Module 14: "Iterator"

Agenda

- Introductory Example: Playing Cards
- Challenges
- `IEnumerable`
- `IEnumerable<T>`
- Implementing the Iterator Pattern
- Pattern: Iterator
- Overview of Iterator Pattern
- Iterator Pattern and LINQ Queries

Introductory Example: Playing Cards

```
class Deck
{
    private List<Card> _cards;

    public Deck()
    {
        _cards = new List<Card>();
        ...
    }
    public Card Deal() { ... }
    public void Shuffle() { ... }
}
```

```
struct Card : IComparable
{
    public Suit Suit { get; }
    public Rank Rank { get; }

    public Card(
        Suit suit, Rank rank )
    {
        Suit = suit;
        Rank = rank;
    }
}
```

```
Deck deck = new Deck();
deck.Shuffle();
Card card = deck.Deal();
Console.WriteLine( card );
```

Challenges

- How can clients iterate through the elements of the Deck without internal state being directly exposed?
- How do we perform LINQ queries on the Card elements in Deck?

Pattern: Iterator

- *Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.*

- Outline
  - Facilitate iteration through a read-only collection of elements of the aggregate using **`foreach`**
  - Facilitate LINQ for querying elements of the aggregate
  - Implement **`IEnumerable<T>`** for element type T

- Origin: Gang of Four (+ extended by .NET)

IEnumerable

- .NET has the **IEnumerable** interface built in.

```
namespace System.Collections
{
    interface IEnumerable
    {
        IEnumerator GetEnumerator();
    }
}
```

```
interface IEnumerator
{
    object Current { get; }
    bool MoveNext();
    void Reset();
}
```

- Arrays and collection classes all implement this interface

Implementing **IEnumerable**

- You can implement the Iterator Pattern by implementing **IEnumerable** in your own types

```
class Deck : IEnumerable
{
    private List<Card> _cards;

    public Deck() { ... }
    public Card Deal() { ... }
    public void Shuffle() { ... }

    public IEnumerator GetEnumerator() { ... }
}
```

Iterator Syntax in C#

- C# provides powerful mechanisms for easy creation of iterator methods

```
public IEnumerator GetEnumerator()
{
    int i = 0;
    while (true)
    {
        yield return _cards[i++];
        if (i == _cards.Count)
        {
            yield break;
        }
    }
}
```

IEnumerable<T>

- Class must implement the **IEnumerable<T>** interface for LINQ to work.

```
namespace System.Collections.Generic
{
    interface IEnumerable<out T> : IEnumerable
    {
        IEnumerator<T> GetEnumerator();
    }
}
```

```
interface IEnumerator<T>
{
    T Current { get; }
    bool MoveNext();
    void Reset();
}
```

- But…

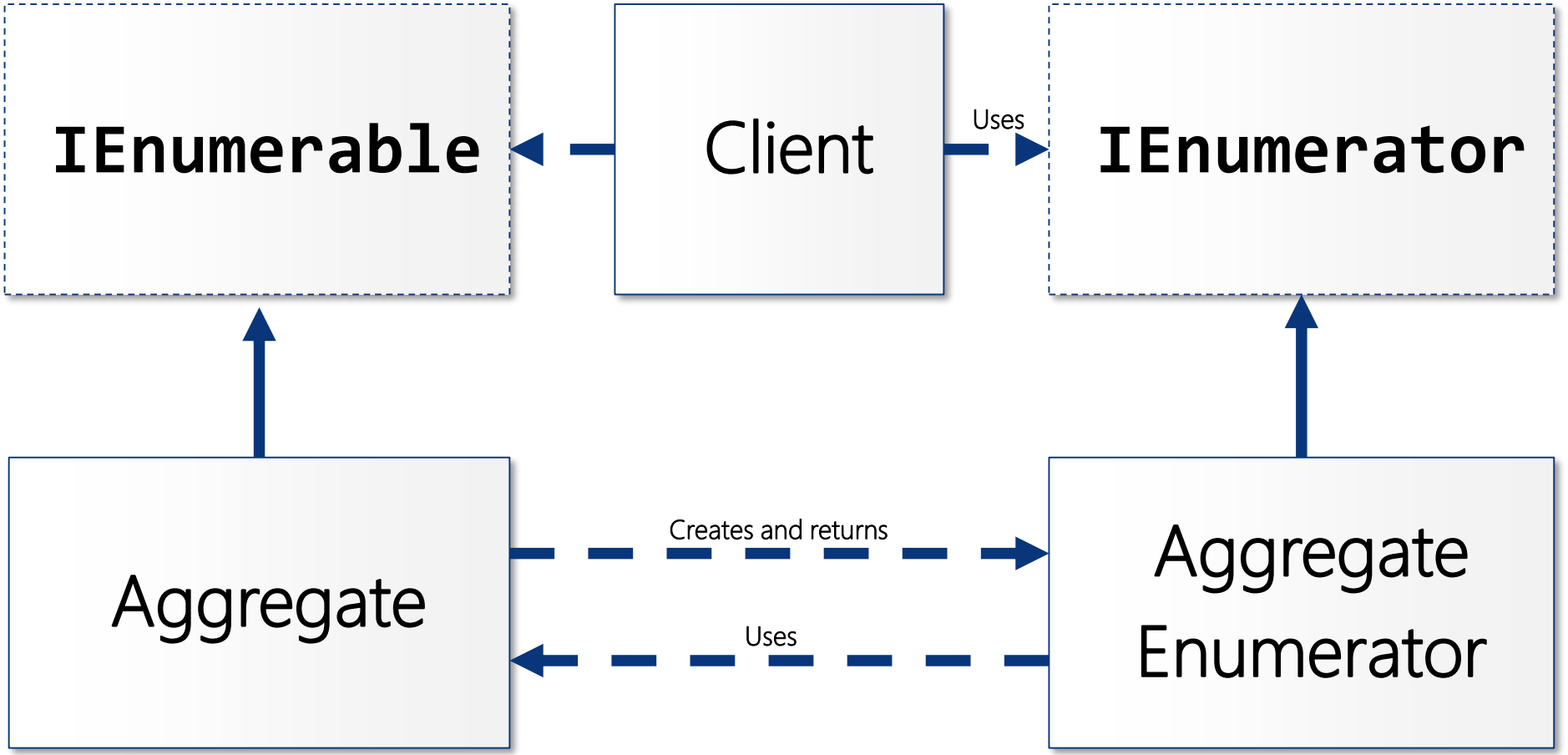
Background: Explicit Interface Implementation

```
interface IArtist
{
    void Draw();
}
```

```
interface IGunslinger
{
    void Draw();
}
```

```
class ArtisticCowboy : IArtist, IGunslinger
{
    public void Draw()
    {
        Console.WriteLine( "Swinging brush, painting canvas..." );
    }
    void IGunslinger.Draw()
    {
        Console.WriteLine("Drawing Colt .45 from gun belt...");
    }
}
```

Overview of Iterator Pattern



Overview of Iterator Pattern

- Client
  - Asks Aggregate for Aggregate Enumerator
  - Uses Aggregate Enumerator for traversing elements
- Aggregate
  - Contains elements to be iterated
  - Creates Aggregate Enumerator and returns it to Client
- Aggregate Enumerator
  - Contains method for iterating the elements of the Aggregate
  - References the elements of the Aggregate when needed by Client

Iterator Pattern and LINQ Queries

- Added bonus:
  - If implementing the Iterator Pattern with `IEnumerable<T>` the type is queryable with LINQ

```
Deck deck = new Deck();
deck.Shuffle();

var query = deck.Where(c => c.Suit == Suit.Hearts);
foreach (Card card in query)
{
    Console.Write($"{card} ");
}
```


WINCUBATE
Jesper Gulmann Henriksen
PhD, MCT, MCSD, MCPD
Phone : +45 22 12 36 31
Email : jgh@wincubate.net
WWW : http://www.wincubate.net
Hasselvangen 243
8355 Solbjerg
Denmark