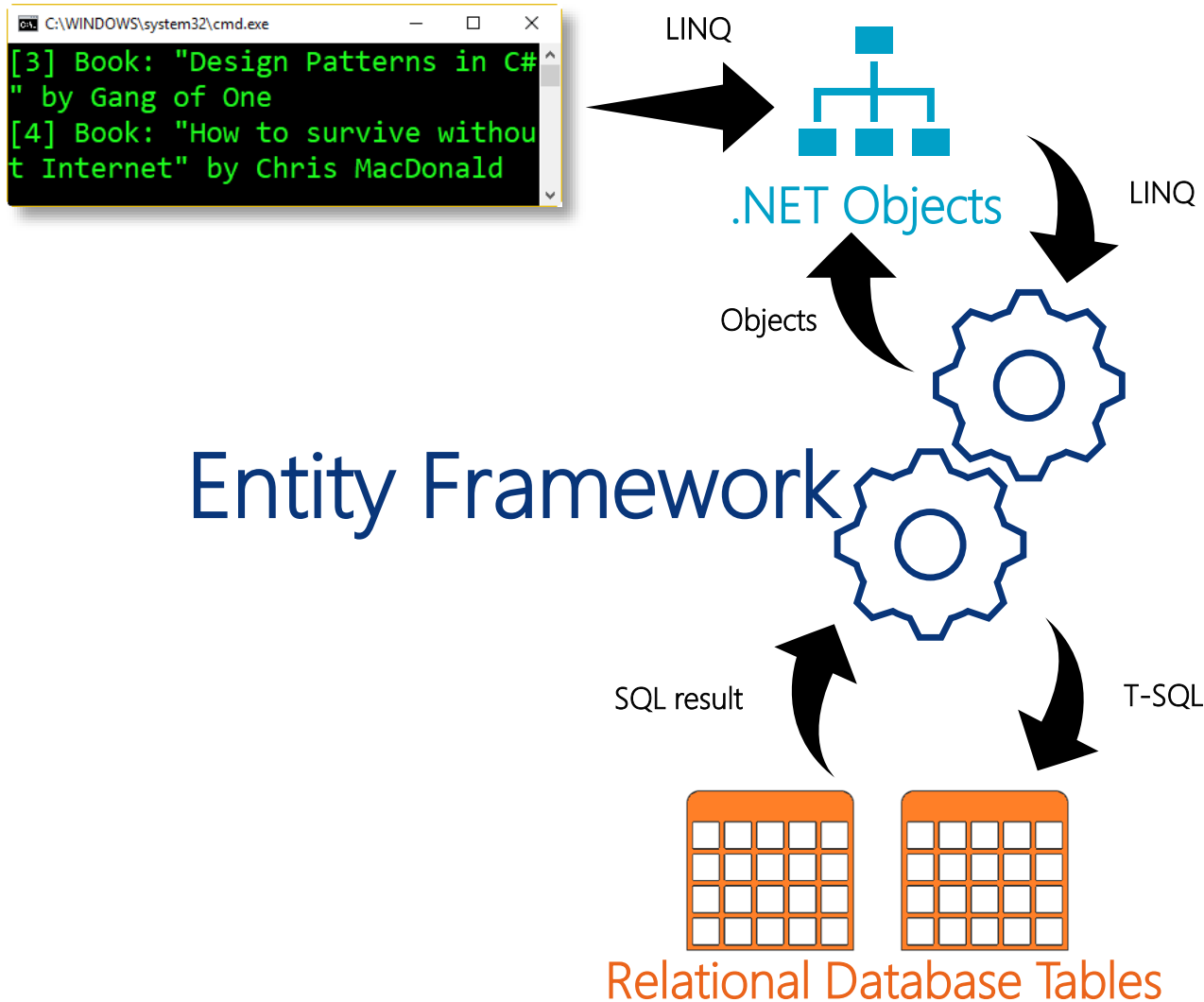# Module 25:
"Repository"
(with Entity Framework)

Agenda

▸ Introductory Example: Products and Data Access

▸ Background: Entity Framework and `IQueryable<T>`

▸ Challenges

▸ Pattern: Repository

▸ 1. "Simple" Repository

▸ 2. Repository returning `IQueryable<T>`

▸ 3. Generic Repository Interface

▸ 4. Generic Repository Implementation

▸ Overview of Repository

Introductory Example: Products and Data Access

```csharp
public class Product
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Manufacturer { get; set; }
    public Category? Category { get; set; }

}
```

```csharp
using (ProductsContext context = new ProductsContext())
{
    var query = context.Products.Where(p => p.Category == Category.Book);
    foreach (var product in query)
    {
        Console.WriteLine(product);
    }
}
```

Background: Entity Framework

LINQ

.NET Objects

LINQ

Objects

Entity Framework

SQL result

T-SQL

Relational Database Tables

```
C:\WINDOWS\system32\cmd.exe                    —   □   ×
[3] Book: "Design Patterns in C#
" by Gang of One
[4] Book: "How to survive withou
t Internet" by Chris MacDonald
```

Challenges

- How do we separate
  - Business logic
  - Data access logic?

- How can we make the business logic testable?

- What if we decide to employ another data source?

Pattern: Repository

▸ *Mediates between the domain and data mapping layers using a collection-like interface for accessing domain objects*

▸ Outline
- Encapsulate data access
- Separate the actual data source from business logic code
- Ensure testability and maintainability of data-driven code

▸ Origin:
- Martin Fowler (2003)
- Eric Evans (2004)

1. "Simple" Repository

▸ Implement a specialized repository for each business object or entity
▸ Disregard any methods not used..! (YAGNI)

```csharp
interface IProductRepository
{
    Product GetById( int id );
    IEnumerable<Product> GetAll();
    IEnumerable<Product> GetAllBooks();
    //void Add( Product product );
    //void Remove( Product product );
}
```

▸ Can implement interface for other data sources, e.g. unit tests

▸ But... What about **GetAllNintendoProducts()** ?

Background: **IQueryable<T>**

▸ Remember the **Expression** class?

```csharp
public interface IQueryable : IEnumerable
{
    Type ElementType { get; }
    Expression Expression { get; }
    IQueryProvider Provider { get; }
}
```

```csharp
public interface IQueryable<out T>
    : IEnumerable<T>, IEnumerable, IQueryable
{
}
```

▸ **IQueryable<T>** represents an "unevaluated" **IEnumerable<T>**

2. Repository returning **IQueryable<T>**

- Incredibly flexible and elegant
- Can efficiently be further queried…!

```csharp
interface IProductRepository
{
    Product GetById( int id );
    IQueryable<Product> GetAll();
    IQueryable<Product> GetAll( Expression<Func<Product, bool>> filter );
    void Add( Product product );
    void Remove( Product product );
}
```

- In-memory implementations can use **AsQueryable()** extension

- <u>Beware</u>: Data access logic might drift into business logic

3. Generic Repository Interface

▸ Abstract element type to be any entity with an **Id** property
▸ Ensures a high degree of consistency and reusability

```csharp
interface IRepository<T> where T : IEntity
{
    T GetById( int id );
    IQueryable<T> GetAll();
    IQueryable<T> GetAll( Expression<Func<T, bool>> filter );
    void Add( T product );
    void Remove( T product );
}
```

```csharp
interface IEntity
{
    int Id { get; }
}
```

▸ **IProductRepository** can add **Product**-specific methods

4 .Generic Repository Implementation

▸ **IRepository<T>** can be implemented generically for EF-contexts

```
class Repository<T> : IRepository<T> where T : class, IEntity
{
    private readonly DbContext _context;

    public Repository( DbContext context ) { _context = context; }

    public T GetById( int id )
        => _context.Set<T>().Single(p => p.Id == id);

    public IQueryable<T> GetAll( Expression<Func<T, bool>> filter )
        => _context.Set<T>().Where(filter);
    ...
}
```
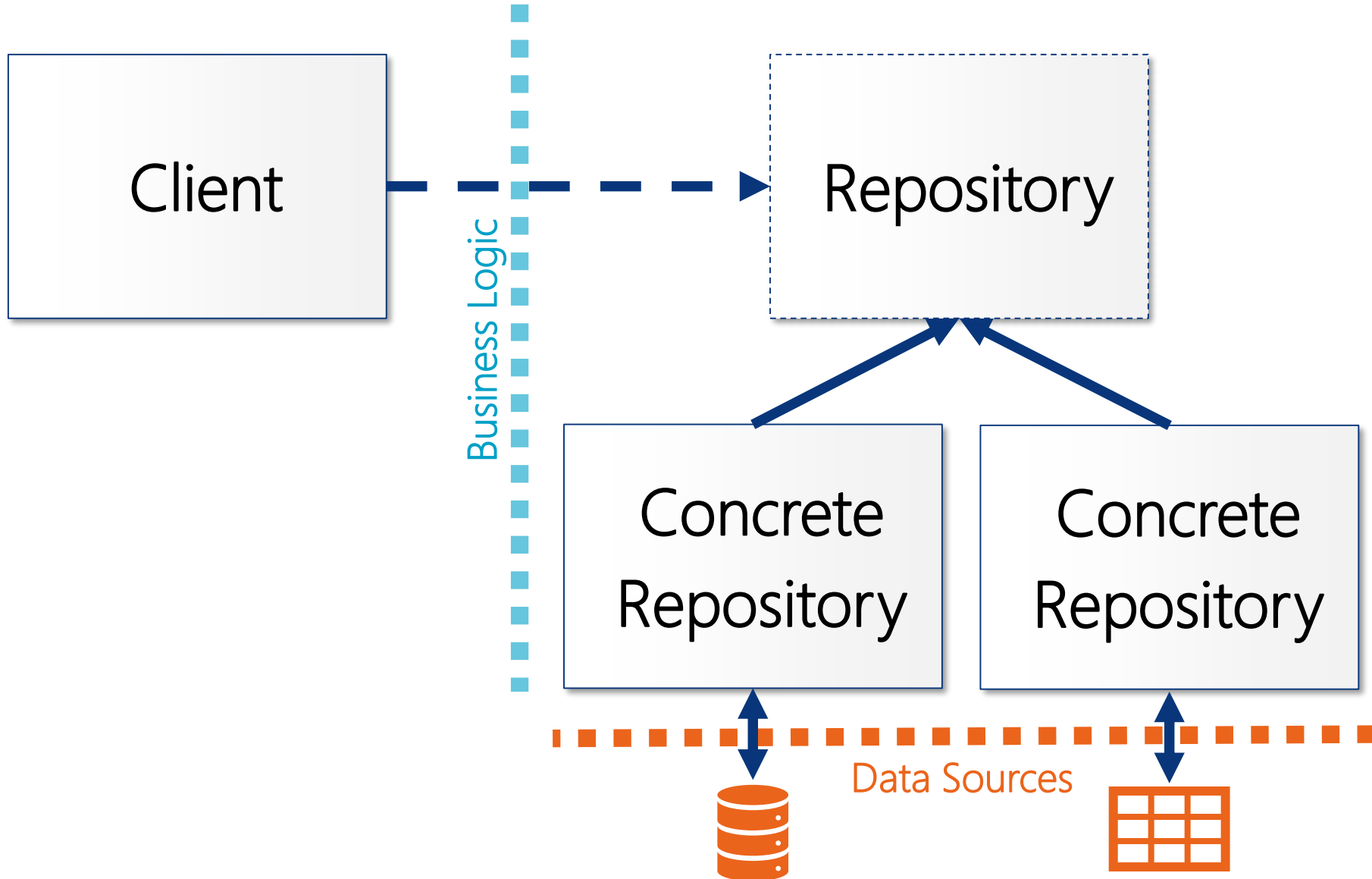
Overview of Repository Pattern

Overview of Repository Pattern

▸ Client
- Queries and updates data through the Repository Interface
- Only know the general Repository interface

▸ Repository
- Interface or base class exposing data access logic in a persistence-independent description

▸ Concrete Repository
- Concrete repository class implementing Repository interface
- Implements persistence-dependent data access code for a specific concrete data source

Discussion

▸ Simple Repository
- Implement a specialized repository for each business object
- Disregard any methods not used! (YAGNI)

▸ `IQueryable`-based Repository
- Flexible and efficient
- Beware: Data access logic might drift into business logic

▸ Generic interfaces and implementation
- High degree of consistency and reusability

▸ <u>Note</u>:
- Can of course do generic interfaces and implementations based on `IEnumerable<T>` (and not `IQueryable<T>`), if preferred

▸ Unit of Work pattern for more complex situations

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