Functional Techniques for C#

Kathleen Dollard
Principal Program Manager, Microsoft

@kathleendollard
kdollard@microsoft.com

You are effective with the imperative, object-oriented core of Java or .NET but you look longingly at the winsome smile of functional languages.

If you play with your language’s functional features, you’re never quite sure if you’re getting it right or taking full advantage of them. This talk is for you.

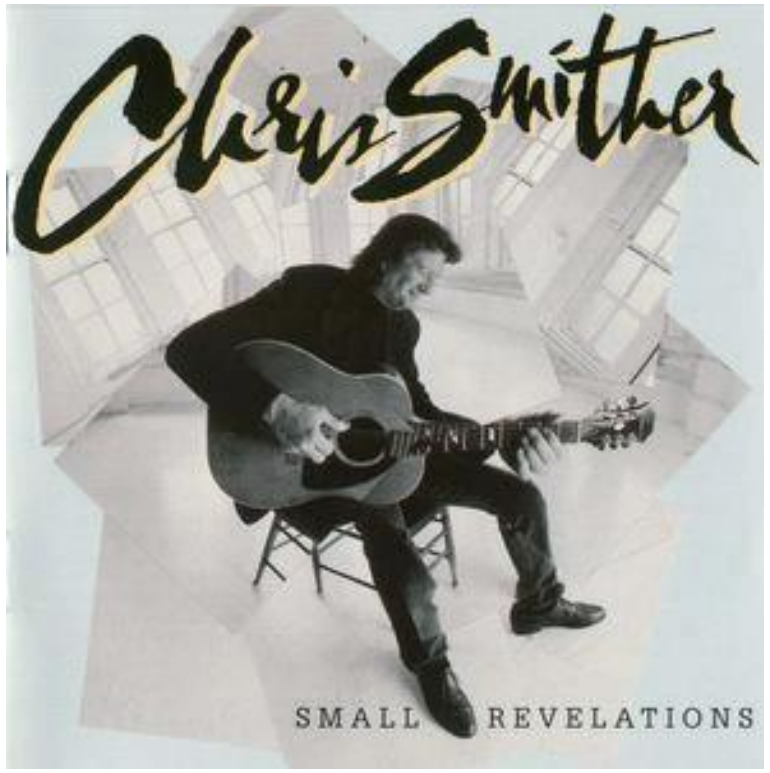
You’ll learn which code to attack with functional ideas and how to do it. You’ll look at code similar to what you write every day, and see it transform from long, difficult-to-follow code to short code that’s easy to understand, hard to mess up, and straightforward to debug. Better yet, functional approaches help you apply patterns in a clear and consistent way.

Apply these techniques while leveraging delegates, lambda expressions, base classes and generics.

Winsome Smile

Chris Smither

Small Revelations album



Winsome –

“attractive or appealing in appearance or character.”

- *Defined by Google*

C# ❤ Functional techniques

Scary words…

- Currying
  - In mathematics and computer science, **currying** is the technique of translating the evaluation of a function that takes multiple arguments (or a tuple of arguments) into evaluating a sequence of functions, each with a single argument. Currying is related to, but not the same as, partial application.
  - f(true, 42) -> f(g(42))
- For C#, currying is useful when it adds intent

```
repo.GetById(true, 42);
repo.GetById(active: true, id: 42);
repo.GetActiveById(id: 42); // curries to above
```

Scary words…

- Tail call recursion
  - Iterate a loop by taking off one item and recursively calling f on remainder
  - Not optimized in C#
  - Just don’t

- You neither need nor want everything in the functional world

Scary words…

- Immutability
  - Once created the state, or data for an object does not change
  - Consider systems, or parts of systems that do not change state

- Partial use of immutability is not very helpful
- Consider across categories of similar classes in your application

Scary words…

- Higher order functions
  - Functions with delegate parameters or return delegates

Delegates – functions as data

- Generic delegate types (Action, Func)
- Type safe function pointers
    - System.Delegate and inherited types
        - “Named” in docs
    - Reference to a method (name without parens)
        - Can be a local method
    - Anonymous methods
        - delegate()
    - Lambdas

Delegates – functions as data

- Generic delegate types (Action, Func)
- Type safe function pointers
  - ~~System.Delegate and inherited types~~
    - ~~“Named” in docs~~
  - Reference to a method (name without parens)
    - Can be a local method
  - ~~Anonymous methods~~
    - ~~delegate()~~
  - Lambdas

Scary words…

- Higher order functions
  - Functions with delegate parameters or return delegates
  - LINQ
    - Select, Where, OrderBy etc. are higher order functions
- C# is great at doing delegates

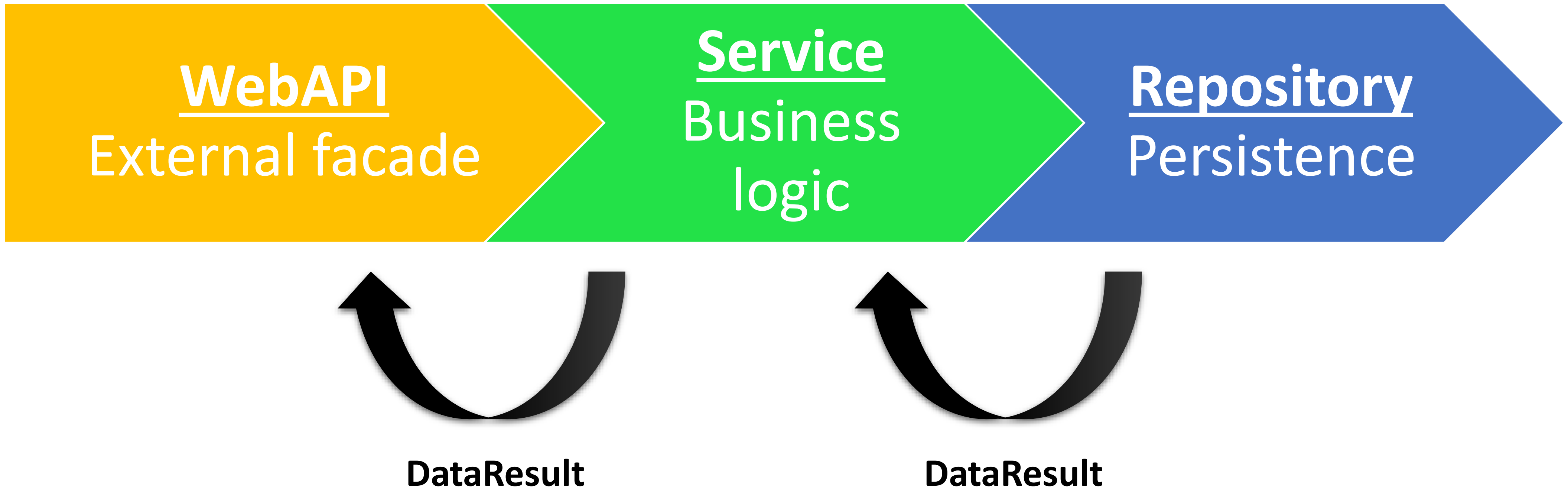
f
Delegate
Lambda
func

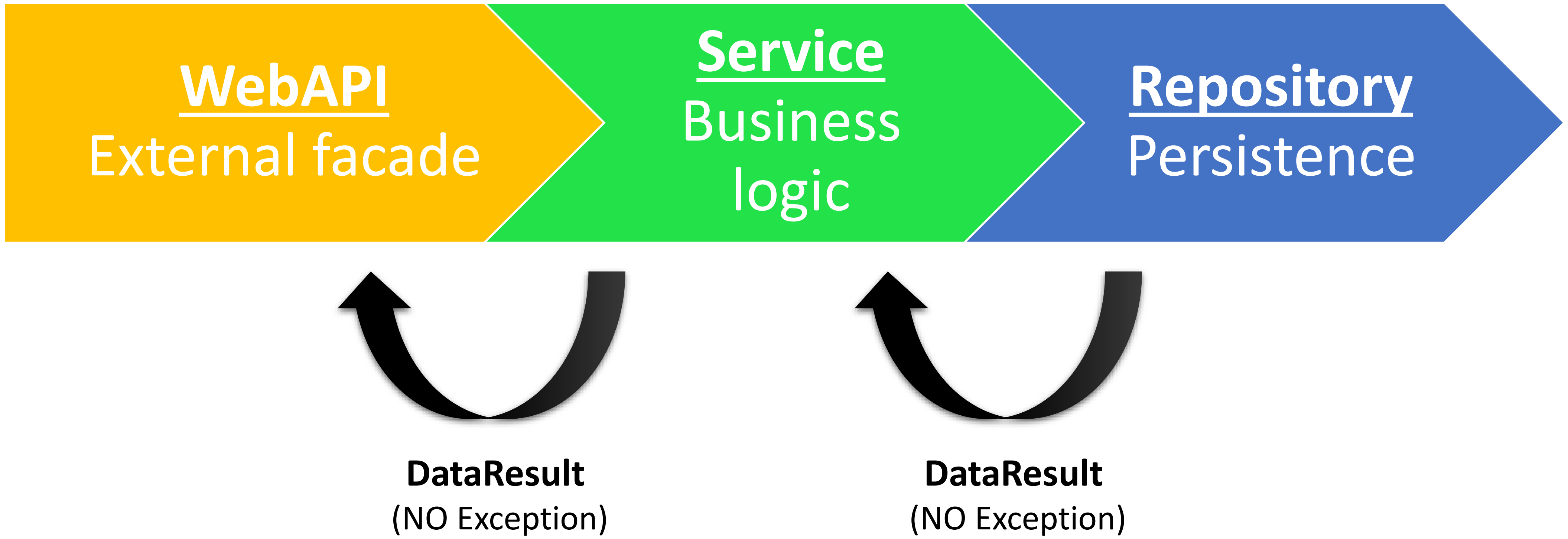
Are the same in today’s context

Lambda review

- **`=>`**
  - **`Func<int, int> f1 = x => x + 2;`**
  - **`Func<int> f2 = () => 42;`**
  - **`Func<int, int, int> f3 = (x, y) => x + y;`**

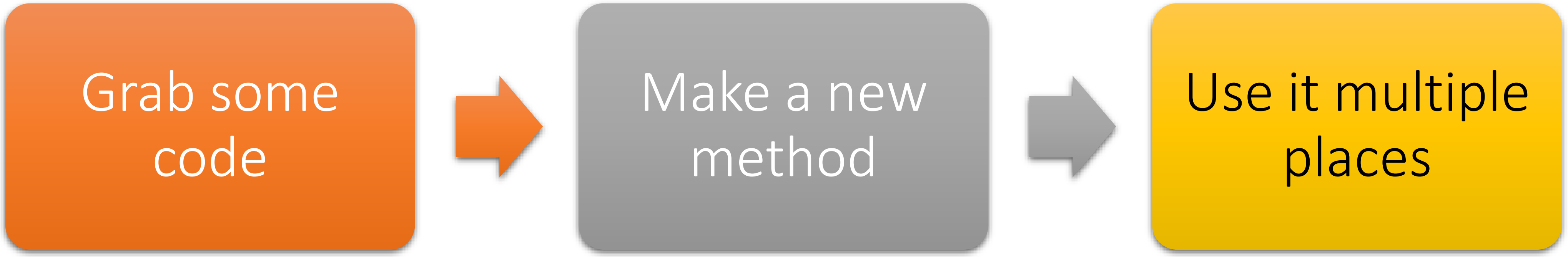
- Delegates: code fragments that can be stored to execute later
- **`Func<T>`**
  - **`Func<T<T1<T2>>>`**
- **`Func<T1, T2>`**
  - **`Func<TParam, T<T1<T2>>>`**
  - **`Func<int, Task<DataResult<List<Student>>>>`**


WebAPI
External facade
Service
Business logic
Repository
Persistence
DataResult
DataResult

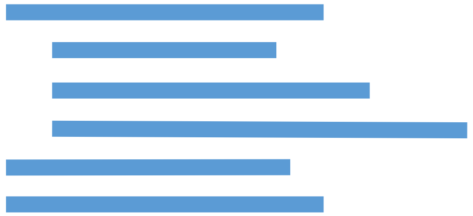

WebAPI
External facade
Service
Business logic
Repository
Persistence
DataResult
(NO Exception)
DataResult
(NO Exception)

Imperative (normal) Refactoring

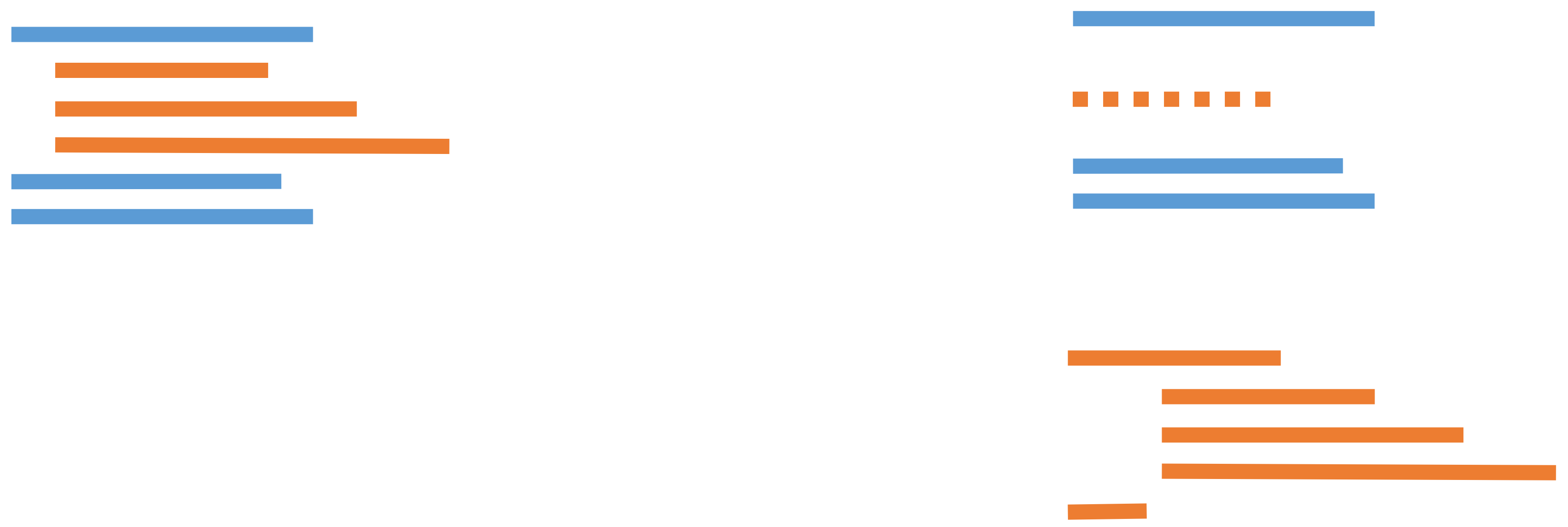
- Inside out refactoring



Imperative Refactoring



Imperative Refactoring

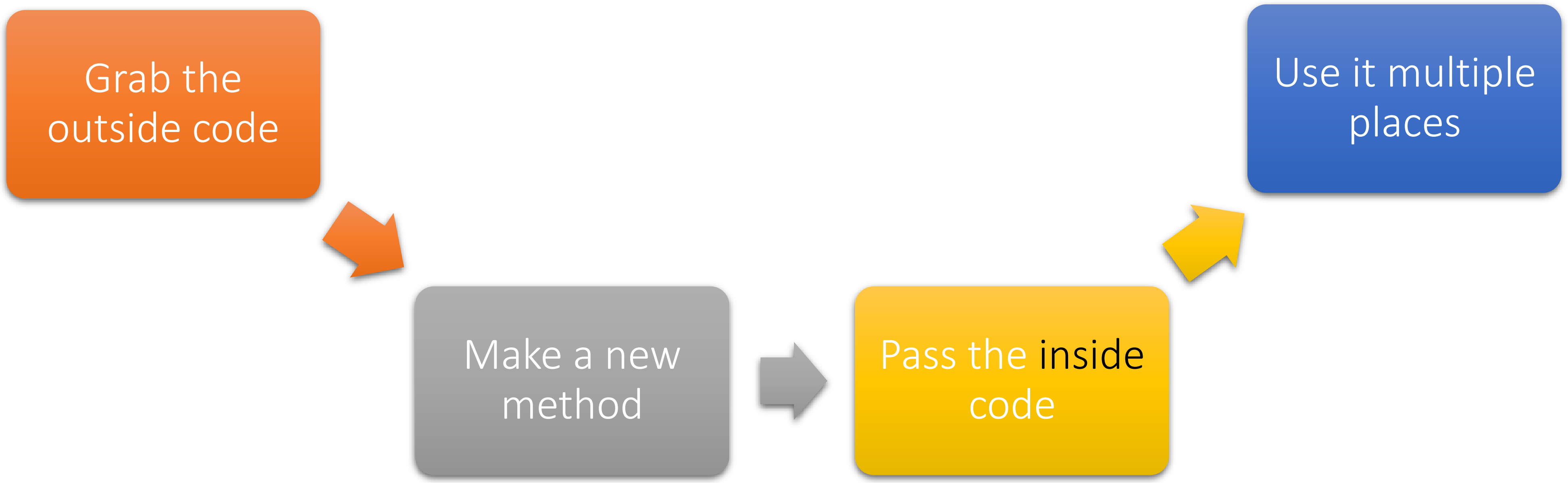


Demo!

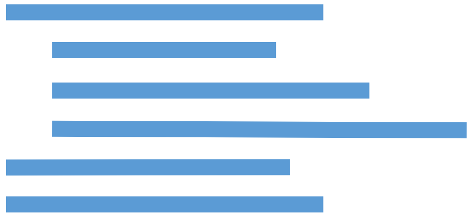
Inside out refactoring (normal)

Functional Refactoring

- Outside in refactoring



Functional Refactoring

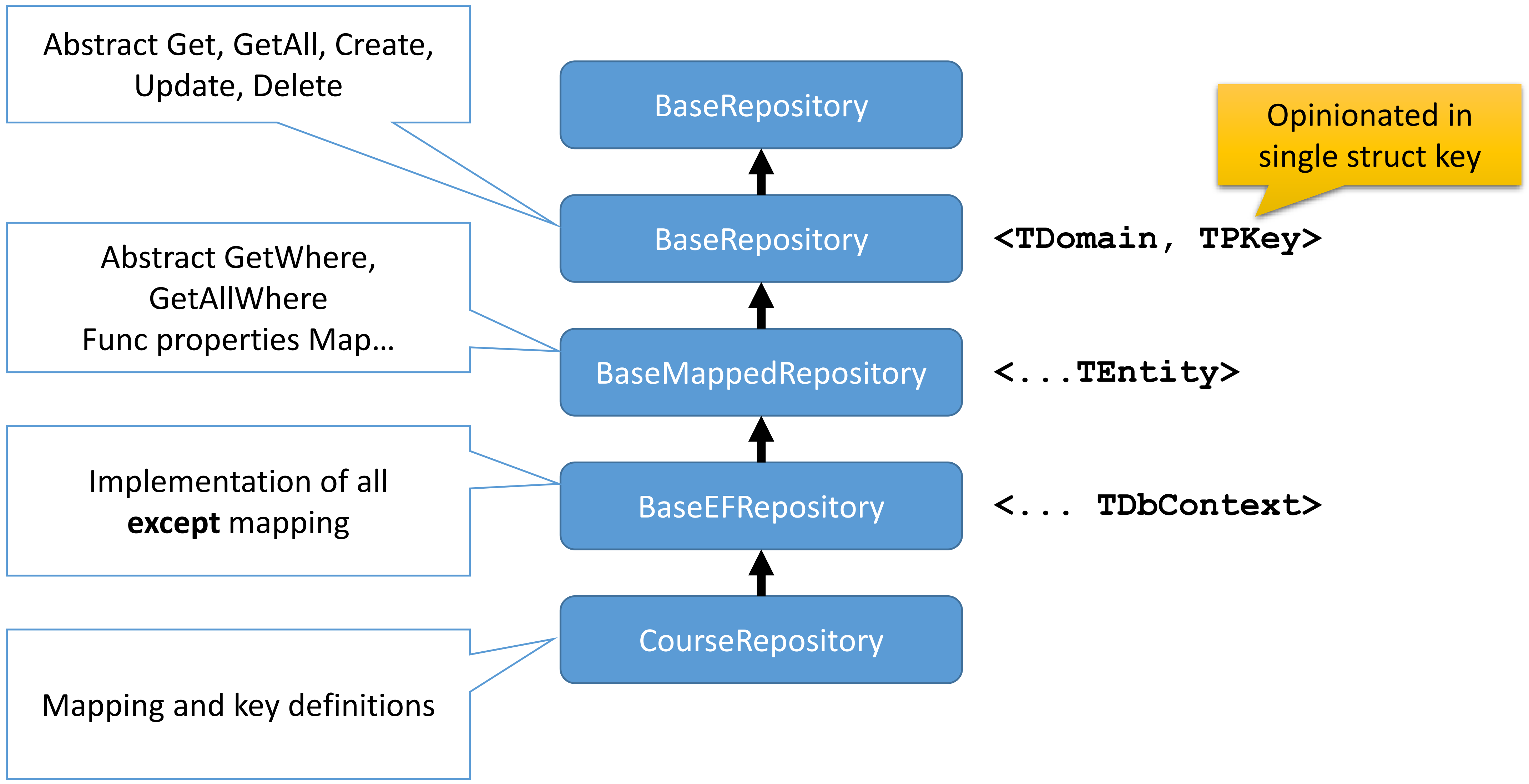


Functional Refactoring

Functional Refactoring

Demo!

Outside in refactoring


Abstract Get, GetAll, Create, Update, Delete
BaseRepository
Opinionated in single struct key
BaseRepository
<TDomain, TPKey>
Abstract GetWhere, GetAllWhere Func properties Map…
BaseMappedRepository
<...TEntity>
Implementation of all **except** mapping
BaseEFRepository
<... TDbContext>
Mapping and key definitions
CourseRepository

Demo!

How this works in an application

Questions?

Functional Techniques for C#

@kathleendollard
kdollard@microsoft.com

References

- Today's code: https://github.com/KathleenDollard/ClassTracker
- Pluralsight : *Applying Functional Principles in C#,* Vladimir Khorikov
- Pluralsight : *Functional Programming with C#,* Dave Fancher