receive(string sender, string message)
  {
    string s = $"{sender}: '{message}'";
    WriteLine($"[{Name}'s chat session] {s}");
    chatLog.Add(s);
  }

  public void Say(string message) => Room.Broadcast(Name, message);

  public void PrivateMessage(string who, string message)
  {
    Room.Message(Name, who, message);
  }
}
```

```csharp
public class ChatRoom
{
  private List<Person> people = new List<Person>();

  public void Broadcast(string source, string message)
  {
    foreach (var p in people)
      if (p != null && p.Name != source)
        p.Receive(source, message);
  }

  public void Join(Person p)
  {
    string joinMsg = $"{p.Name} joins the chat";
    Broadcast("room", joinMsg);

    p.Room = this;
    people.Add(p);
  }

  public void Message(string source, string destination, string message)
  {
    people.FirstOrDefault(p => p.Name == destination)
      ?.Receive(source, message);
  }
}
```

Mediator Propagation

Each component references and uses the mediator
Can be passed manually or using DI
Delegate factories are useful!
Instead of

```
    Foo(string name, Mediator m) { ... }
```

We write

```
    delegate Foo Factory(string name);
    Foo(string name, Mediator m) { ... }
```

and then use the factory

Observer

One component needs to be notified when something happens in another component

Imperative:

- .NET events (`+=`, `-=`): not disposable, memory leaks
- Subscribing on an `IObservable<T>`

Declarative:

- Interfaces: `ISend<TEvent>`, `IHandle<TEvent>`
- Attributes: [Publishes(typeof(…))], [Handles(…)]

```csharp
public interface IEvent {}

public interface ISend<TEvent> where TEvent : IEvent
{
  event EventHandler<TEvent> Sender;
}

public interface IHandle<TEvent> where TEvent : IEvent
{
  void Handle(object sender, TEvent args);
}
```

```
public class ButtonPressedEvent : IEvent
{
  public int NumberOfClicks;
}

public class Button : ISend<ButtonPressedEvent>
{
  public event EventHandler<ButtonPressedEvent> Sender;

  public void Fire(int clicks)
  {
    Sender?.Invoke(this, new ButtonPressedEvent
    {
      NumberOfClicks = clicks
    });
  }
}

public class Logging : IHandle<ButtonPressedEvent>
{
  public void Handle(object sender, ButtonPressedEvent args)
  {
    Console.WriteLine(
      $"Button clicked {args.NumberOfClicks} times");
  }
}
```

subsciption happens
automatically in
IoC container

DEMO

ContainerWireup.cs

Problems with Declarative Subscriptions

An event publisher is created when some subscribers already exist

- How do we find and auto-subscribe them?

A component is destroyed

- But doesn’t get GCd → memory leak!

Conclusion: need a separate mechanism for disposable subscriptions

```
public interface IObservable<out T>
{
  IDisposable Subscribe(IObserver<T> observer);
}
```

Manufacturing Subscriptions

Two implementation choices

1) Use a prepackaged observable (e.g., `Subject<T>`) — instance members not guaranteed to be thread safe, etc.
2) Roll your own subscription mechanism: can introduce neat threading safety, async/await and other magic

DEMO

ObserverInterfaces.cs

Putting it all together

An event broker is…

A mediator

- Injected into each component via DI

That routes commands (and other messages)

Implements `IObservable<T>`

- Allows Rx operators on the message stream

DEMO

MediatorWithRx.cs

Message Taxonomy

A command is an example of a message being sent

Other message examples include:

Queries (give me some data; immutable)

- Entities that process commands and queries can be segregated
- CQRS = Command Query Responsibility Segregation

Events (just letting you know this happened)

- Current state of object = sequence of events
- Can reconstruct any past state and how we got there
- Event Sourcing

Queries

A query is a request for a value (result)

Sometimes the returned value depends on other (lifetime-bound) aspects of the system

In-process query:

- Put a query object on the bus
- Handlers can set or modify query result

Property getters/setters can act as proxies for command/query routing

DEMO

BrokerChain.cs

Next Steps

Thread safety, async/await

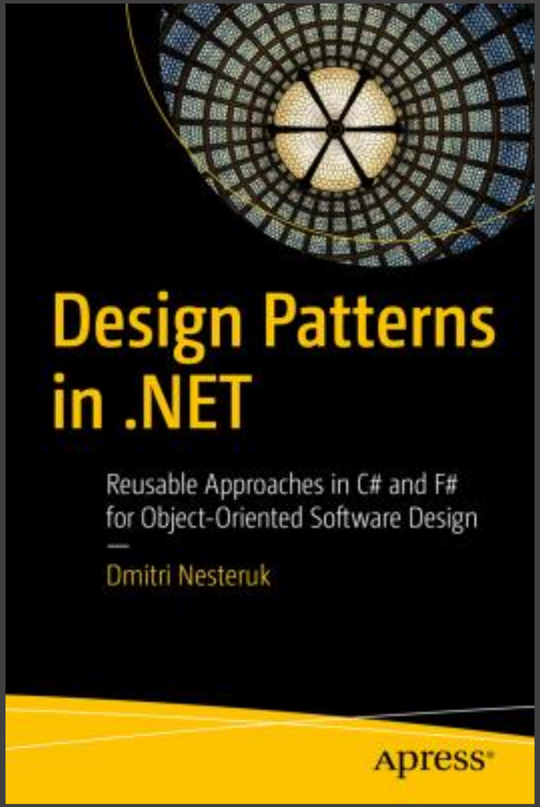
Inclusion of events + event store

Aggregate roots defined as sequence of events (rebuild, temporal query, event replay)

That's it!

Questions? Answers? Hate mail? @dnesteruk

Design Patterns in .NET online course at http://bit.ly/2p3aZww


Design Patterns in .NET
Reusable Approaches in C# and F# for Object-Oriented Software Design
Dmitri Nesteruk
Apress