Module 09:
"Repository"
(with Entity Framework)

TEKNOLOGISK
INSTITUT

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

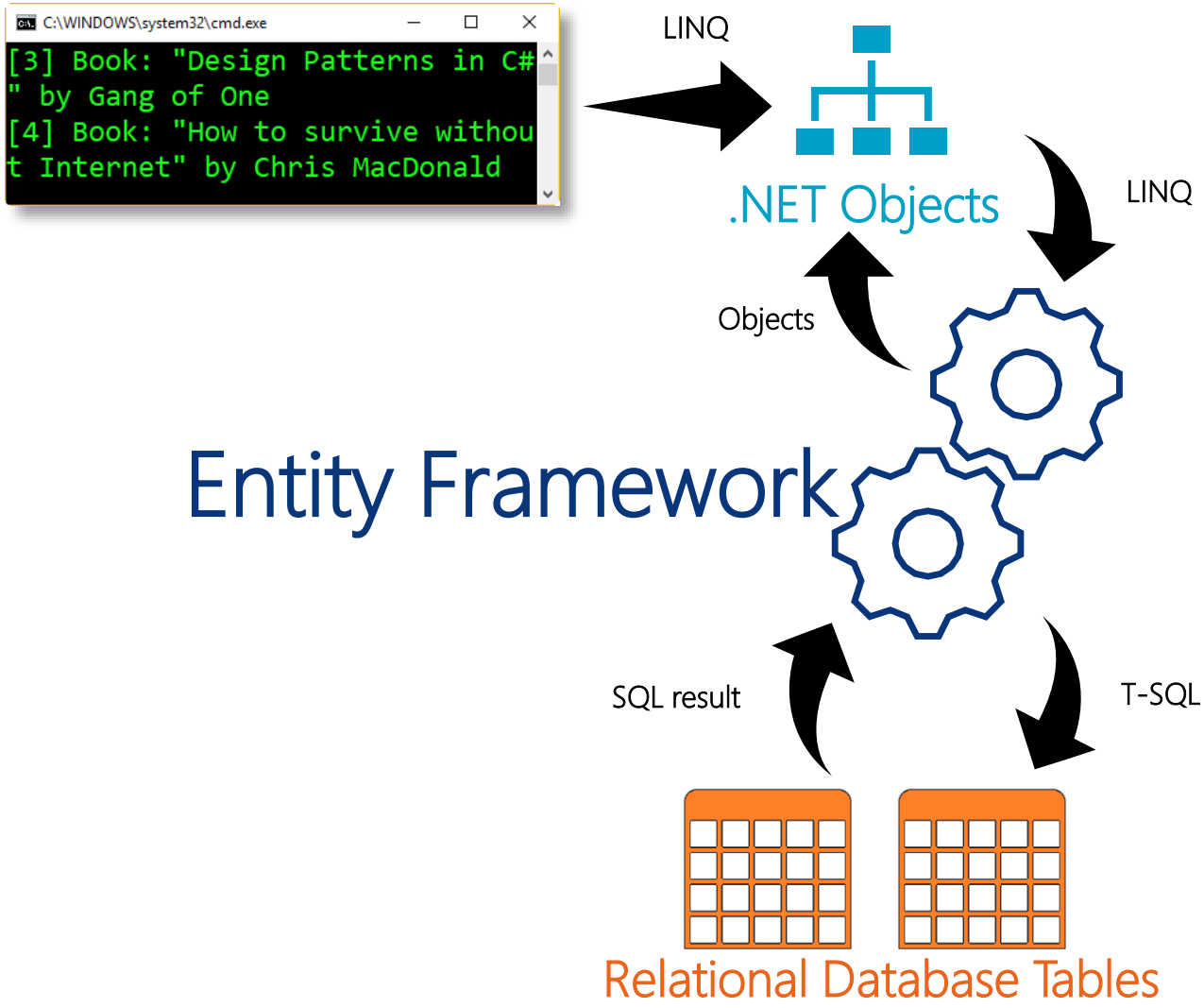
TEKNOLOGISK INSTITUT

Introductory Example: Products and Data Access

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

```
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



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
  - Separate the actual data source from business logic code
  - Avoid repetition of queries 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)

```
interface IProductRepository
{
    Product GetById( int id );
    IEnumerable<Product> GetAll();
    IEnumerable<Product> GetForCategory( Category? category );
    //void Add( Product product );
    //void Remove( Product product );
}
```

- Can implement interface for other data sources, e.g. unit tests
- But... Isn't most stuff of this not generic to every repository?

Background: `IQueryable<T>`

- Remember the `Expression` class?

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

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

- `IQueryable<T>` represents an AST for an `IEnumerable<T>`

2. Repository returning `IQueryable<T>`

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

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

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

- Beware: Data access (and logic) might drift into business logic

3. Generic Repository Interface

- We reuse as much as possible, but leak no **`IQueryable<T>`**
- Ensures a high degree of consistency and reusability

```
interface IRepository<TEntity> where T : class
{
    TEntity GetById( int id );
    IEnumerable<TEntity> GetAll();
    IEnumerable<TEntity> Find( Expression<Func<TEntity,bool>> filter );
    void Add( TEntity entity );
    void AddRange( IEnumerable<TEntity> entities );
    void Remove( TEntity entity );
    void RemoveRange( IEnumerable<TEntity> entities );
}
```

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

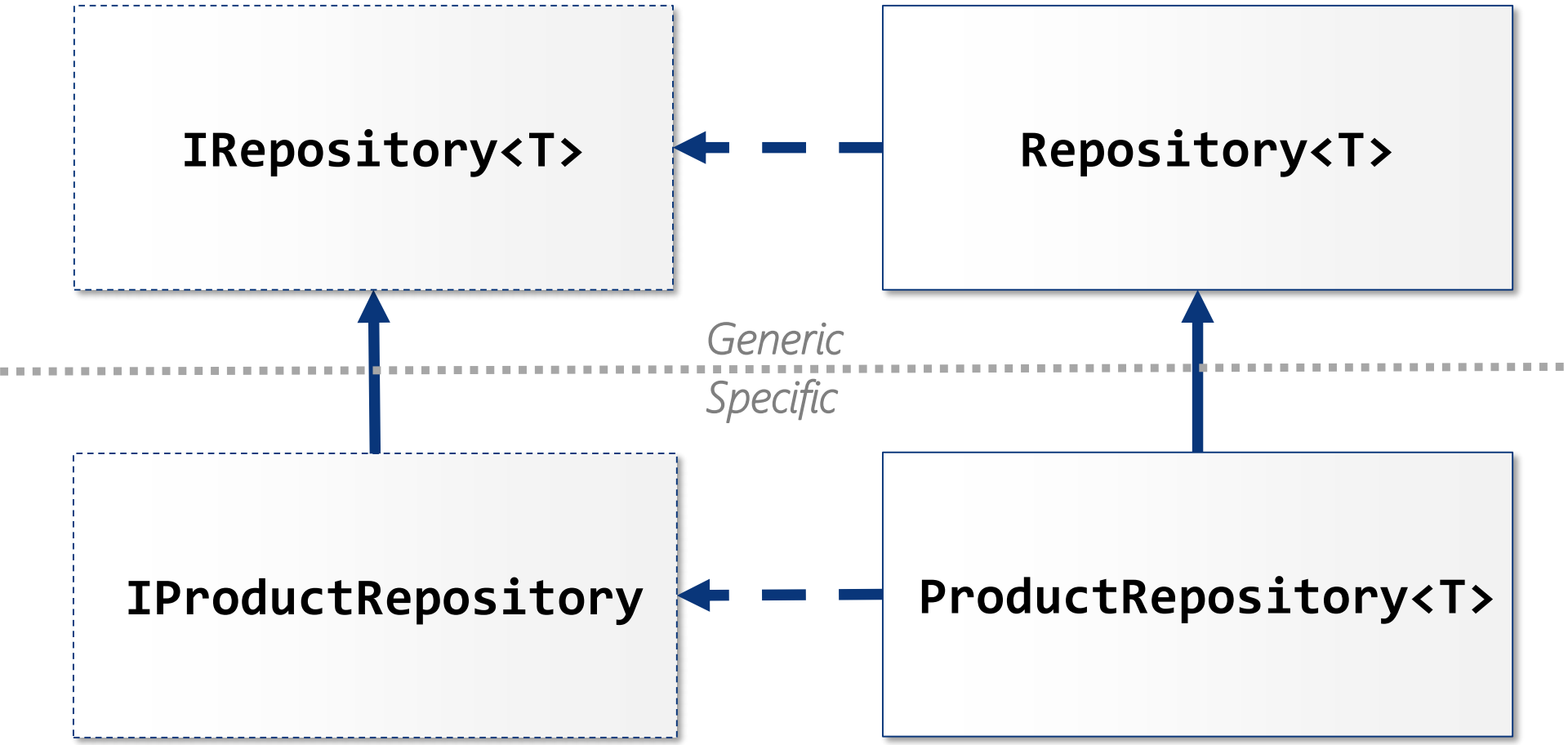
4 .Generic Repository Implementation

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

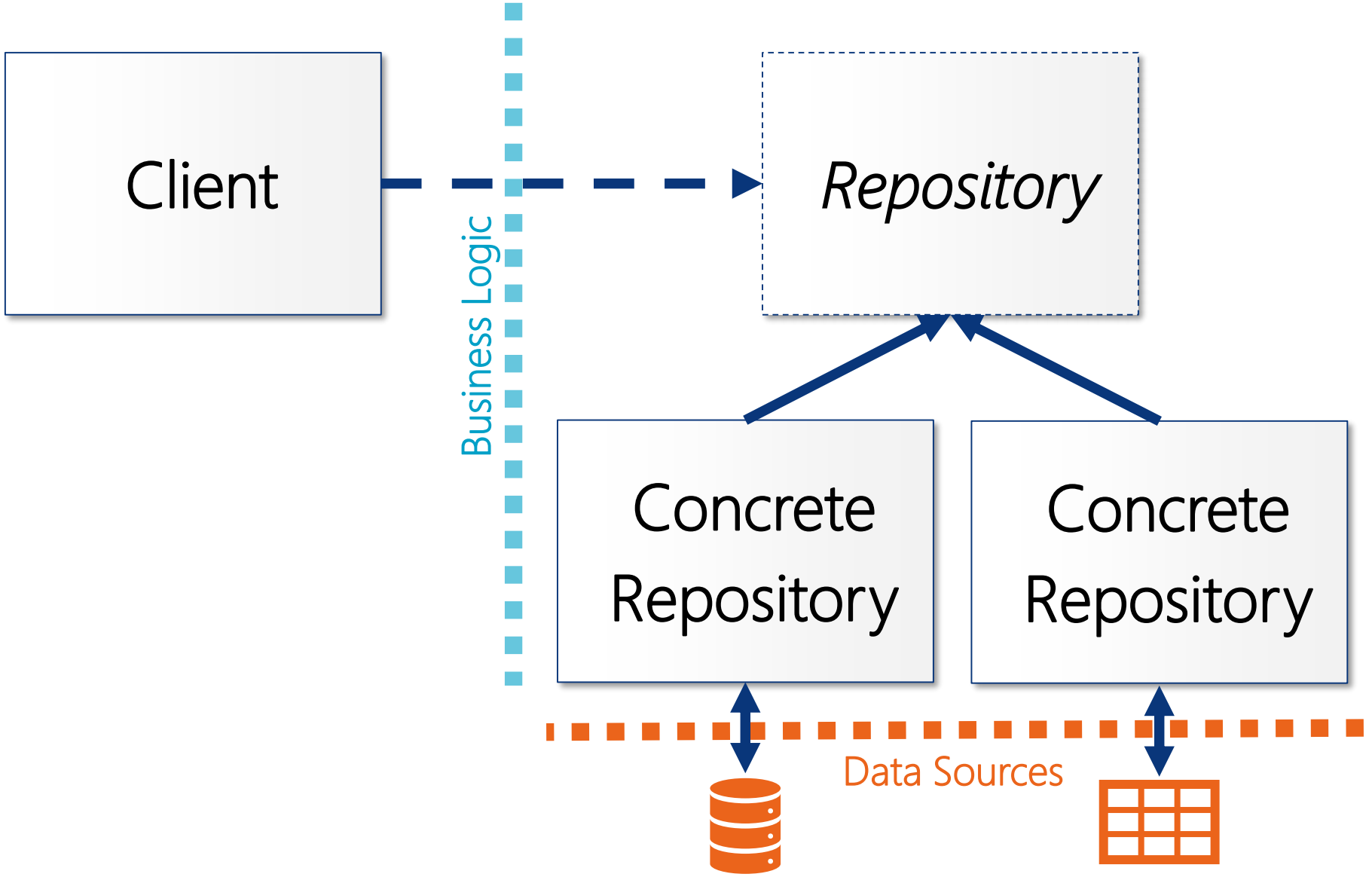
```
class Repository<TEntity> : IRepository<TEntity> where TEntity : class
{
    protected DbContext Context { get; }

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

    public TEntity GetById( int id )
        => Context.Set<TEntity>().Single(p => p.Id == id);

     public IEnumerable<TEntity> Find( Expression<Func<TEntity,bool>> f )
        => Context.Set<TEntity>().Where(f);
    ...
}
```

TEKNOLOGISK INSTITUT

Generic vs. Specific Implementation



Overview of Repository Pattern



Overview of Repository Pattern

- Client
  - Queries and updates data through the Repository Interface
  - Only knows the general Repository interface
- Repository
  - Interface or base class exposing data access logic in a collection-styled 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
- Note:
  - Can of course do generic interfaces and implementations based on `IEnumerable<T>` (and not `IQueryable<T>`), if preferred

Next Up: Unit of Work

- Unit of Work pattern for more complex situations
- Widely misunderstood implementations
  - Collection-like vs. Persistence-specific
  - Mixes Repository and Unit of Work
- Controversy
  - Repository should not have Save() – Unit of Work has!
  - Repository should have Update() – Not collection-like!


WINCUBATE
Jesper Gulmann Henriksen
PhD, MCT, MCSD, MCPD
Phone : +45 22 12 36 31
Email : jgh@wincubate.net
WWW : http://www.wincubate.net
Ringgårdsvej 4A
8270 Højbjerg
Denmark