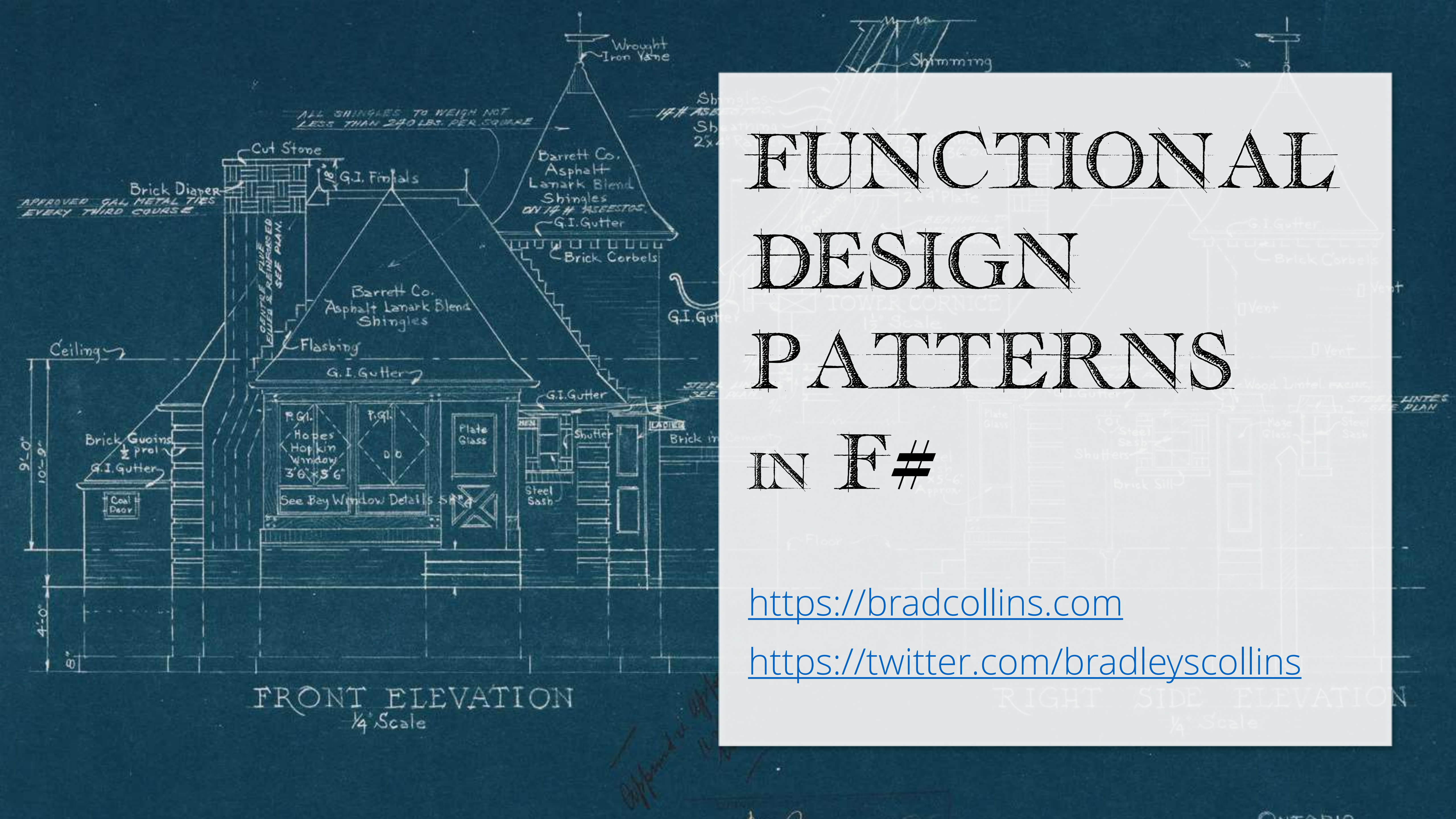
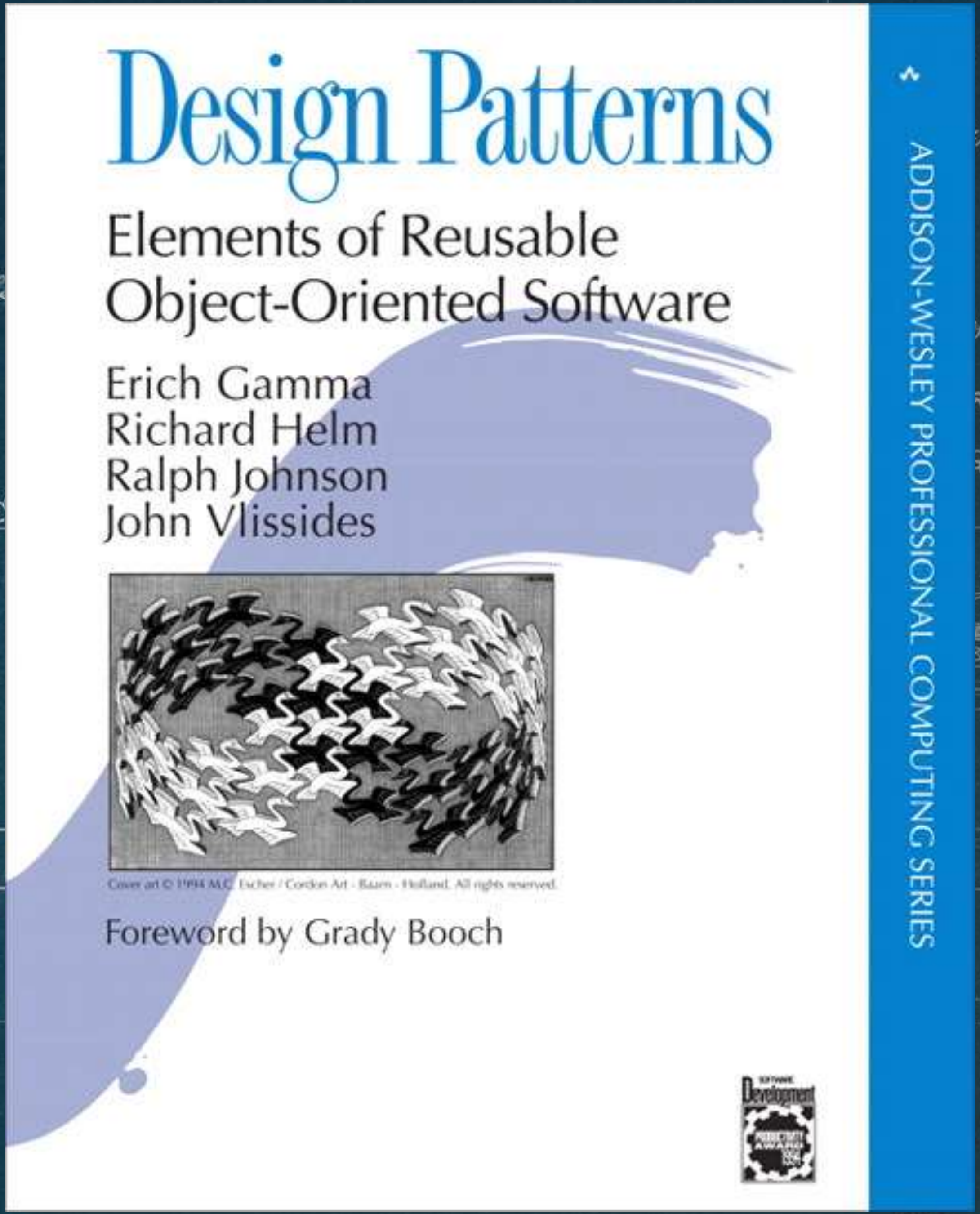


FUNCTIONAL DESIGN PATTERNS IN F#

https://bradcollins.com

https://twitter.com/bradleyscollins



Design Patterns

Elements of Reusable
Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.

Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

The Joy of
Using Patte
The Addison-Wesley Signature Series
A Martin Fowler Signature Book
Refactoring to Patterns
Joshua Kerievsky
Forewords by Ralph Johnson and Martin Fowler
Afterword by John Brant and Don Roberts
Design Patterns in Java
Series
ion
terns
ot 4
hing
Applied
John Vlissides
Foreword by James O. Coplien
Software Patterns Series
The Design Patterns Smalltalk Companion
Head First Design Patterns
A Brain-Friendly Guide
10th Anniversary
Updated for Java 8
Eric Freeman & Elisabeth Robson
with Kathy Sierra & Bert Bates
Des
A New P
Second Ed
Design Patterns in Ruby
Foreword by Obie Fernandez
Professional Ruby Series Editor
Russ Olsen
Steven John Metsker
Software Patterns Series

Real-World Functional Programming

With examples in F# and C#

SAMPLE CHAPTER

Tomas Petricek with Jon Skeet

Foreword by Mads Torgersen

MANNING

THE BOOK OF F#

BREAKING FREE WITH

FUNCTIONAL PROG

DAVE FANCHER

COVERS F# 3.1

F# DEEP

Editors Tomas Petricek • Phillip Trelford

Contributors Chris Ballard • Keith Battocchi • Colin Bull • Chao-Jen Chen • Yan Cui •
Evelina Gabasova • Dmitry Morozov • Tomas Petricek • Don Syme • Phil

Expert F# 4.0

Fourth Edition

Don Syme
Adam Granicz
Antonio Cisternino

Apress®

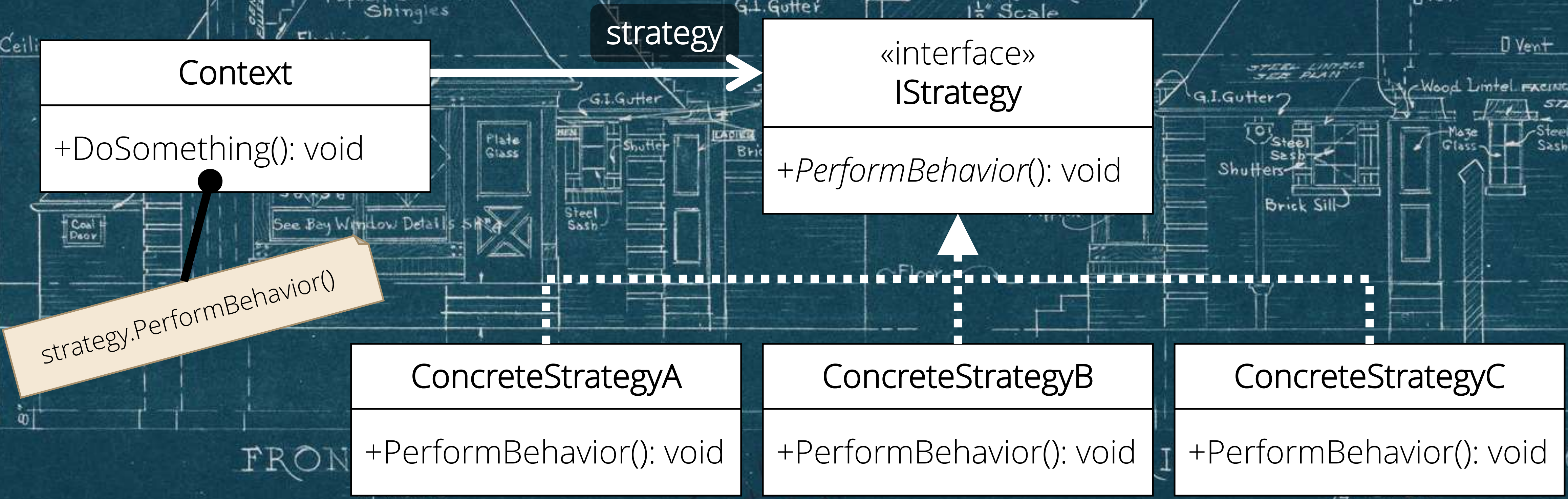
OBJECT-ORIENTED

- Builder
- Null Object
- Template Method
- Strategy
- Adapter
- State
- Dependency Injection
- Visitor

FUNCTIONAL

- Currying
- Pattern Matching
- Partial Application
- Function Builder
- Monads
- Map-Reduce
- Algebraic Data Types

STRATEGY



Context

+DoSomething(): void

strategy

«interface»
IStrategy

+*PerformBehavior*(): void

strategy.PerformBehavior()

ConcreteStrategyA

+PerformBehavior(): void

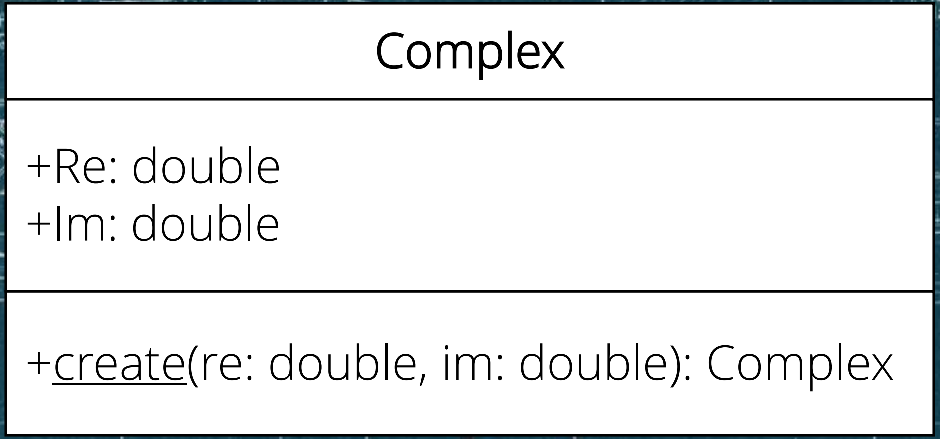
ConcreteStrategyB

+PerformBehavior(): void

ConcreteStrategyC

+PerformBehavior(): void

STRATEGY

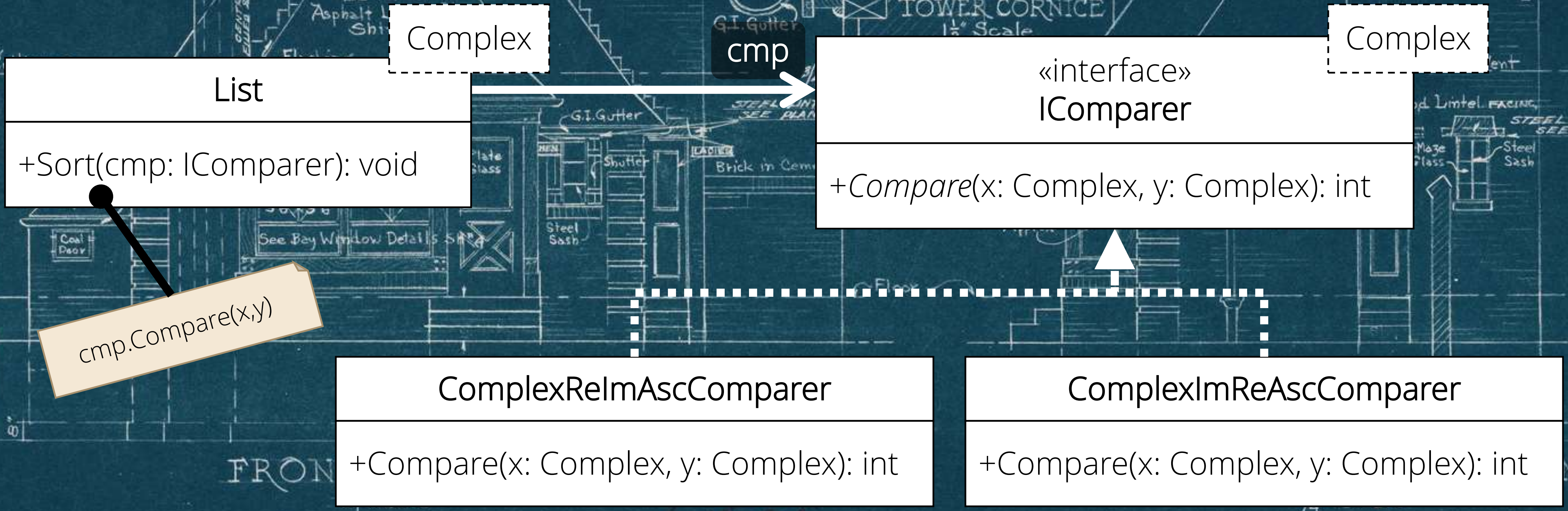


Complex

+Re: double
+Im: double

+create(re: double, im: double): Complex

STRATEGY



Complex

List

+Sort(cmp: IComparer): void

cmp.Compare(x,y)

cmp

Complex

«interface»
IComparer

+*Compare*(x: Complex, y: Complex): int

ComplexReImAscComparer

+Compare(x: Complex, y: Complex): int

ComplexImReAscComparer

+Compare(x: Complex, y: Complex): int

C#

STRATEGY

```csharp
public class Complex
{
    public readonly double Re;
    public readonly double Im;
    public Complex(double re, double im)
    {
        Re = re;
        Im = im;
    }
}
```

C#

STRATEGY

```
public class ComplexReImAscComparer : IComparer<Complex>
{
    public int Compare(Complex x, Complex y)
    {
        var re = Math.Sign(x.Re - y.Re);
        return re != 0 ? re : Math.Sign(x.Im - y.Im);
    }
}
```

C#

STRATEGY

```
public class ComplexImReAscComparer : IComparer<Complex>
{
    public int Compare(Complex x, Complex y)
    {
        var im = Math.Sign(x.Im - y.Im);
        return im != 0 ? im : Math.Sign(x.Re - y.Re);
    }
}
```

C#

STRATEGY

```
var xs = new List<Complex>()
{
    new Complex( 4,  9),
    new Complex(-3,  1),
    new Complex( 1, -6),
    new Complex(-3, -2),
    new Complex( 7,  1),
};
```

C#

STRATEGY

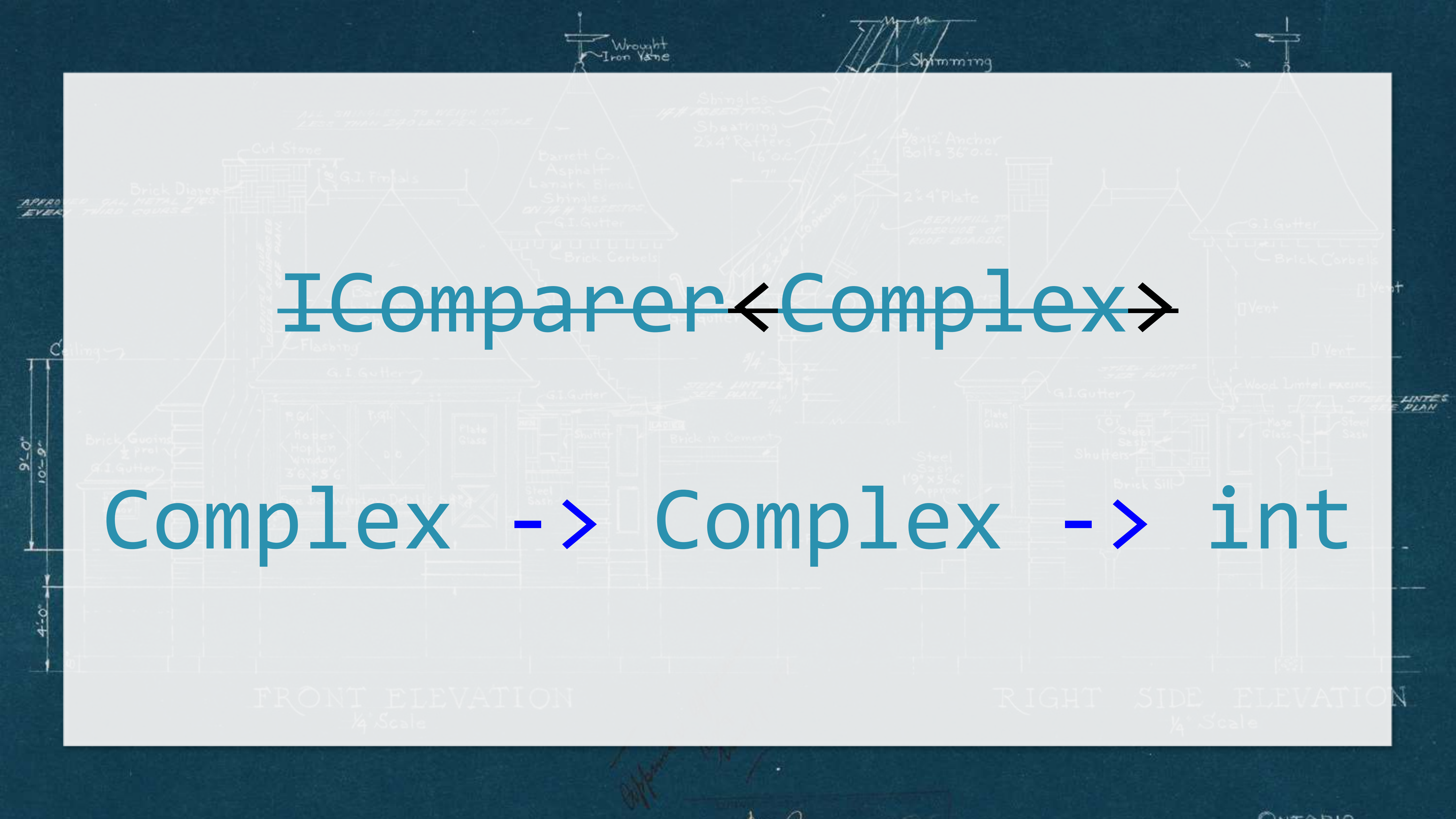
```
Console.WriteLine("Ascending by Re, Im");
xs.Sort(new ComplexReImAscComparer());
foreach (var x in xs) Console.WriteLine(x);

Console.WriteLine();
Console.WriteLine("Ascending by Im, Re");
xs.Sort(new ComplexImReAscComparer());
foreach (var x in xs) Console.WriteLine(x);
```

```
Ascending by Re, Im
-3.0 - i2.0
-3.0 + i1.0
 1.0 - i6.0
 4.0 + i9.0
 7.0 + i1.0
Ascending by Im, Re
 1.0 - i6.0
-3.0 - i2.0
-3.0 + i1.0
 7.0 + i1.0
 4.0 + i9.0
```

C#

STRATEGY

```csharp
public class ComplexReImAscComparer : IComparer<Complex>
{
    public int Compare(Complex x, Complex y)
    {
        var re = Math.Sign(x.Re - y.Re);
        return re != 0 ? re : Math.Sign(x.Im - y.Im);
    }
}
```



~~IComparer<Complex>~~

Complex -> Complex -> int

F#

STRATEGY

```
type Complex = { Re : float; Im : float }

let xs = [ { Re =  4.0; Im =  9.0 }
           { Re = -3.0; Im =  1.0 }
           { Re =  1.0; Im = -6.0 }
           { Re = -3.0; Im = -2.0 }
           { Re =  7.0; Im =  1.0 } ]
```

F#

STRATEGY

```
let compareReImAsc x y =
    let re = sign (x.Re - y.Re)
    if re <> 0 then re else sign (x.Im - y.Im)

let compareImReAsc x y =
    let im = sign (x.Im - y.Im)
    if im <> 0 then im else sign (x.Re - y.Re)
```

F#

STRATEGY

```
printfn "Ascending by Re, Im"
xs
|> List.sortWith compareReImAsc
|> List.iter (printfn "%O")
printfn ""
printfn "Ascending by Im, Re"
xs
|> List.sortWith compareImReAsc
|> List.iter (printfn "%O")
```

```
Ascending by Re, Im
-3.0 - i2.0
-3.0 + i1.0
 1.0 - i6.0
 4.0 + i9.0
 7.0 + i1.0
Ascending by Im, Re
 1.0 - i6.0
-3.0 - i2.0
-3.0 + i1.0
 7.0 + i1.0
 4.0 + i9.0
```

STRATEGY

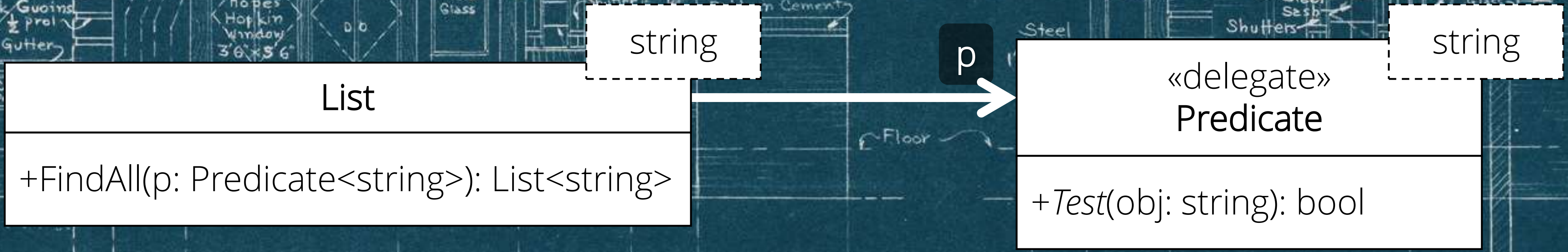
C#

```
public class ComplexReImAscComparer : IComparer<Complex>
{
    public int Compare(Complex x, Complex y)
    {
        var re = Math.Sign(x.Re - y.Re);
        return re != 0 ? re : Math.Sign(x.Im - y.Im);
    }
}

public class ComplexImReAscComparer : IComparer<Complex>
{
    public int Compare(Complex x, Complex y)
    {
        var im = Math.Sign(x.Im - y.Im);
        return im != 0 ? im : Math.Sign(x.Re - y.Re);
    }
}
```

F#

```
let compareReImAsc x y =
    let re = sign (x.Re - y.Re)
    if re <> 0 then re else sign (x.Im - y.Im)

let compareImReAsc x y =
    let im = sign (x.Im - y.Im)
    if im <> 0 then im else sign (x.Re - y.Re)
```

STRATEGY



string

List

+FindAll(p: Predicate<string>): List<string>

p

string

«delegate»
Predicate

+*Test*(obj: string): bool

STRATEGY

C#

```
public class StartsWith
{
    public string Prefix { get; private set; }
    public StartsWith(string prefix)
    {
        Prefix = prefix;
    }
    public bool Test(string sample)
    {
        return sample.StartsWith(Prefix);
    }
}
```

C#

STRATEGY

```
var words = new List<string>()
{
    "Lorem",      "ipsum",
    "dolor",      "sit",
    "amet",       "consectetur",
    "adipiscing", "elit",
};
```

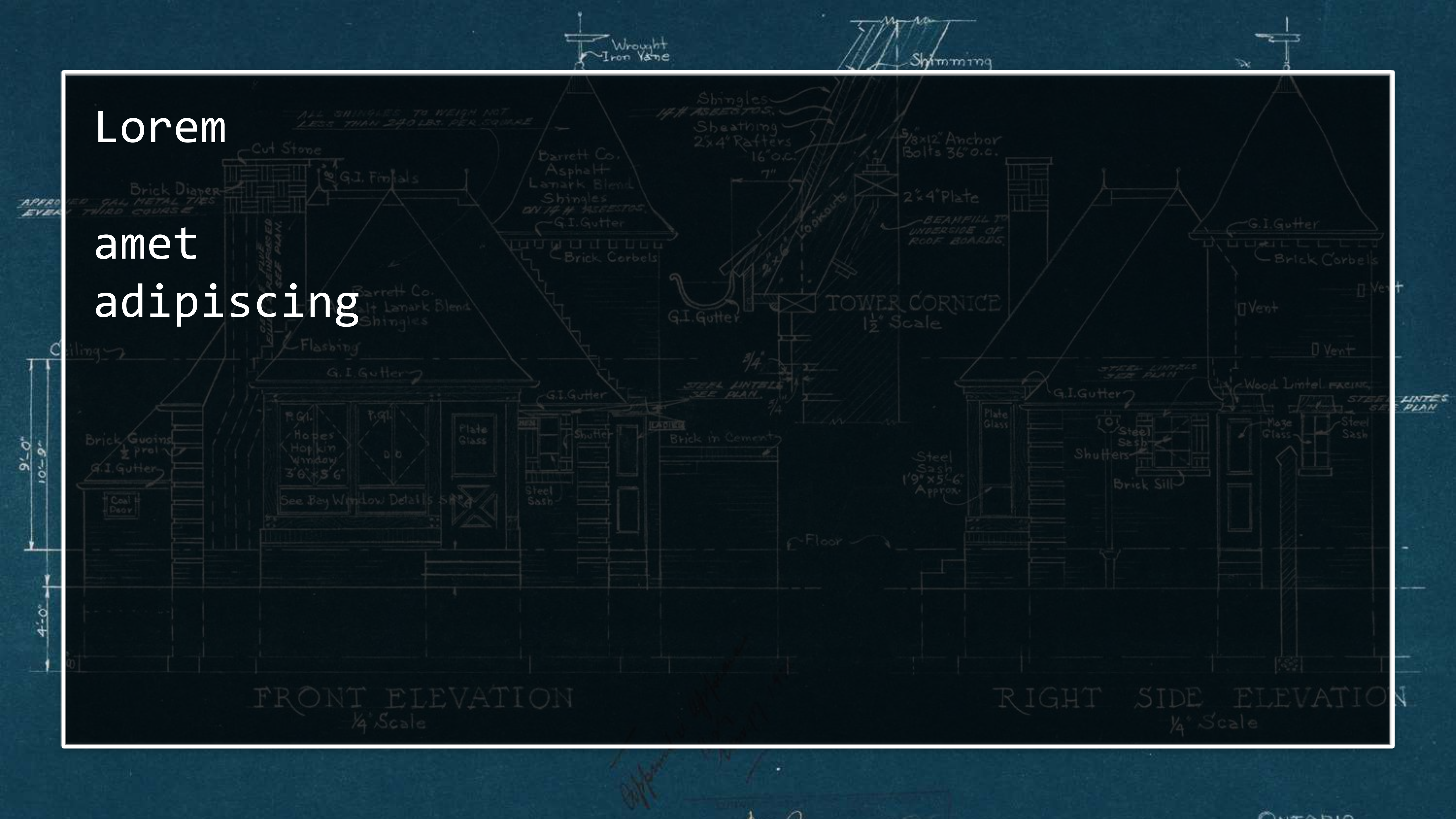
C#

STRATEGY

```csharp
var startsWithL = new StartsWith("L");
var lWords = words.FindAll(startsWithL.Test);
foreach (var word in lWords) Console.WriteLine(word);

var startsWithA = new StartsWith("a");
var aWords = words.FindAll(startsWithA.Test);
foreach (var word in aWords) Console.WriteLine(word);
```

Lorem

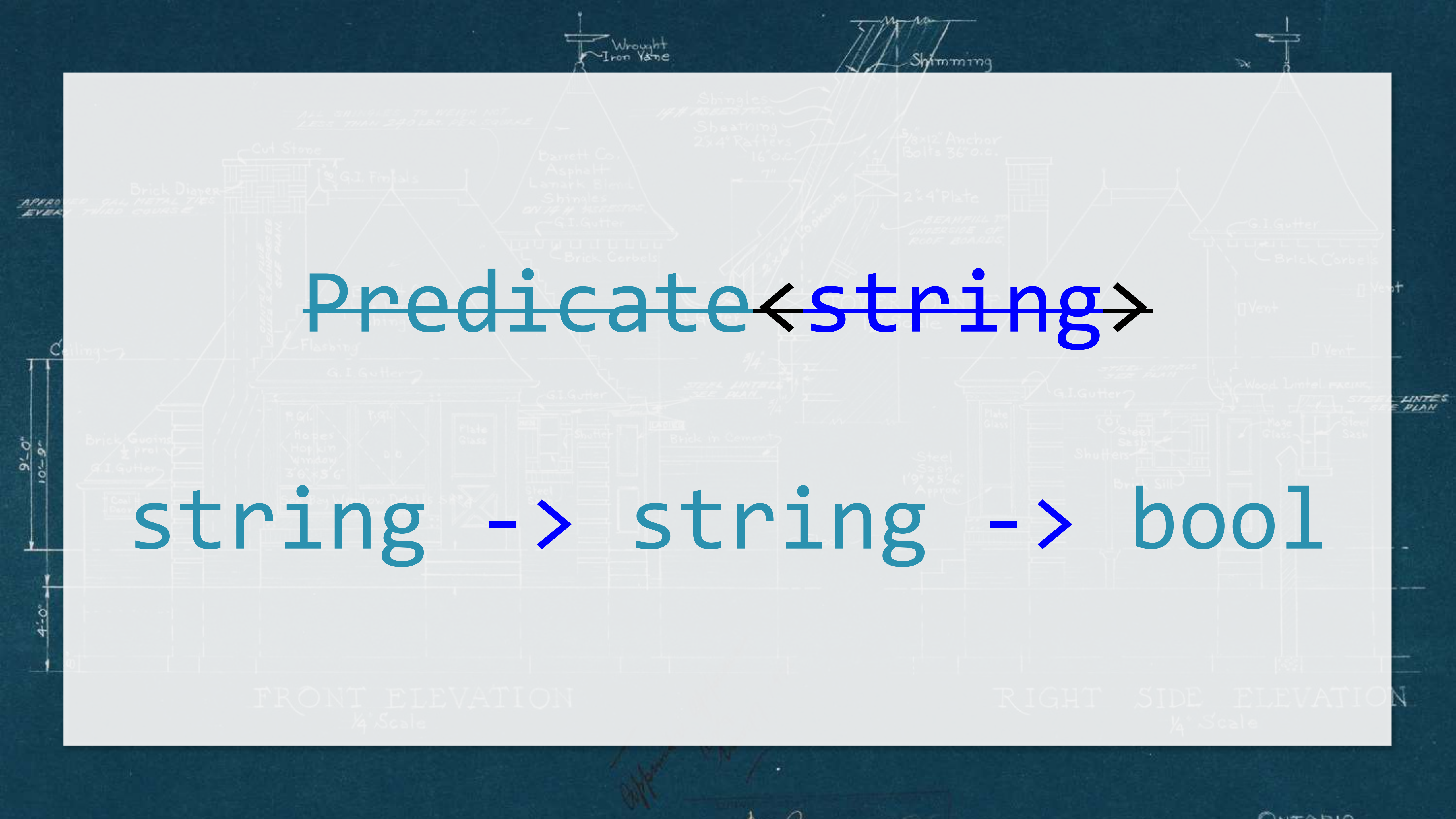
amet adipiscing



C#

STRATEGY

```csharp
public class StartsWith
{
    public string Prefix { get; private set; }
    public StartsWith(string prefix)
    {
        Prefix = prefix;
    }
    public bool Test(string sample)
    {
        return sample.StartsWith(Prefix);
    }
}
```



~~Predicate<string>~~

string -> string -> bool

F#

STRATEGY

```
let startsWith prefix (str : string) =
    str.StartsWith prefix

let words = [ "Lorem";      "ipsum"
              "dolor";      "sit"
              "amet";       "consectetur"
              "adipiscing"; "elit" ]
```

F#

STRATEGY

```
let startsWithL = startsWith "L"
words
|> List.filter startsWithL
|> List.iter (printfn "%s")

let startsWithA = startsWith "a"
words
|> List.filter startsWithA
|> List.iter (printfn "%s")
```

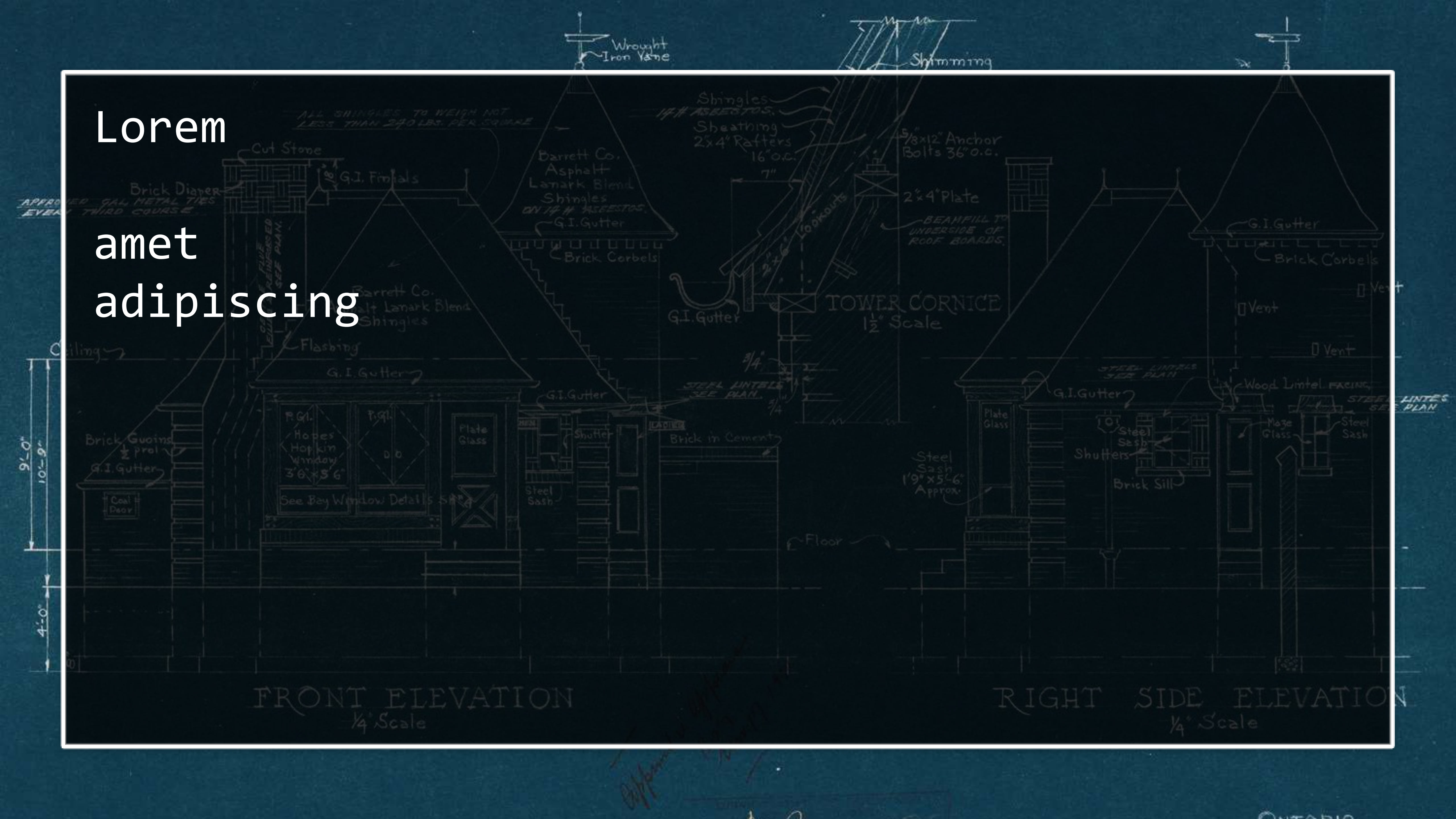
F#

STRATEGY

```
words
|> List.filter (startsWith "L")
|> List.iter (printfn "%s")

words
|> List.filter (startsWith "a")
|> List.iter (printfn "%s")
```

Lorem

amet adipiscing



STRATEGY

C#

```
public class StartsWith
{
    public string Prefix { get; private set; }
    public StartsWith(string prefix)
    {
        Prefix = prefix;
    }
    public bool Test(string sample)
    {
        return sample.StartsWith(Prefix);
    }
}
```

F#

```
let startsWith prefix (str : string) =
    str.StartsWith prefix
```

C#

AN OBSERVATION

```
public class ComplexReImAscComparer : IComparer<Complex>
{
    public int Compare(Complex x, Complex y)
    {
        var re = Math.Sign(x.Re - y.Re);
        return re != 0 ? re : Math.Sign(x.Im - y.Im);
    }
}
```

Just a function

C#

AN OBSERVATION

```
public class StartsWith
{
    public string Prefix { get; private set; }
    public StartsWith(string prefix)
    {
        Prefix = prefix;
    }
    public bool Test(string sample)
    {
        return sample.StartsWith(Prefix);
    }
}
```

A little state

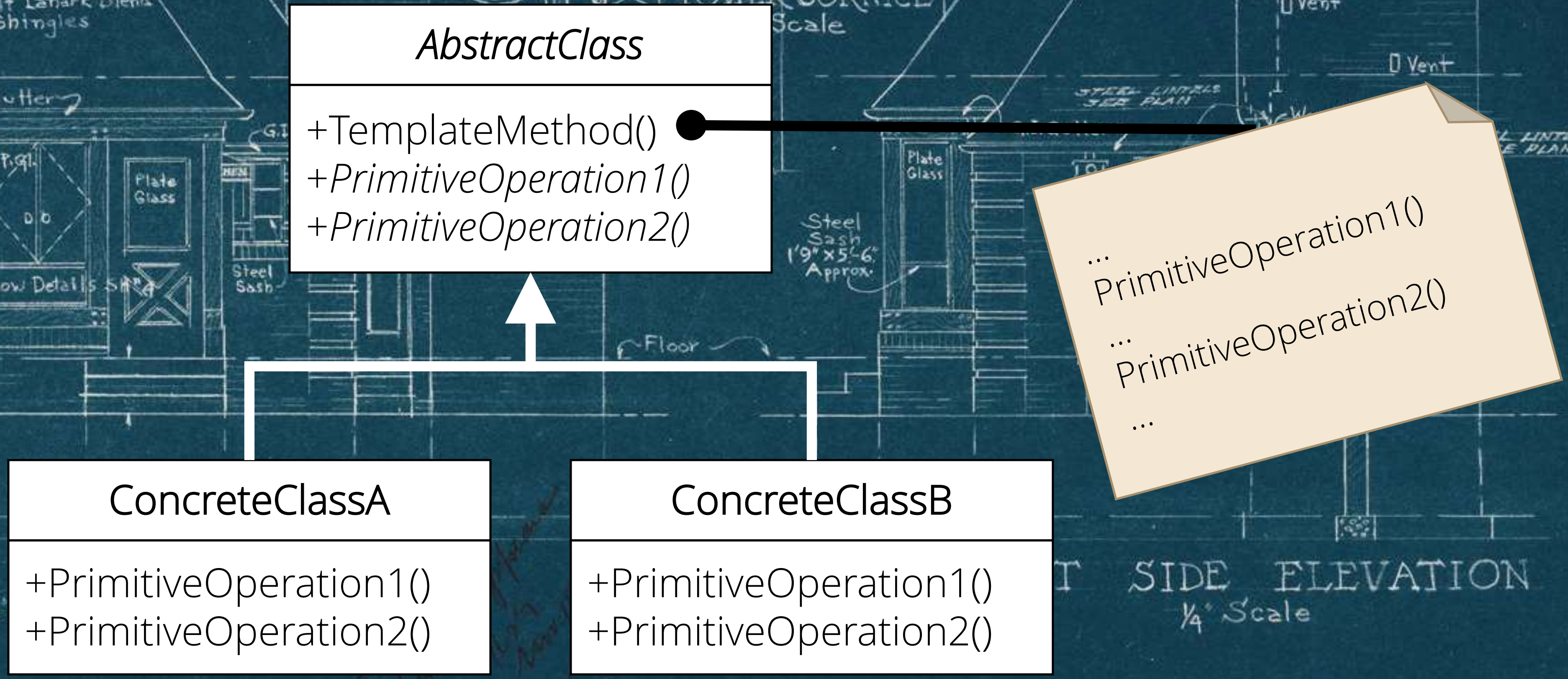
Just a function

> If you have a class with two methods, and one of them is the constructor, you have **a function**.

— @JackDied via @ReidNEvans

emphasis mine

TEMPLATE METHOD



AbstractClass

+TemplateMethod()
+*PrimitiveOperation1()*
+*PrimitiveOperation2()*

ConcreteClassA

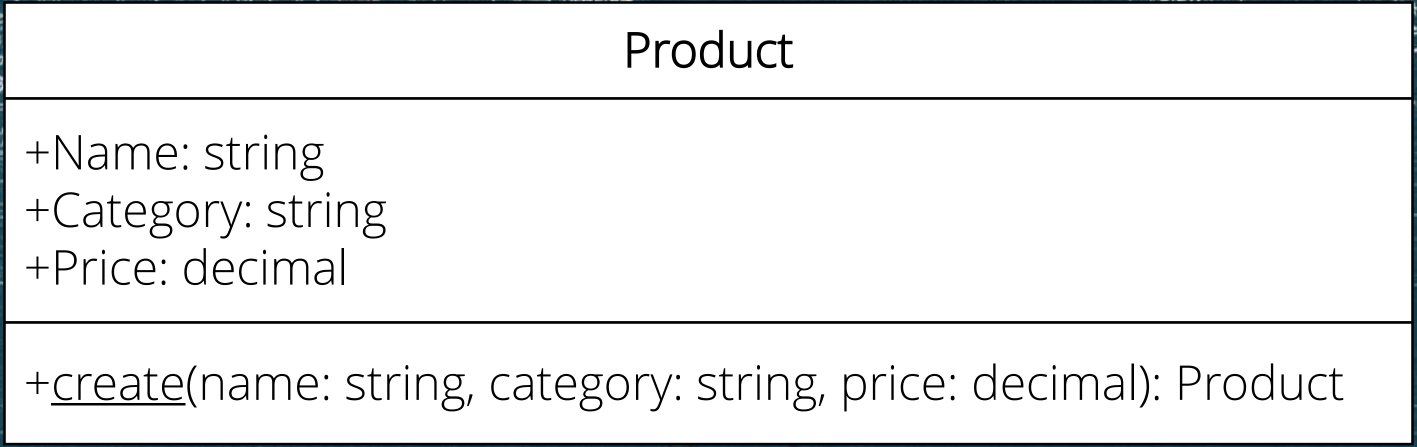
+PrimitiveOperation1()
+PrimitiveOperation2()

ConcreteClassB

+PrimitiveOperation1()
+PrimitiveOperation2()

…
PrimitiveOperation1()
…
PrimitiveOperation2()
…

TEMPLATE METHOD

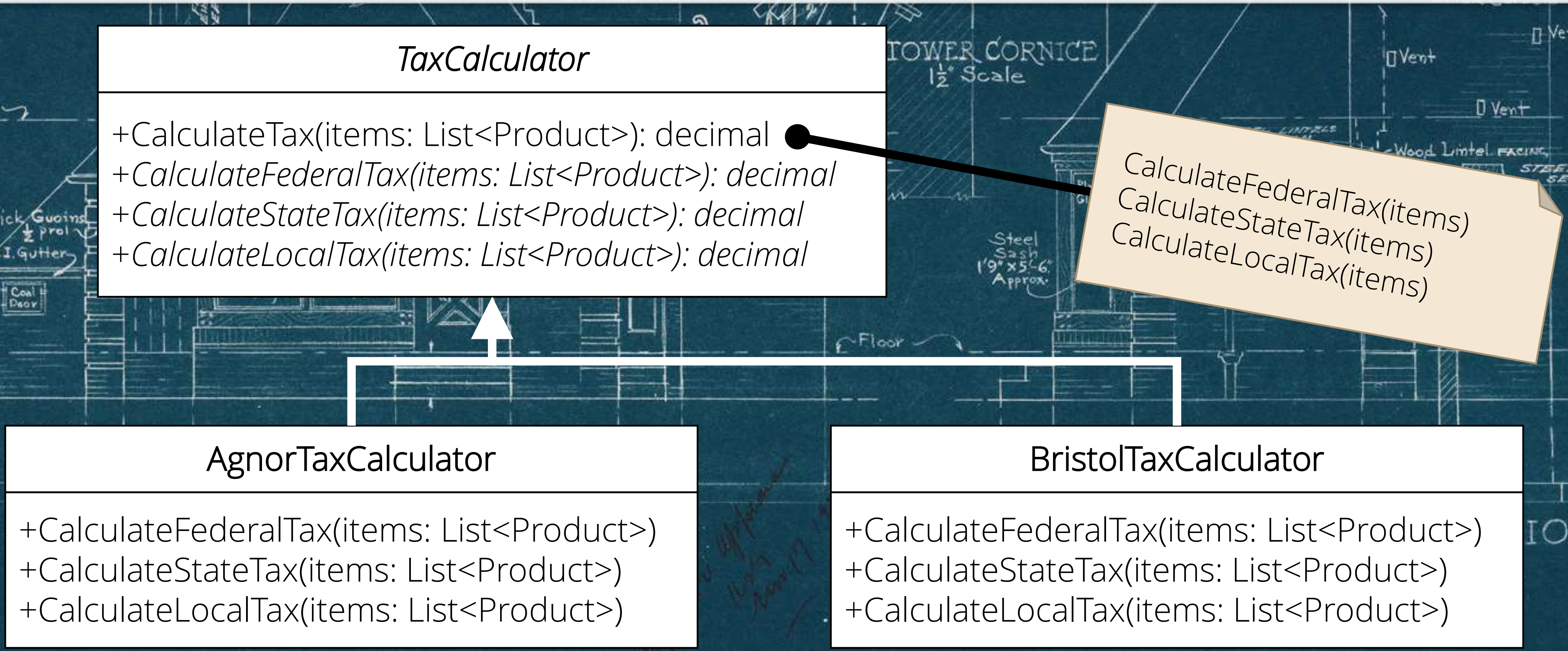


Product

+Name: string
+Category: string
+Price: decimal

+create(name: string, category: string, price: decimal): Product

TEMPLATE METHOD



TaxCalculator

+CalculateTax(items: List<Product>): decimal
+CalculateFederalTax(items: List<Product>): decimal
+CalculateStateTax(items: List<Product>): decimal
+CalculateLocalTax(items: List<Product>): decimal

CalculateFederalTax(items)
CalculateStateTax(items)
CalculateLocalTax(items)

AgnorTaxCalculator

+CalculateFederalTax(items: List<Product>)
+CalculateStateTax(items: List<Product>)
+CalculateLocalTax(items: List<Product>)

BristolTaxCalculator

+CalculateFederalTax(items: List<Product>)
+CalculateStateTax(items: List<Product>)
+CalculateLocalTax(items: List<Product>)

C#

TEMPLATE METHOD

```csharp
public struct Product
{
    public readonly string Name;
    public readonly string Category;
    public readonly decimal Price;
    public Product(string name, string category, decimal price)
    {
        Name = name;
        Category = category;
        Price = price;
    }
}
```

C#

TEMPLATE METHOD

```
public abstract class TaxCalculator
{
    public decimal CalculateTax(List<Product> items)
    {
        return CalculateFederalTax(items)
            + CalculateStateTax(items)
            + CalculateLocalTax(items);
    }
    public abstract decimal CalculateFederalTax(List<Product> items);
    public abstract decimal CalculateStateTax(List<Product> items);
    public abstract decimal CalculateLocalTax(List<Product> items);
}
```

C#

TEMPLATE METHOD

```csharp
public class AgnorTaxCalculator : TaxCalculator
{
    public override decimal CalculateFederalTax(List<Product> items)
    {
        return 0;
    }
    public override decimal CalculateStateTax(List<Product> items)
    {
        const decimal rate = 0.06m;
        var subtotal = 0.0m;
        foreach (var item in items) subtotal += item.Price;
        return subtotal * rate;
    }
}
```

C#

TEMPLATE METHOD

```
public class AgnorTaxCalculator : TaxCalculator
{
    // ...

    public override decimal CalculateLocalTax(List<Product> items)
    {
        const decimal rate = 0.03m;
        var subtotal = 0.0m;
        foreach (var item in items) subtotal += item.Price;
        return subtotal * rate;
    }
}
```

TEMPLATE METHOD

C#

```
public class BristolTaxCalculator : TaxCalculator
{
    public override decimal CalculateFederalTax(List<Product> items)
    {
        const decimal rate = 0.20m;
        var subtotal = 0.0m;
        foreach (var item in items) subtotal += item.Price;
        return subtotal * rate;
    }
    public override decimal CalculateStateTax(List<Product> items)
    {
        return 0;
    }
}
```

C#

TEMPLATE METHOD

```
public class BristolTaxCalculator : TaxCalculator
{
    // ...

    public override decimal CalculateLocalTax(List<Product> items)
    {
        const decimal rate = 0.015m;
        var subtotal = 0.0m;
        foreach (var item in items)
        {
            if (item.Category == "Food" || item.Category == "Medical") continue;
            subtotal += item.Price;
        }
        return subtotal * rate;
    }
}
```

C#

TEMPLATE METHOD

```
var items = new List<Product>()
{
    new Product("Fan",               "Appliance",   19.99m),
    new Product("Nexium",            "Medical",     69.99m),
    new Product("Chicken Thighs",    "Food",         7.99m),
    new Product("Corn Flakes",       "Food",         4.99m),
    new Product("Bed Sheets",        "Linen",      129.99m),
    new Product("Adjustable Wrench", "Hardware",     6.99m),
};
```

C#

TEMPLATE METHOD

```
var subtotal = 0.0m;
foreach (var item in items) subtotal += item.Price;

var agnorTax = new AgnorTaxCalculator().CalculateTax(items);
var bristolTax = new BristolTaxCalculator()
    .CalculateTax(items);

Console.WriteLine("Tax on ${0} of goods in Agnor: ${1:0.00}",
    subtotal, agnorTax);
Console.WriteLine("Tax on ${0} of goods in Bristol: ${1:0.00}",
    subtotal, bristolTax);
```

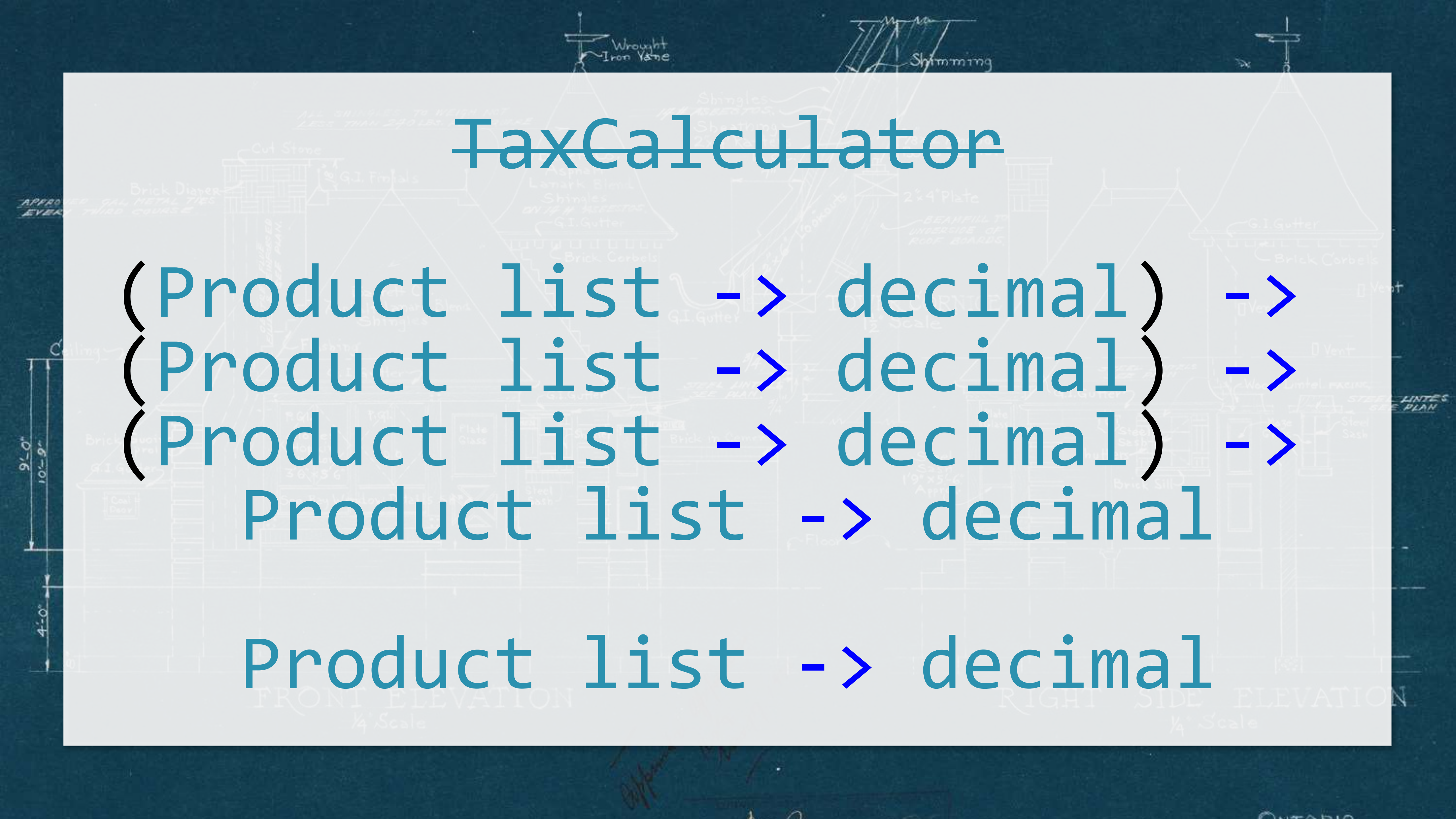
Tax on $239.94 of goods in Agnor: $21.59

Tax on $239.94 of goods in Bristol: $50.34

C#

TEMPLATE METHOD

```
public abstract class TaxCalculator
{
    public decimal CalculateTax(List<Product> items)
    {
        return CalculateFederalTax(items)
            + CalculateStateTax(items)
            + CalculateLocalTax(items);
    }
    public abstract decimal CalculateFederalTax(List<Product> items);
    public abstract decimal CalculateStateTax(List<Product> items);
    public abstract decimal CalculateLocalTax(List<Product> items);
}
public class AgnorTaxCalculator : TaxCalculator
{
    public override decimal CalculateFederalTax(List<Product> items)
    {
        return 0;
    }
    public override decimal CalculateStateTax(List<Product> items)
    {
        const decimal rate = 0.06m;
        var subtotal = 0.0m;
        foreach (var item in items) subtotal += item.Price;
        return subtotal * rate;
    }
    public override decimal CalculateLocalTax(List<Product> items)
    {
        const decimal rate = 0.03m;
        var subtotal = 0.0m;
        foreach (var item in items) subtotal += item.Price;
        return subtotal * rate;
    }
}
```



~~TaxCalculator~~

```
(Product list -> decimal) ->
(Product list -> decimal) ->
(Product list -> decimal) ->
  Product list -> decimal

  Product list -> decimal
```

F#

TEMPLATE METHOD

```fsharp
type Product =
  { Name : string
    Category : string
    Price : decimal }

let calculateTax calcFedTax calcStateTax calcLocalTax items =
    calcFedTax items +
    calcStateTax items +
    calcLocalTax items
```

F#

TEMPLATE METHOD

```
let getPrice item = item.Price
let taxAt rate subtotal = rate * subtotal
let buildTaxCalculator exemptCategories rate =
    let isTaxable item = exemptCategories
                         |> List.contains item.Category
                         |> not

    Seq.filter isTaxable
    >> Seq.map getPrice
    >> Seq.sum
    >> taxAt rate
```

F#

TEMPLATE METHOD

```fsharp
let calculateTaxInAgnor =
    let calcFedTax _ = 0.0m
    let calcStateTax = buildTaxCalculator [] 0.06m
    let calcLocalTax = buildTaxCalculator [] 0.03m

    calculateTax calcFedTax calcStateTax calcLocalTax
```

F#

TEMPLATE METHOD

```fsharp
let calculateTaxInBristol =
    let calcFedTax = buildTaxCalculator [] 0.20m
    let calcStateTax _ = 0.0m
    let calcLocalTax =
        buildTaxCalculator ["Food"; "Medical"] 0.015m

    calculateTax calcFedTax calcStateTax calcLocalTax
```

TEMPLATE METHOD

F#

```
let items = [
  { Name="Fan";               Category="Appliance"; Price= 19.99m }
  { Name="Nexium";            Category="Medical";   Price= 69.99m }
  { Name="Chicken Thighs";    Category="Food";      Price=  7.99m }
  { Name="Corn Flakes";       Category="Food";      Price=  4.99m }
  { Name="Bed Sheets";        Category="Linen";     Price=129.99m }
  { Name="Adjustable Wrench"; Category="Hardware";  Price=  6.99m }
]
```

F#

TEMPLATE METHOD

```
let subtotal = items
               |> Seq.map getPrice
               |> Seq.sum

let agnorTax = calculateTaxInAgnor items
let bristolTax = calculateTaxInBristol items

printfn "Tax on $%.2f of goods in Agnor: $%.2f"
    subtotal agnorTax
printfn "Tax on $%.2f of goods in Bristol: $%.2f"
    subtotal bristolTax
```

Tax on $239.94 of goods in Agnor: $21.59

Tax on $239.94 of goods in Bristol: $50.34

TEMPLATE METHOD

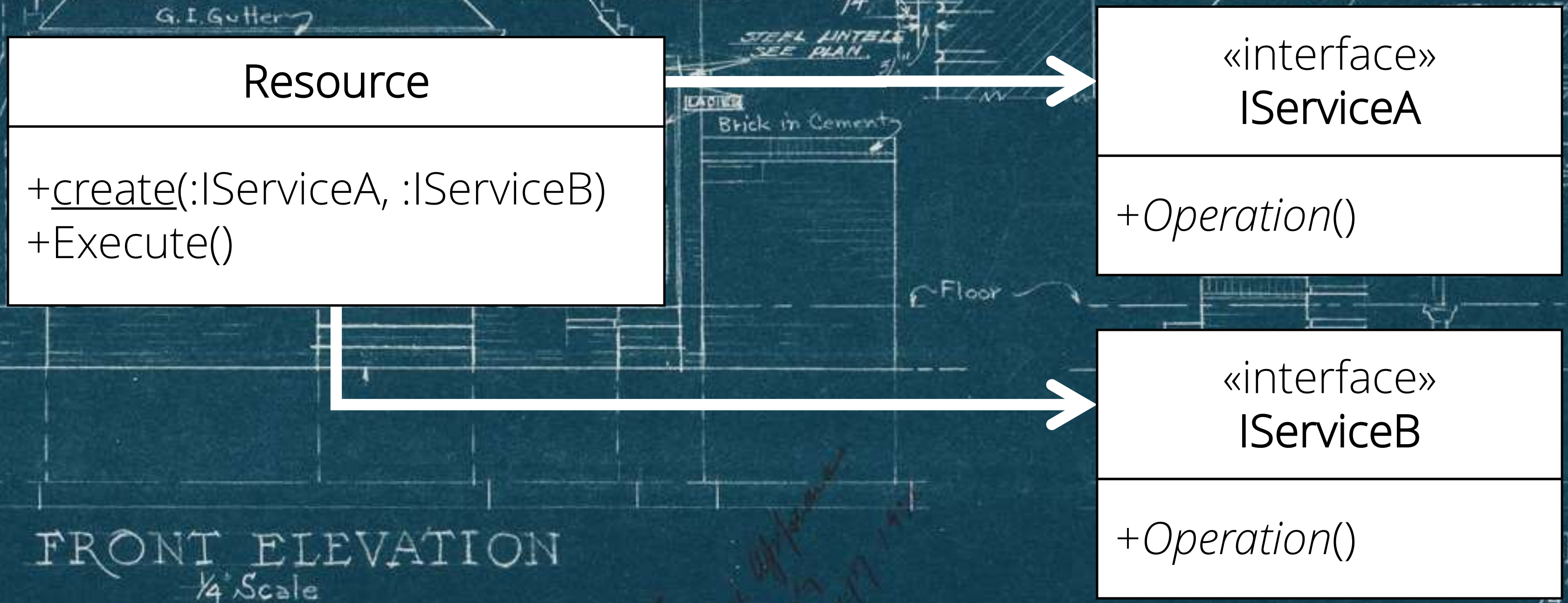
C#

```csharp
public abstract class TaxCalculator
{
    public decimal CalculateTax(List<Product> items)
    {
        return CalculateFederalTax(items)
            + CalculateStateTax(items)
            + CalculateLocalTax(items);
    }
    public abstract decimal CalculateFederalTax(List<Product> items);
    public abstract decimal CalculateStateTax(List<Product> items);
    public abstract decimal CalculateLocalTax(List<Product> items);
}
public class AgnorTaxCalculator : TaxCalculator
{
    public override decimal CalculateFederalTax(List<Product> items)
    {
        return 0;
    }
    public override decimal CalculateStateTax(List<Product> items)
    {
        const decimal rate = 0.06m;
        var subtotal = 0.0m;
        foreach (var item in items) subtotal += item.Price;
        return subtotal * rate;
    }
    public override decimal CalculateLocalTax(List<Product> items)
    {
        const decimal rate = 0.03m;
        var subtotal = 0.0m;
        foreach (var item in items) subtotal += item.Price;
        return subtotal * rate;
    }
}
public class BristolTaxCalculator : TaxCalculator
{
    public override decimal CalculateFederalTax(List<Product> items)
    {
        const decimal rate = 0.20m;
        var subtotal = 0.0m;
        foreach (var item in items) subtotal += item.Price;
        return subtotal * rate;
    }
    public override decimal CalculateStateTax(List<Product> items)
    {
        return 0;
    }
    public override decimal CalculateLocalTax(List<Product> items)
    {
        const decimal rate = 0.015m;
        var subtotal = 0.0m;
        foreach (var item in items)
        {
            if (item.Category == "Food" || item.Category == "Medical") continue;
            subtotal += item.Price;
        }
        return subtotal * rate;
    }
}
```

F#

```fsharp
let calculateTax calcFedTax calcStateTax calcLocalTax items =
    calcFedTax items +
    calcStateTax items +
    calcLocalTax items
let getPrice item = item.Price
let taxAt rate subtotal = rate * subtotal
let buildTaxCalculator exemptCategories rate =
    let isTaxable item = exemptCategories
                         |> List.contains item.Category
                         |> not

    Seq.filter isTaxable
    >> Seq.map getPrice
    >> Seq.sum
    >> taxAt rate
let calculateTaxInAgnor =
     let calcFedTax _ = 0.0m
     let calcStateTax = buildTaxCalculator [] 0.06m
     let calcLocalTax = buildTaxCalculator [] 0.03m

     calculateTax calcFedTax calcStateTax calcLocalTax
 let calculateTaxInBristol =
    let calcFedTax = buildTaxCalculator [] 0.20m
    let calcStateTax _ = 0.0m
    let calcLocalTax =
        buildTaxCalculator ["Food"; "Medical"] 0.015m

    calculateTax calcFedTax calcStateTax calcLocalTax
```

DEPENDENCY INJECTION



Resource

+create(:IServiceA, :IServiceB)
+Execute()

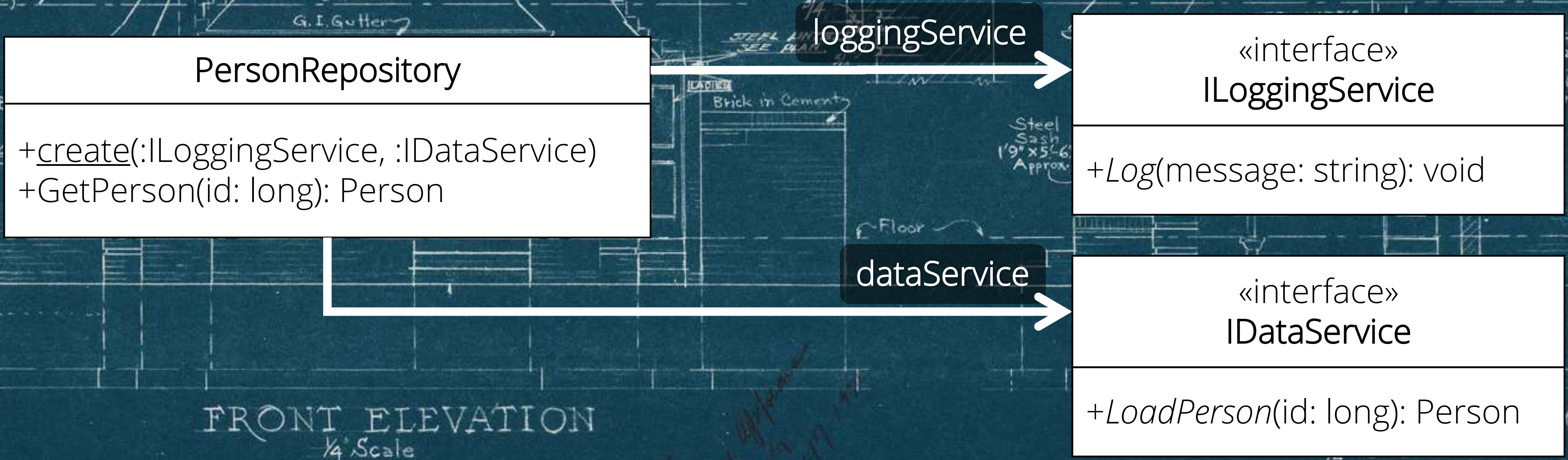
«interface»
IServiceA

+*Operation*()

«interface»
IServiceB

+*Operation*()

DEPENDENCY INJECTION



PersonRepository

+create(:ILoggingService, :IDataService)
+GetPerson(id: long): Person

loggingService

dataService

«interface»
ILoggingService

+*Log*(message: string): void

«interface»
IDataService

+*LoadPerson*(id: long): Person

C#

DEPENDENCY INJECTION

```csharp
public class PersonRepository
{
    private readonly ILoggingService loggingSvc;
    private readonly IDataService dataSvc;
    public PersonRepository(ILoggingService loggingSvc,
                            IDataService dataSvc)
    {
        this.loggingSvc = loggingSvc;
        this.dataSvc = dataSvc;
    }
}
```

C#

DEPENDENCY INJECTION

```
public class PersonRepository
{
    // ...

    public Person GetPerson(long id)
    {
        var person = dataSvc.LoadPerson(id);
        loggingSvc.Log($"Loaded Person: {person?.ToString() ?? "(null)"}");
        return person;
    }
}
```

C#

DEPENDENCY INJECTION

```
public interface ILoggingService
{
    void Log(string message);
}

public class ConsoleLogger : ILoggingService
{
    public void Log(string message)
    {
        Console.WriteLine(message);
    }
}
```

C#

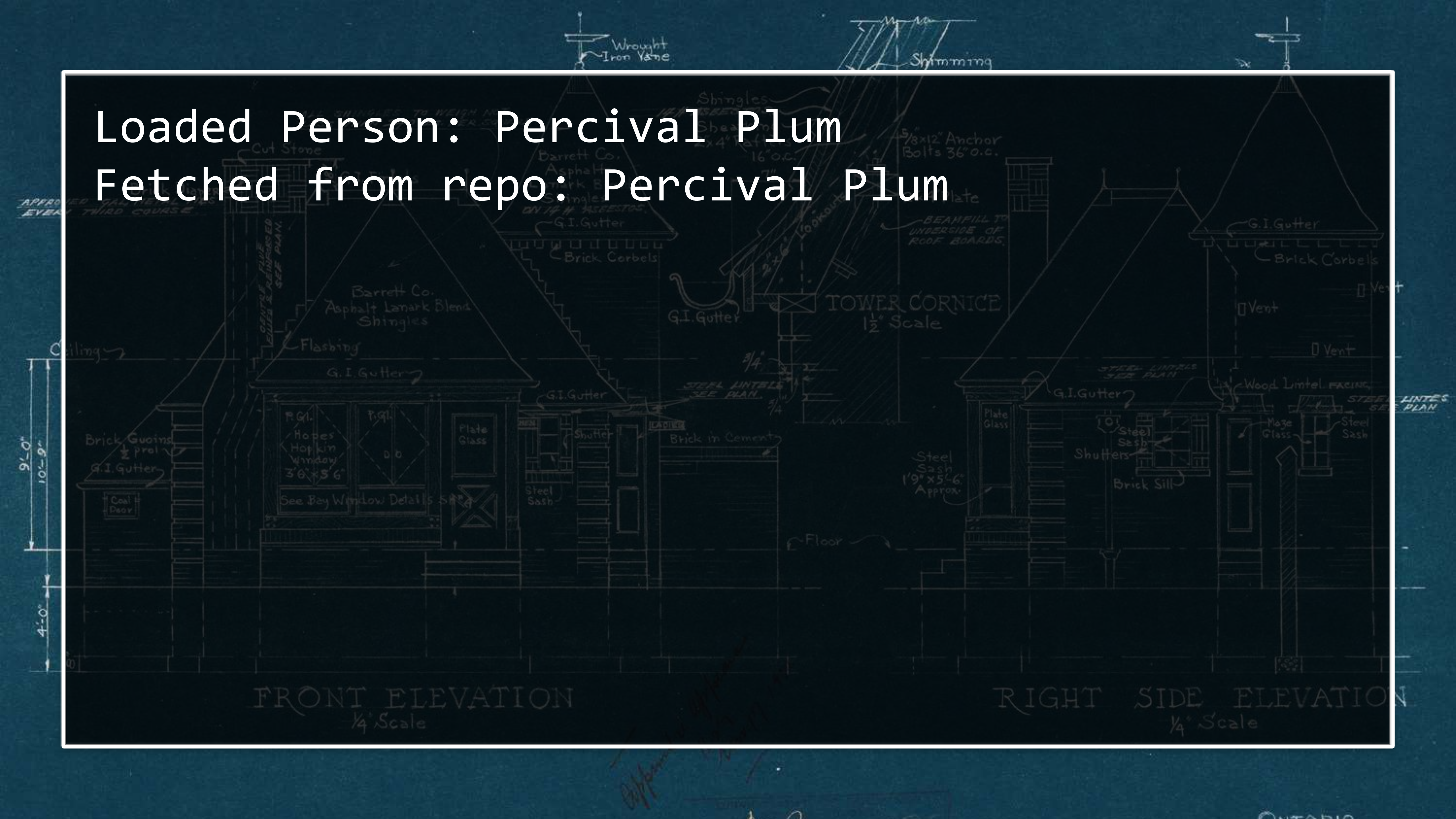
DEPENDENCY INJECTION

```
public interface IDataService
{
    Person LoadPerson(long id);
}

public class SimpleDataService : IDataService
{
    private readonly List<Person> persons = new List<Person>() { /* ... */ };
    public Person LoadPerson(long id)
    {
        return id < persons.Count ? persons[(int)id] : null;
    }
}
```

C#

DEPENDENCY INJECTION

```
var loggerSvc = new ConsoleLogger();
var dataSvc = new SimpleDataService();
var repo = new PersonRepository(loggerSvc, dataSvc);

var person = repo.GetPerson(5);

Console.WriteLine($"Fetched from repo: {person}");
```



```
Loaded Person: Percival Plum
Fetched from repo: Percival Plum
```

C#

DEPENDENCY INJECTION

```
public class PersonRepository
{
    private readonly ILoggingService loggingSvc;
    private readonly IDataService dataSvc;
    public PersonRepository(ILoggingService loggingSvc, IDataService dataSvc)
    {
        this.loggingSvc = loggingSvc;
        this.dataSvc = dataSvc;
    }
    public Person GetPerson(long id)
    {
        var person = dataSvc.LoadPerson(id);
        loggingSvc.Log($"Loaded Person: {person?.ToString() ?? "(null)"}");
        return person;
    }
}
public class ConsoleLogger : ILoggingService
{
    public void Log(string message)
    {
        Console.WriteLine(message);
    }
}
public class SimpleDataService : IDataService
{
    private readonly List<Person> persons = new List<Person>() { /* ... */ };
    public Person LoadPerson(long id)
    {
        return id < persons.Count ? persons[(int)id] : null;
    }
}
```

~~PersonRepository~~

```
(string -> unit) ->
(long -> Person option) ->
long -> Person option

long -> Person option
```

DEPENDENCY INJECTION

F#

```
let repository log loadPerson id =
    let person = loadPerson id
    log (sprintf "Loaded Person: %O" person)
    person

let logger = printfn "%s"
let dataSvc id =
    [  (* ... *) ]
    |> List.tryItem id
```

F#

DEPENDENCY INJECTION

```
let getPerson = repository logger dataSvc
let person = getPerson 5
printfn "Fetched from repo: %O" person
```

```
Loaded Person: Some(Percival Plum)
Fetched from repo: Some(Percival Plum)
```

DEPENDENCY INJECTION

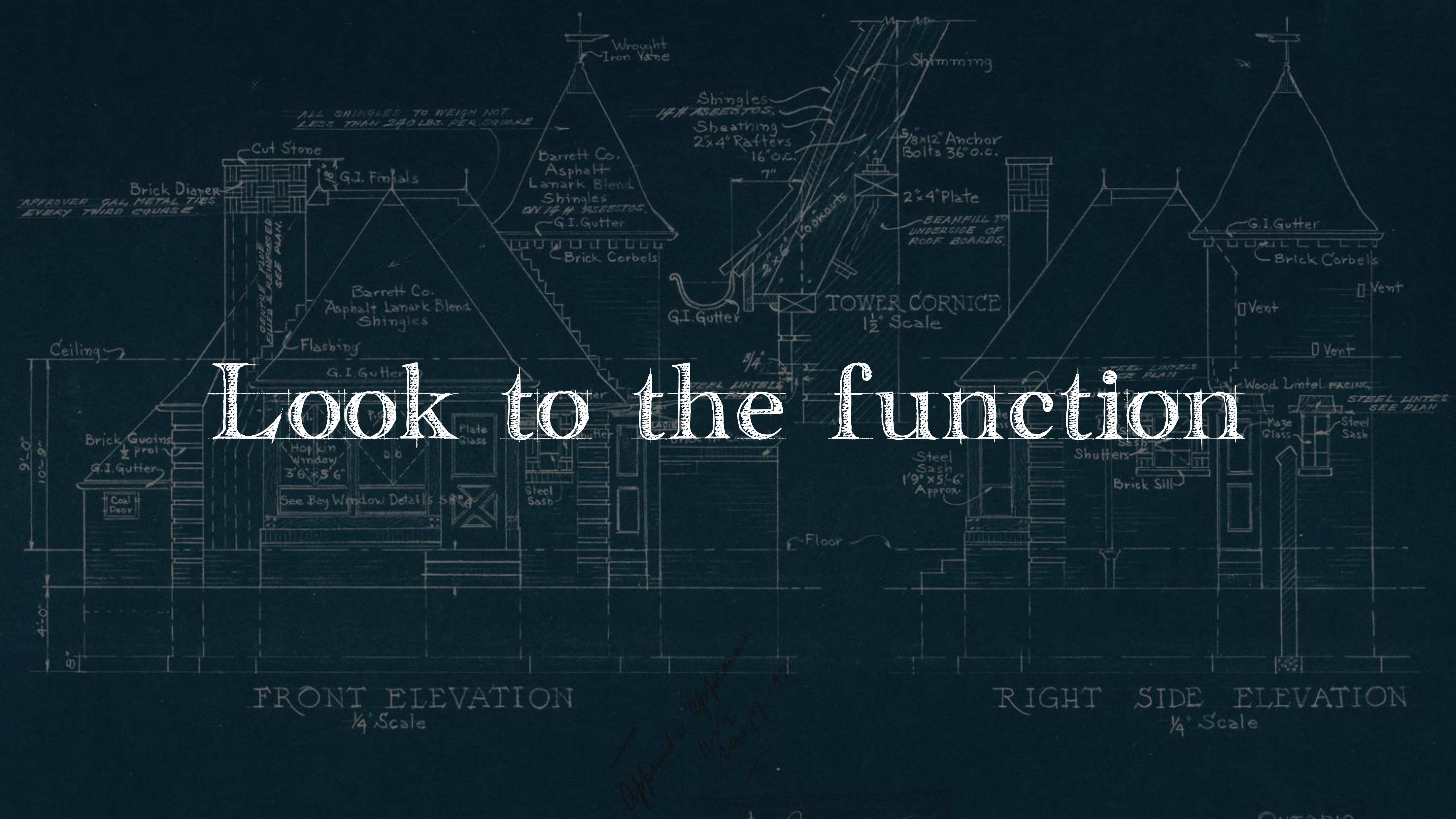
C#

```
public interface ILoggingService
{
    void Log(string message);
}
public interface IDataService
{
    Person LoadPerson(long id);
}
public class PersonRepository
{
    private readonly ILoggingService loggingSvc;
    private readonly IDataService dataSvc;
    public PersonRepository(ILoggingService loggingSvc, IDataService dataSvc)
    {
        this.loggingSvc = loggingSvc;
        this.dataSvc = dataSvc;
    }
    public Person GetPerson(long id)
    {
        var person = dataSvc.LoadPerson(id);
        loggingSvc.Log($"Loaded Person: {person?.ToString() ?? "(null)"}");
        return person;
    }
}
public class ConsoleLogger : ILoggingService
{
    public void Log(string message)
    {
        Console.WriteLine(message);
    }
}
public class SimpleDataService : IDataService
{
    private readonly List<Person> persons = new List<Person>() { /* ... */ };
    public Person LoadPerson(long id)
    {
        return id < persons.Count ? persons[(int)id] : null;
    }
}
```

F#

```
let repository log loadPerson id =
    let person = loadPerson id
    log (sprintf "Loaded Person: %O" person)
    person

let logger = printfn "%s“
let dataSvc id =
    [ (* ... *) ]
    |> List.tryItem id
```

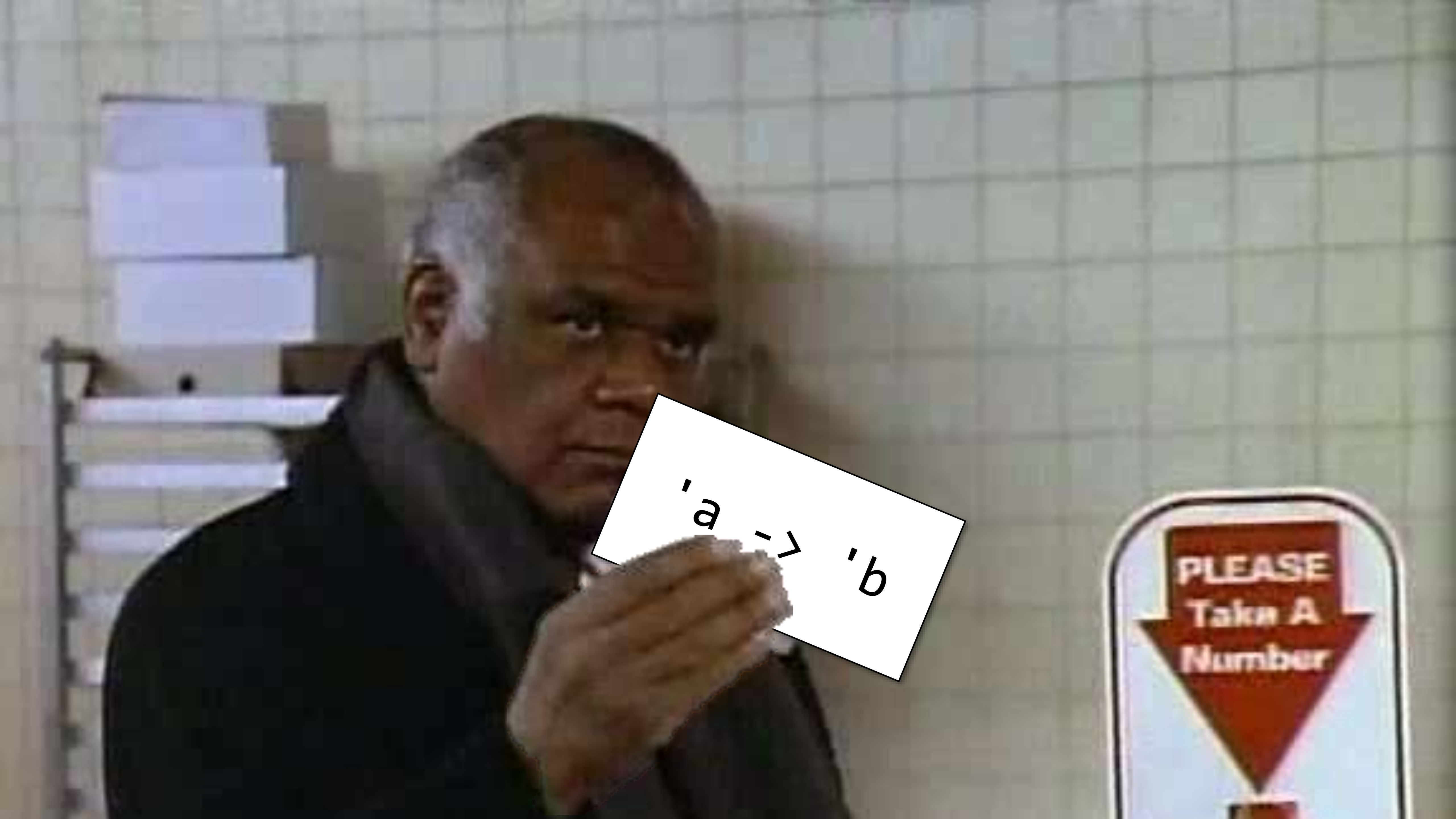
Look to the function

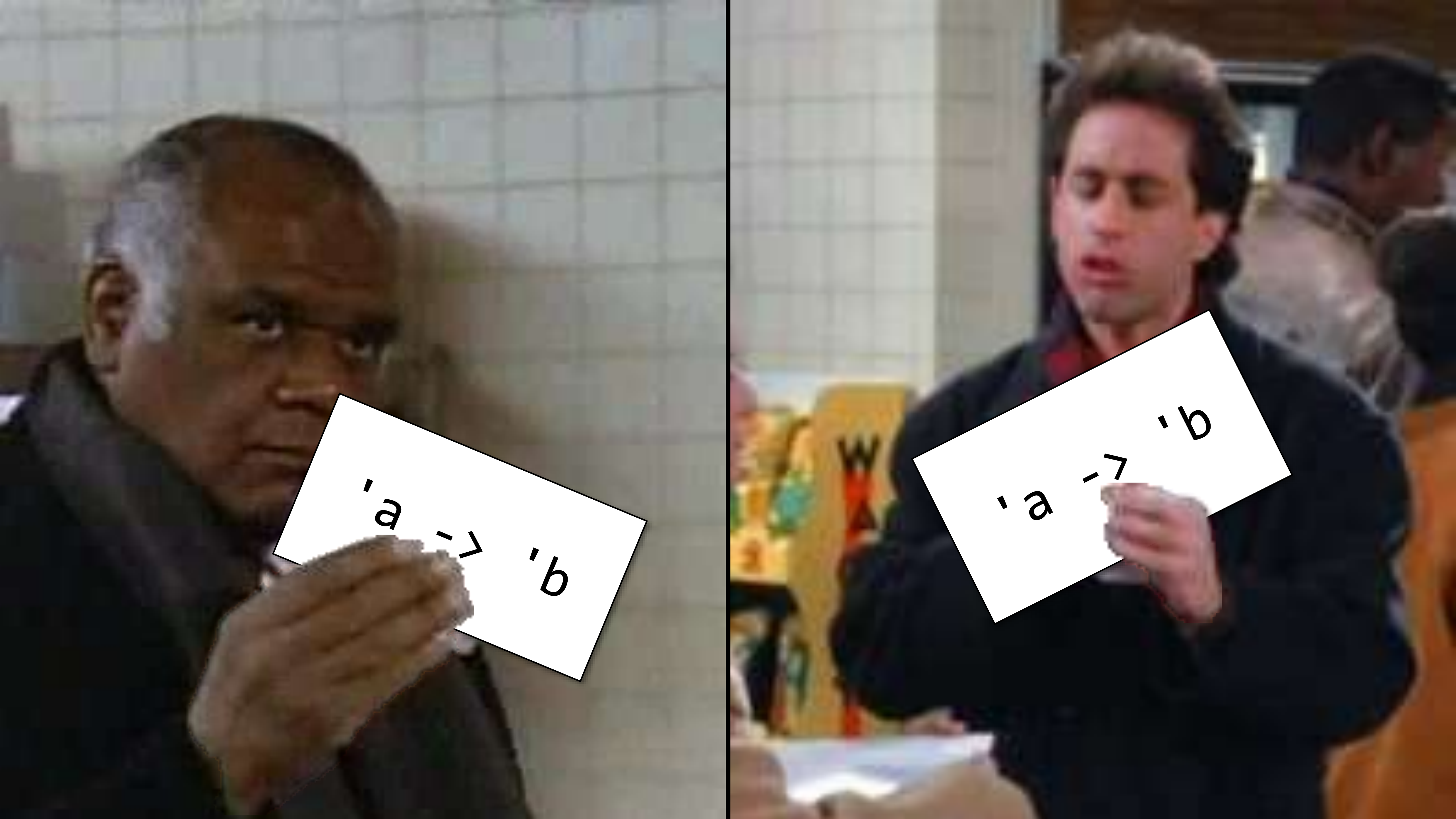


LOOK TO THE FUNCTION

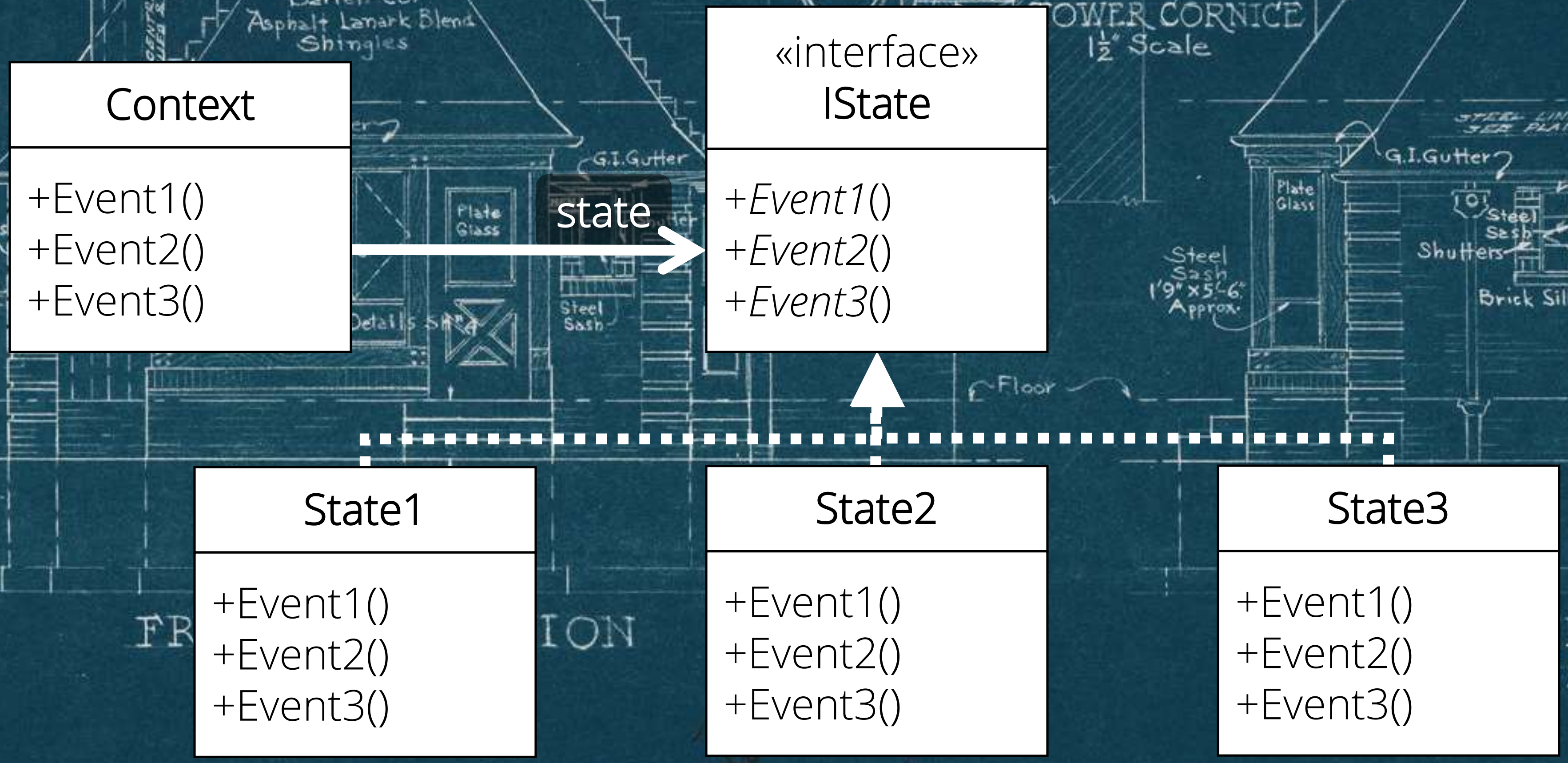
- Pass around as a first-class value
- Load with state/functionality via partial application
- No class baggage: Just a function signature
- Easier testing: Match a function signature, not a class/interface

'a -> 'b


'a -> 'b
PLEASE Take A Number

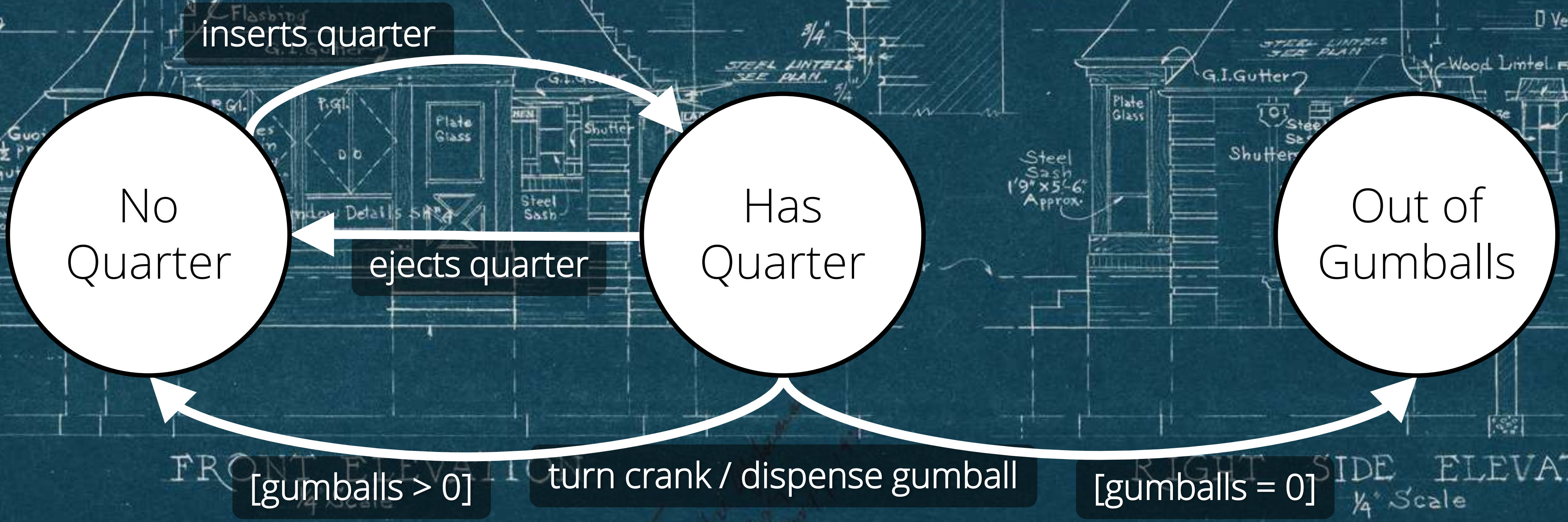

'a -> 'b
'a -> 'b

STATE

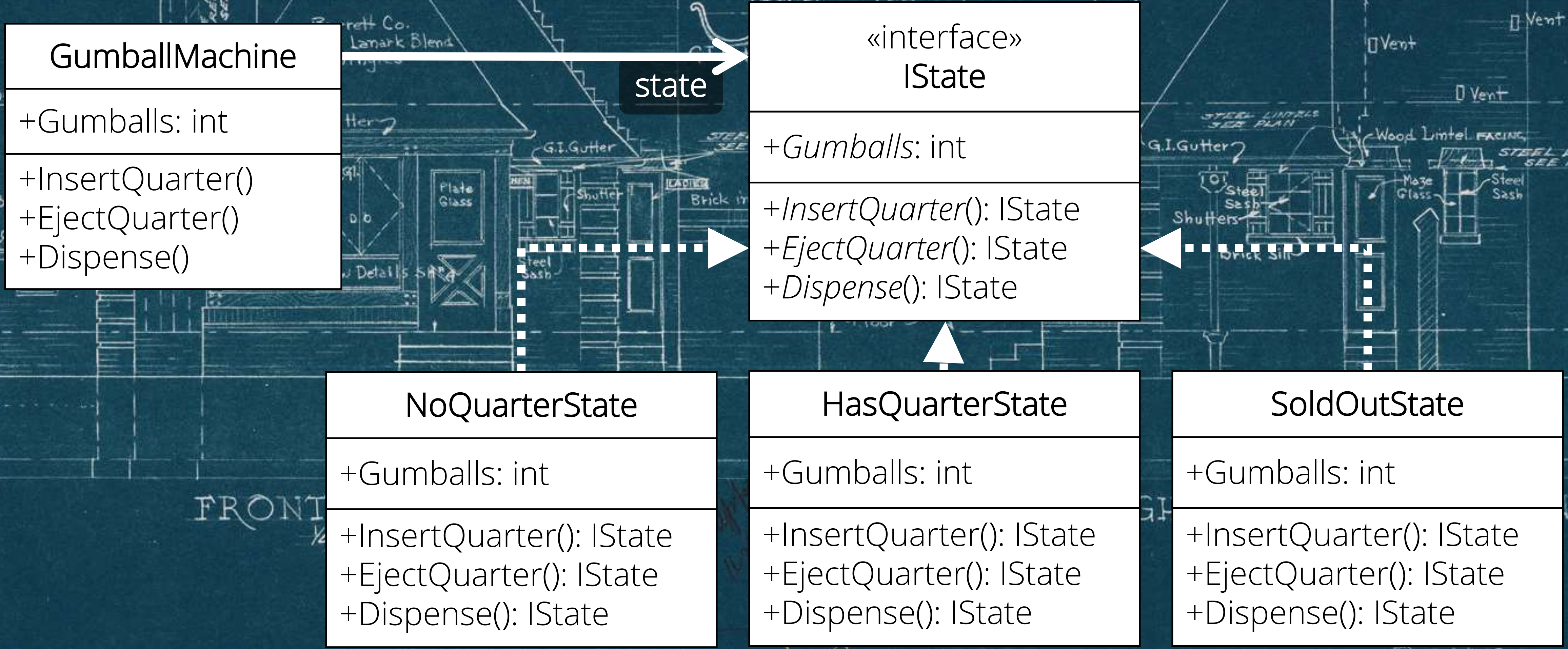


Context
+Event1()
+Event2()
+Event3()
state
«interface»
IState
+*Event1*()
+*Event2*()
+*Event3*()
State1
+Event1()
+Event2()
+Event3()
State2
+Event1()
+Event2()
+Event3()
State3
+Event1()
+Event2()
+Event3()

STATE


No Quarter
Has Quarter
Out of Gumballs
inserts quarter
ejects quarter
turn crank / dispense gumball
[gumballs > 0]
[gumballs = 0]

STATE



GumballMachine

+Gumballs: int

+InsertQuarter()
+EjectQuarter()
+Dispense()

state

«interface»
IState

+*Gumballs*: int

+*InsertQuarter*(): IState
+*EjectQuarter*(): IState
+*Dispense*(): IState

NoQuarterState

+Gumballs: int

+InsertQuarter(): IState
+EjectQuarter(): IState
+Dispense(): IState

HasQuarterState

+Gumballs: int

+InsertQuarter(): IState
+EjectQuarter(): IState
+Dispense(): IState

SoldOutState

+Gumballs: int

+InsertQuarter(): IState
+EjectQuarter(): IState
+Dispense(): IState

C#

STATE

```
public interface IState
{
    int Gumballs { get; }
    IState InsertQuarter();
    IState EjectQuarter();
    IState Dispense();
}
```

C#

STATE

```
public class GumballMachine
{
    private IState state;
    public GumballMachine(int gumballs)
    {
        this.state = new NoQuarterState(gumballs);
    }

    public int Gumballs { get { return state.Gumballs; } }
}
```

C#

STATE

```csharp
public class GumballMachine
{
    // ...
    public void InsertQuarter() { state = state.InsertQuarter(); }
    public void EjectQuarter()  { state = state.EjectQuarter(); }
    public void Dispense()      { state = state.Dispense(); }
    public override string ToString()
    {
        return $"Gumball Machine with {Gumballs} gumballs";
    }
}
```

C#

STATE

```
public class NoQuarterState : IState
{
    public NoQuarterState(int gumballs) { Gumballs = gumballs; }
    public int Gumballs { get; private set; }
    public IState InsertQuarter()
    {
        return new HasQuarterState(Gumballs);
    }
    public IState EjectQuarter() { return this; }
    public IState Dispense() { return this; }
}
```

C#

STATE

```
public class HasQuarterState : IState
{
    public HasQuarterState(int gumballs) { Gumballs = gumballs; }
    public int Gumballs { get; private set; }
    public IState InsertQuarter() { return this; }
    public IState EjectQuarter()
    {
        return new NoQuarterState(Gumballs);
    }
}
```

C#

STATE

```
public class HasQuarterState : IState
{
    // ...

    public IState Dispense()
    {
        return Gumballs == 1
            ? new SoldOutState() as IState
            : new NoQuarterState(Gumballs - 1);
    }
}
```

C#

STATE

```
public class SoldOutState : IState
{
    public int Gumballs { get { return 0; } }
    public IState InsertQuarter() { return this; }
    public IState EjectQuarter() { return this; }
    public IState Dispense() { return this; }
}
```

C#

STATE

```
var machine = new GumballMachine(3);
machine.InsertQuarter();
Console.WriteLine($"After Insert Quarter: {machine}");
machine.Dispense();
Console.WriteLine($"After Dispense: {machine}");
machine.InsertQuarter();
Console.WriteLine($"After Insert Quarter: {machine}");
machine.EjectQuarter();
Console.WriteLine($"After Eject Quarter: {machine}");
// ...
```

```
After Insert Quarter: Gumball Machine with 3 gumballs
After Dispense: Gumball Machine with 2 gumballs
After Insert Quarter: Gumball Machine with 2 gumballs
After Eject Quarter: Gumball Machine with 2 gumballs
After Dispense: Gumball Machine with 2 gumballs
After Insert Quarter: Gumball Machine with 2 gumballs
After Dispense: Gumball Machine with 1 gumballs
After Insert Quarter: Gumball Machine with 1 gumballs
After Dispense: Gumball Machine with 0 gumballs
After Insert Quarter: Gumball Machine with 0 gumballs
After Dispense: Gumball Machine with 0 gumballs
```

F#

STATE

```
type State =
| NoQuarter of int
| HasQuarter of int
| SoldOut

type Event =
| InsertQuarter
| EjectQuarter
| Dispense
```

F#

STATE

```
let execute event state =
    match event, state with
    | InsertQuarter, NoQuarter n -> HasQuarter n
    | EjectQuarter, HasQuarter n -> NoQuarter n
    | Dispense, HasQuarter gumballs ->
        match gumballs with
        | 1 -> SoldOut
        | _ -> NoQuarter (gumballs - 1)
    | _ -> state
```

F#

STATE

```
let events = [ InsertQuarter
               Dispense
               InsertQuarter
               EjectQuarter
               Dispense
               InsertQuarter
               Dispense
               InsertQuarter
               Dispense
               InsertQuarter
               Dispense ]
```

F#

STATE

```fsharp
let init = EjectQuarter, NoQuarter 3
events
|> Seq.scan (fun current event ->
                let _, state = current
                event, execute event state)
            init
|> Seq.skip 1
|> Seq.iter (fun (evt, state) ->
                printfn "%A -> %A" evt state)
```

```
InsertQuarter -> HasQuarter 3
Dispense -> NoQuarter 2
InsertQuarter -> HasQuarter 2
EjectQuarter -> NoQuarter 2
Dispense -> NoQuarter 2
InsertQuarter -> HasQuarter 2
Dispense -> NoQuarter 1
InsertQuarter -> HasQuarter 1
Dispense -> SoldOut
InsertQuarter -> SoldOut
Dispense -> SoldOut
```

STATE

C#

```
public interface Istate
{
    int Gumballs { get; }
    IState InsertQuarter();
    IState EjectQuarter();
    IState Dispense();
}
public class NoQuarterState : Istate
{
    public NoQuarterState(int gumballs) { Gumballs = gumballs; }
    public int Gumballs { get; private set; }
    public IState InsertQuarter() { return new HasQuarterState(Gumballs); }
    public IState EjectQuarter() { return this; }
    public IState Dispense() { return this; }
}
public class HasQuarterState : Istate
{
    public HasQuarterState(int gumballs) { Gumballs = gumballs; }
    public int Gumballs { get; private set; }
    public IState InsertQuarter() { return this; }
    public IState EjectQuarter() { return new NoQuarterState(Gumballs); }
    public IState Dispense()
    {
        return Gumballs == 1 ? new SoldOutState() as IState : new NoQuarterState(Gumballs - 1);
    }
}
public class SoldOutState : Istate
{
    public int Gumballs { get { return 0; } }
    public IState InsertQuarter() { return this; }
    public IState EjectQuarter() { return this; }
    public IState Dispense() { return this; }
}
public class GumballMachine
{
    private IState state;
    public GumballMachine(int gumballs) { this.state = new NoQuarterState(gumballs); }
    public int Gumballs { get { return state.Gumballs; } }
    public void InsertQuarter() { state = state.InsertQuarter(); }
    public void EjectQuarter() { state = state.EjectQuarter(); }
    public void Dispense() { state = state.Dispense(); }
    public override string ToString() { return $"Gumball Machine with {Gumballs} gumballs"; }
}
```

F#

```
type State =
| NoQuarter of int
| HasQuarter of int
| SoldOut

type Event =
| InsertQuarter
| EjectQuarter
| Dispense

let execute event state =
    match event, state with
    | InsertQuarter, NoQuarter n -> HasQuarter n
    | EjectQuarter, HasQuarter n -> NoQuarter n
    | Dispense, HasQuarter gumballs ->
        match gumballs with
        | 1 -> SoldOut
        | _ -> NoQuarter (gumballs - 1)
    | _ -> state
```

STATE

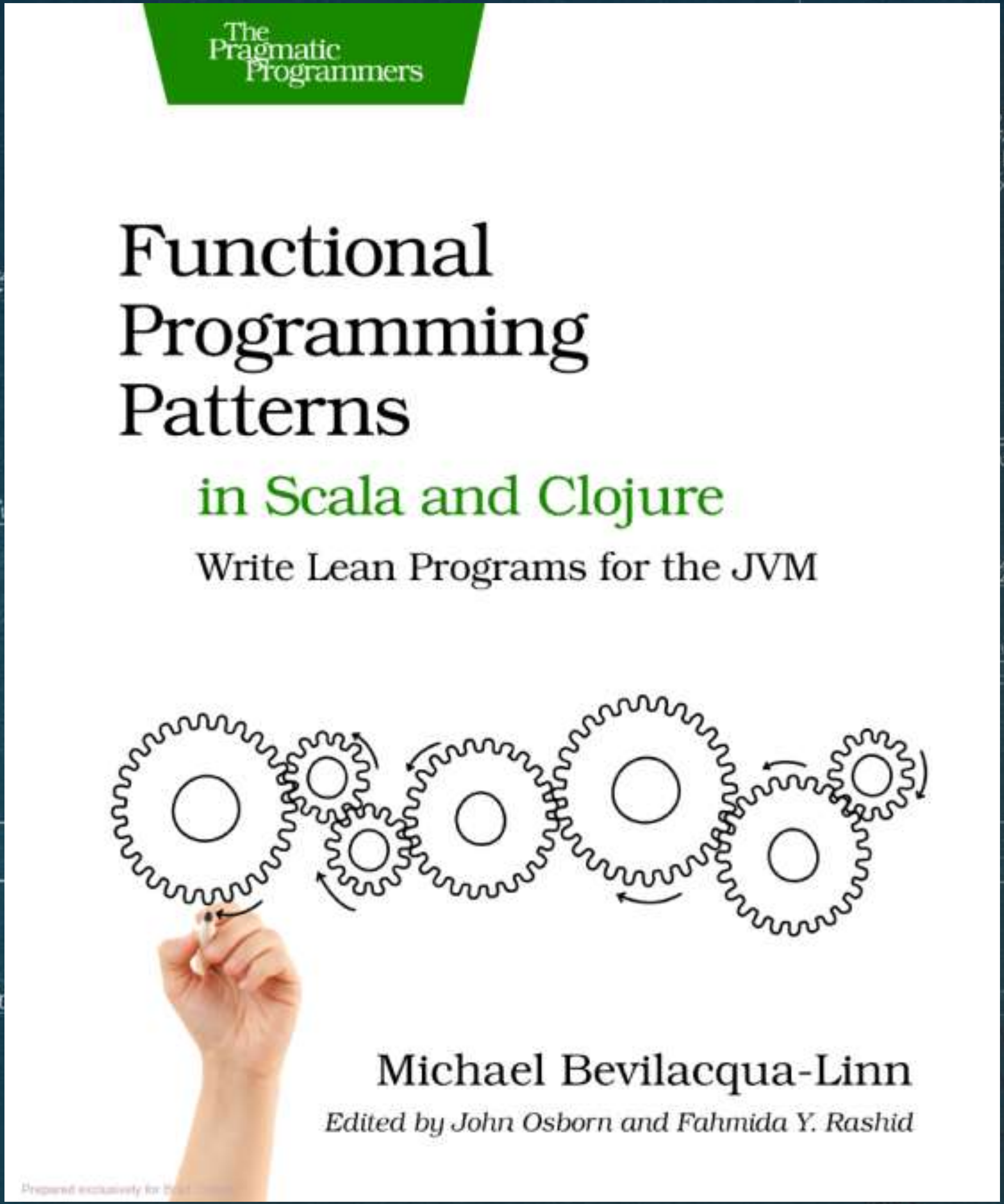
- Discriminated unions:
  - Poor man’s inheritance/polymorphism
  - Represent events as values instead of method calls
- Pattern matching:
  - Handle catch-all/default cases all in one place
  - Compiler fails to compile if all cases are not covered

SUMMARY

- Look to the function
- Use partial application to load functions with state
- Use discriminated unions for polymorphic values and represent events as values
- Use pattern matching to employ the compiler as guarantee that all cases are covered

OTHER EXAMPLES

- Visitor
- Composite
- Adapter
- Null Object



The Pragmatic Programmers

Functional Programming Patterns

in Scala and Clojure

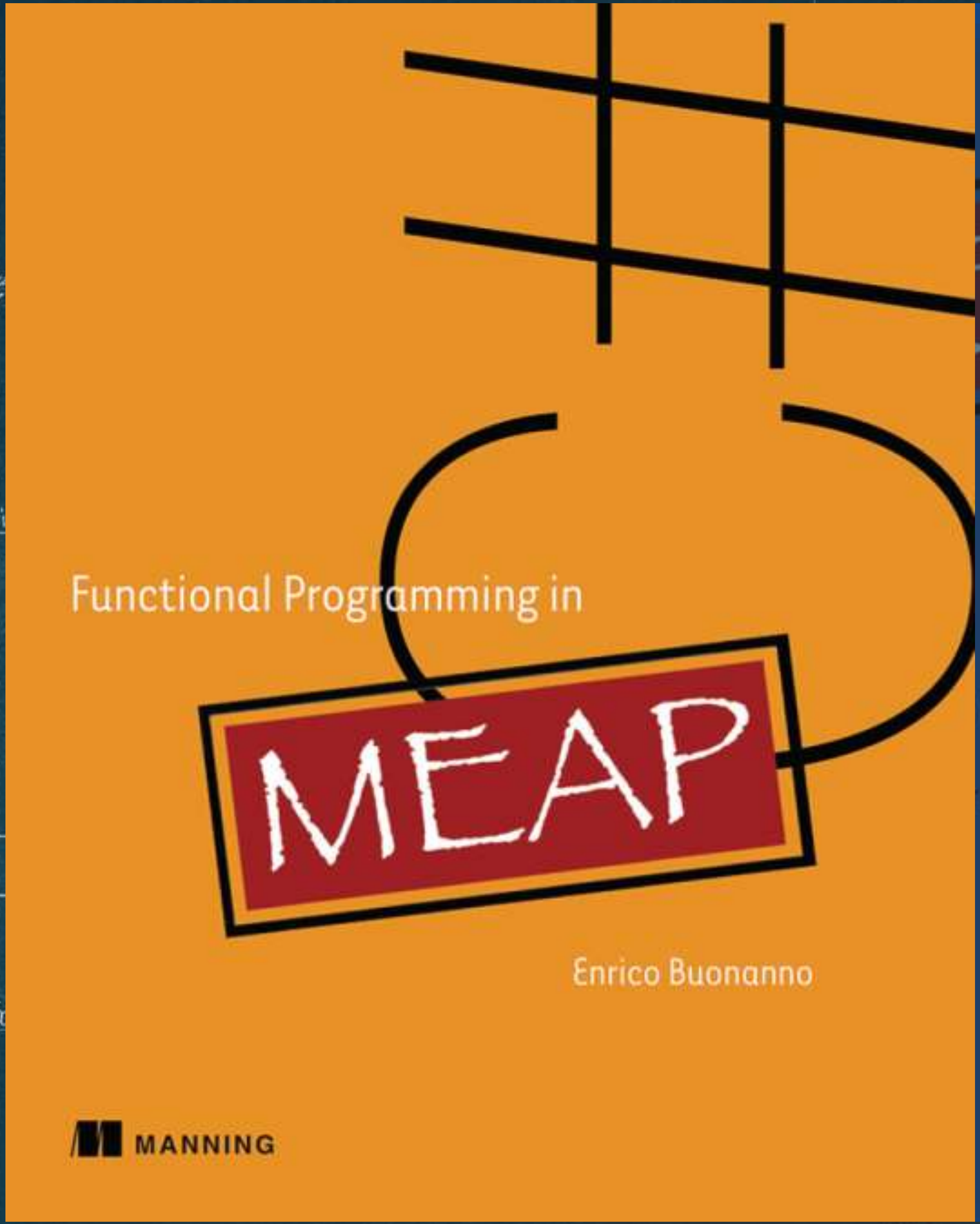
Write Lean Programs for the JVM

Michael Bevilacqua-Linn

Edited by John Osborn and Fahmida Y. Rashid

- Additional examples
- Gentle introduction to Scala & Clojure

Can’t do F# at work?



C#

Functional Programming in

MEAP

Enrico Buonanno

MANNING

EAP

Java

in

ograms using functional techniques

Pierre-Yves Saumont

Atencio

JavaScript

nming in

t programs using functional techniques

```
int(*func)(int, int)

int (Class::*method)(int, int)
```

```
int(*func)(int, int)

int (Class::*method)(int, int)

Func<int, int, int> func

function func(x,y) { /*...*/ }
```

DevSpace would like to thank our sponsors!

dealnews

cabinet
Document Management Solutions

novation systems

CURSE
Enjoy the Game

CODEROCKET

COLSA
Corporation

Dan Nagle

FOUNDRY
Huntsville's Software Startup Incubator

U.S. Space & Rocket Center

T&W Operations, Inc.
Smart. Solutions.

IMAGE CREDITS

- Joy Gas service station elevation drawings, 910 Lake Shore Road Blvd W., Toronto, Canada. https://commons.wikimedia.org/wiki/File:Joy_Oil_gas_station_blueprints.jpg
- Top Ten Seinfeld Locations in NYC. http://guestofaguest.com/wp-content/uploads/2010/07/g57zu7unymda43ucsrltelcto1_r1_500.jpg
- Your Options, According to Yoda. http://i.imgur.com/sxtu4l.png
- Amelia's Sad Face | Donnie Ray Jones | Flickr https://www.flickr.com/photos/donnieray/9436653177
- Club Jade • "But I was going into Tosche Station to pick up… http://clubjade.tumblr.com/post/74278187340/but-i-was-going-into-tosche-station-to-pick-up
- Monty Python and the Holy Grail images Ack! wallpaper and background photos http://images.fanpop.com/images/image_uploads/Ack--monty-python-and-the-holy-grail-591667_800_441.jpg