Advanced Defensive Programming Techniques

UNDERSTANDING LIMITATIONS OF TRADITIONAL DEFENSIVE CODE

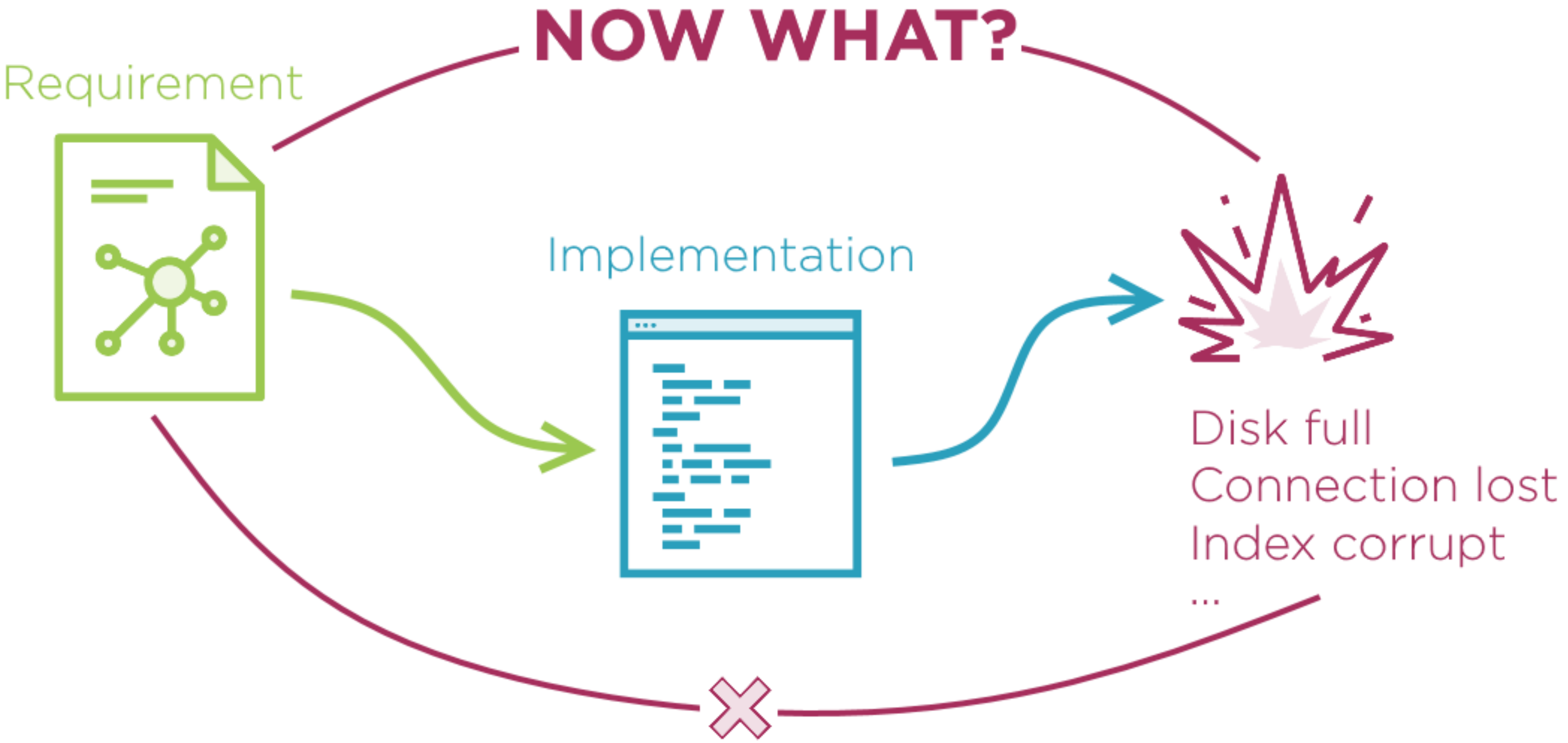
Zoran Horvat
PRINCIPAL CONSULTANT AT CODING HELMET

@zoranh75 csharpmentor.com

Preceding Courses

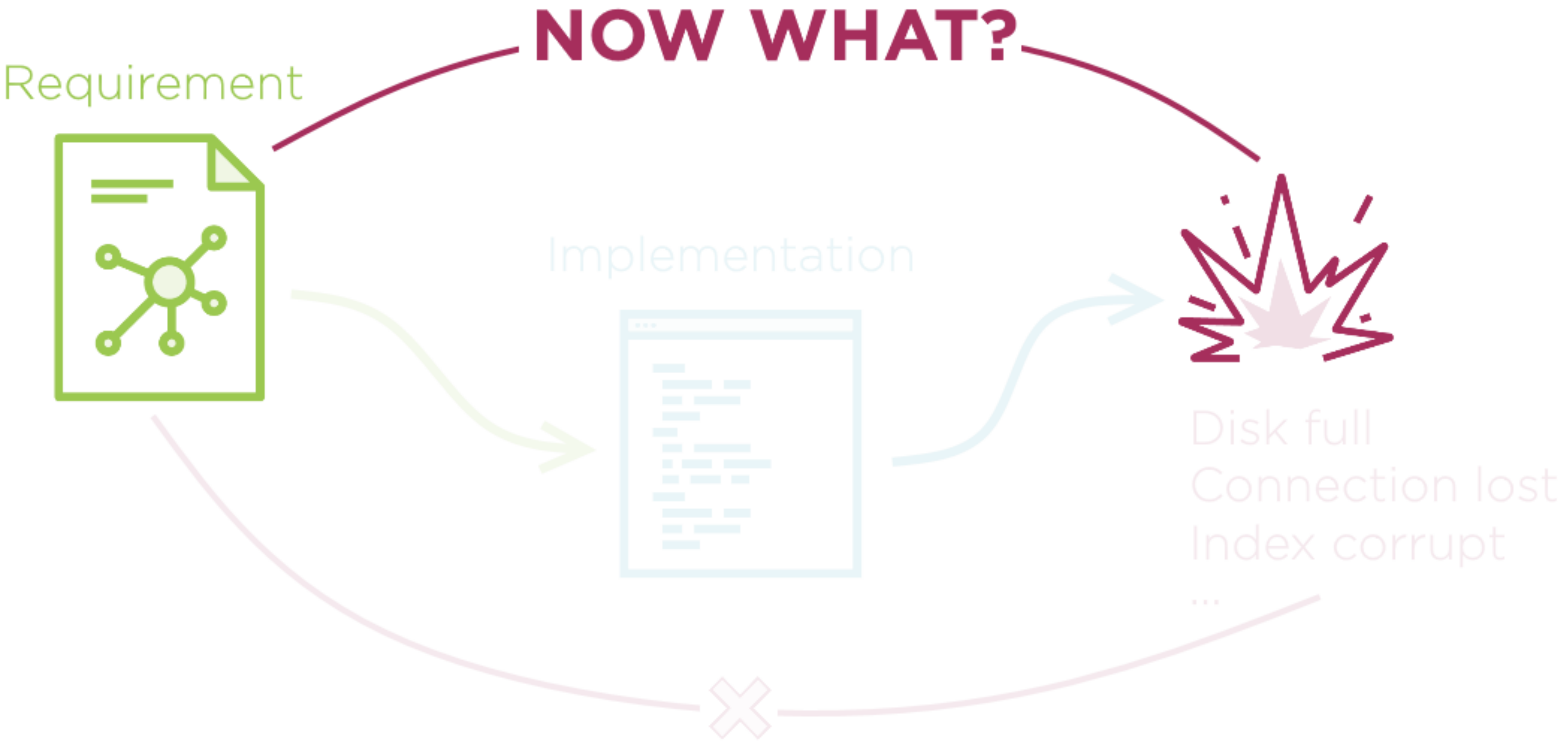
- Tactical Design Patterns in .NET: Managing Responsibilities
- Tactical Design Patterns in .NET: Control Flow
- Tactical Design Patterns in .NET: Creating Objects
- Making Your C# Code More Object-Oriented
- Writing Highly Maintainable Unit Tests
- Improving Testability Through Design

Defense – What exactly is it?



Defense – What exactly is it?

An idea:
Defend by design!



Defense – What exactly is it?

Defensive Coding	Defensive Programming
Write code which explicitly defends from negative cases	Produce design which defends out of the box
	Positive and negative execution scenarios treated the same
Always applicable *and produces poor results*	Sometimes applicable *and produces great results*

The First Law of Defensive Programming

WHEN YOU HAVE TO DEFEND,
YOU HAVE ALREADY LOST.

Build safety into high-level design.

Consistent objects

Consistent mutations

No primitive types

Function domains

Defensive design

Objects, not nulls

Rich domain models

No exceptions

What Follows in This Course

Create consistent
and complete objects

Consequence: No defense

Consistent objects

Consistent mutations

No primitive types

Function domains

Defensive design

Objects, not nulls

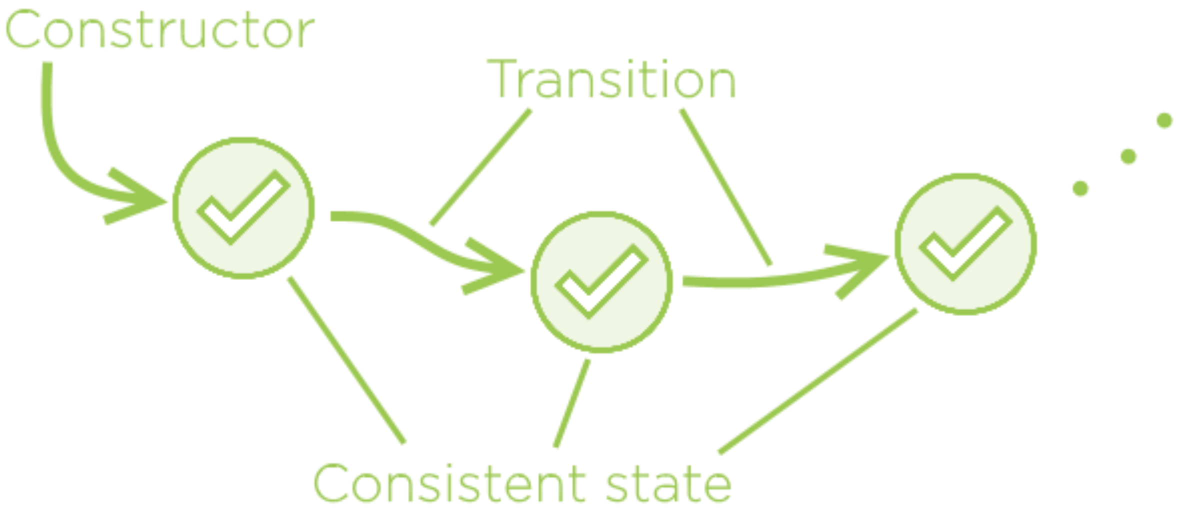
Rich domain models

No exceptions

What Follows in This Course

All transitions must lead to complete and consistent states

Consequence: No defense



Consistent objects

Consistent mutations

No primitive types

Function domains

Defensive design

Objects, not nulls

Rich domain models

No exceptions

What Follows in This Course

Avoid excessive use of primitive types

Replace primitive types with custom domain types

Consequence: No defense

Consistent objects

Consistent mutations

No primitive types

Function domains

Defensive design

Objects, not nulls

Rich domain models

No exceptions

What Follows in This Course

Satisfy mandatory conditions before making a function call

Consistent objects

Consistent mutations

No primitive types

Function domains

Defensive design

Objects, not nulls

Rich domain models

No exceptions

What Follows in This Course

Satisfy mandatory conditions before making a function call

```
try
{
  obj.DoSomething();
}
catch (InvalidOperationException)
{
  ...
}
```

Am I safe making this call?

Defensive code

Consistent objects

Consistent mutations

No primitive types

Function domains

Defensive design

Objects, not nulls

Rich domain models

No exceptions

What Follows in This Course

Satisfy mandatory conditions before making a function call

Defensive design starts after this point

Consistent objects

Consistent mutations

No primitive types

Function domains

Defensive design

Objects, not nulls

Rich domain models

No exceptions

What Follows in This Course

Encapsulate state and operations

Unify positive and negative execution flows

Consistent objects

Consistent mutations

No primitive types

Function domains

Defensive design

Objects, not nulls

Rich domain models

No exceptions

What Follows in This Course

Do not use null references

THERE ARE REASONS WHY NULL EXISTS, AND YOUR CODE IS NOT ONE OF THEM!

Without null: No defense

Consistent objects

Consistent mutations

No primitive types

Function domains

Defensive design

Objects, not nulls

Rich domain models

No exceptions

What Follows in This Course

Immutability:
Good servant, bad master.

Restricted mutability
for the best of both worlds

Historical modeling
for powerful features

Consistent objects

Consistent mutations

No primitive types

Function domains

Defensive design

Objects, not nulls

Rich domain models

No exceptions

What Follows in This Course

Do not introduce alternate execution paths with exceptions

Do not use exceptions as just another heavyweight `if-else`

Better approach:
Pass either the result or the error in a discriminated union

Consequence: No defense

Consistent objects

Consistent mutations

No primitive types

Function domains

Defensive design

Objects, not nulls

Rich domain models

No exceptions

What Follows in This Course

Defensive code is a matter of choice!

Cheap feature:	Defend in code
Expensive feature:	Defend in design

What Follows in This Course

Consistent objects

Consistent mutations

No primitive types

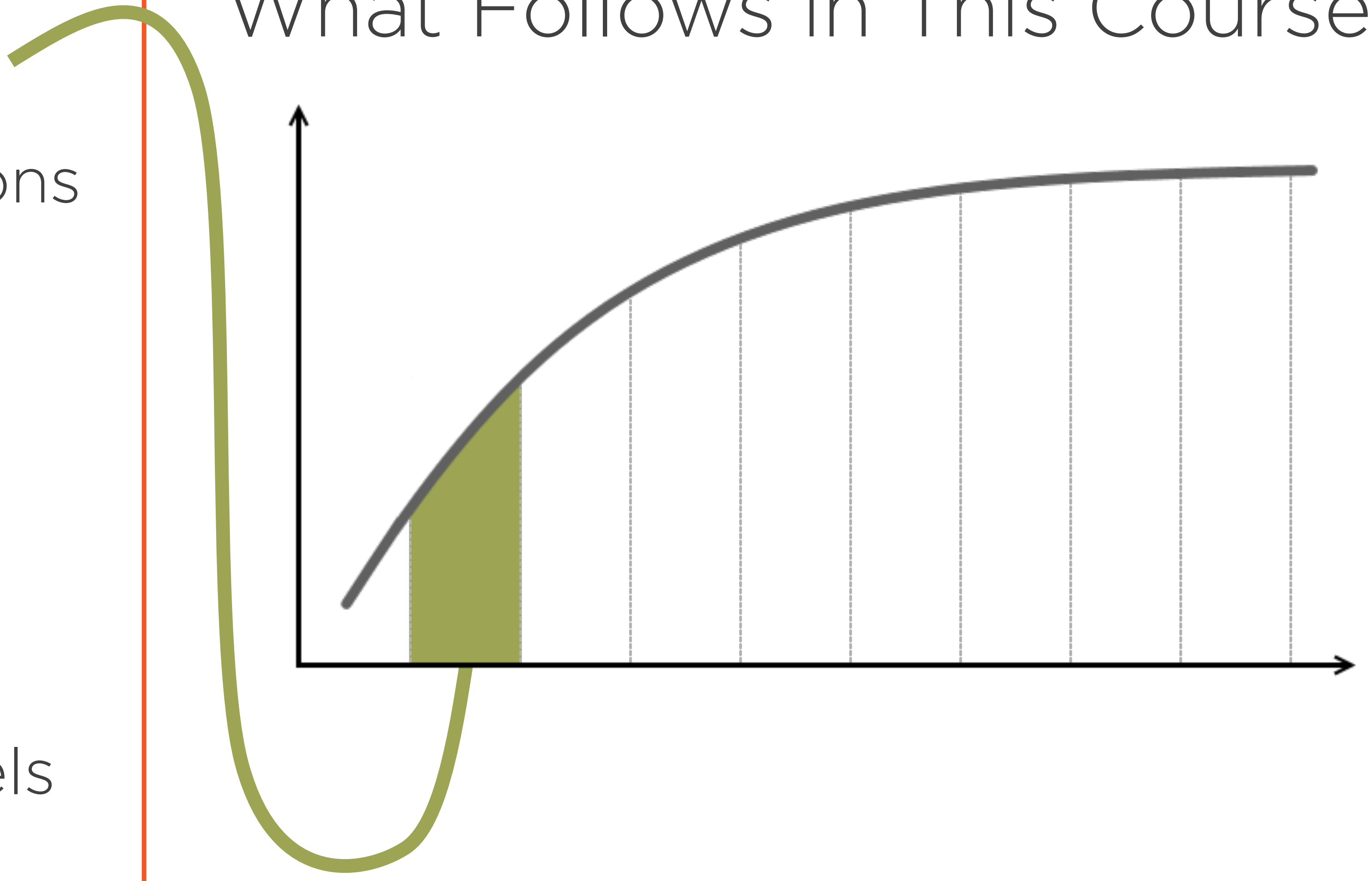
Function domains

Defensive design

Objects, not nulls

Rich domain models

No exceptions



What Follows in This Course

Consistent objects

Consistent mutations

No primitive types

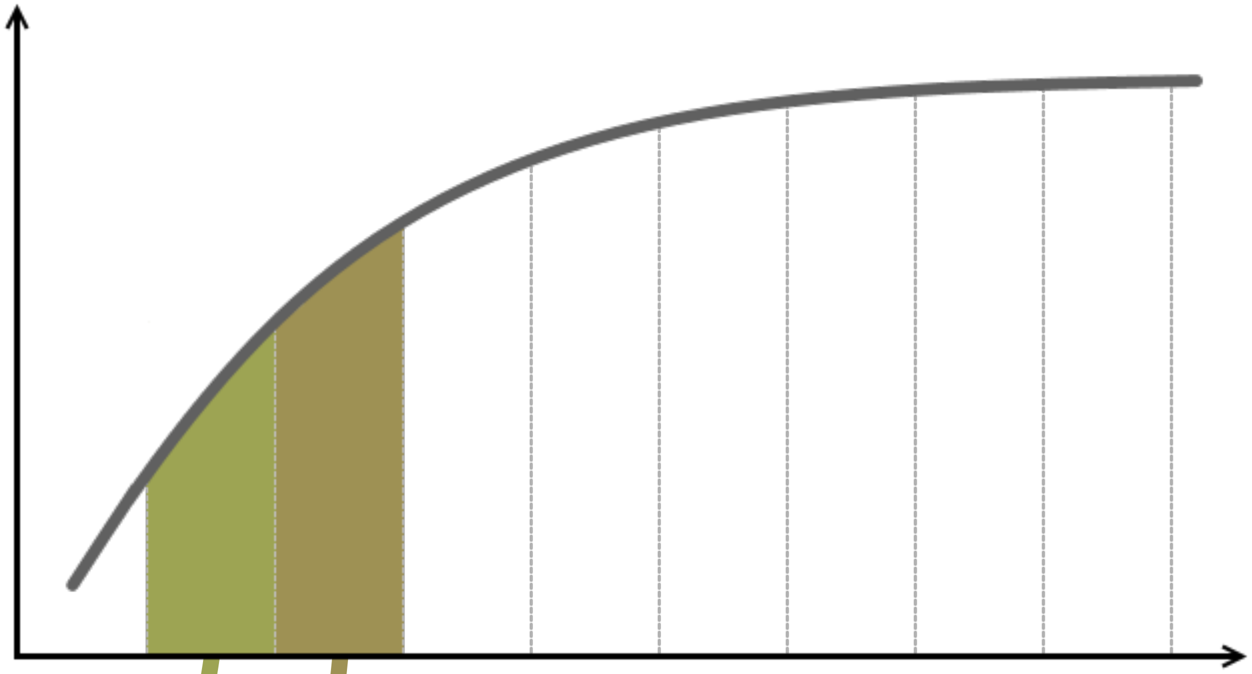
Function domains

Defensive design

Objects, not nulls

Rich domain models

No exceptions



What Follows in This Course

Consistent objects

Consistent mutations

No primitive types

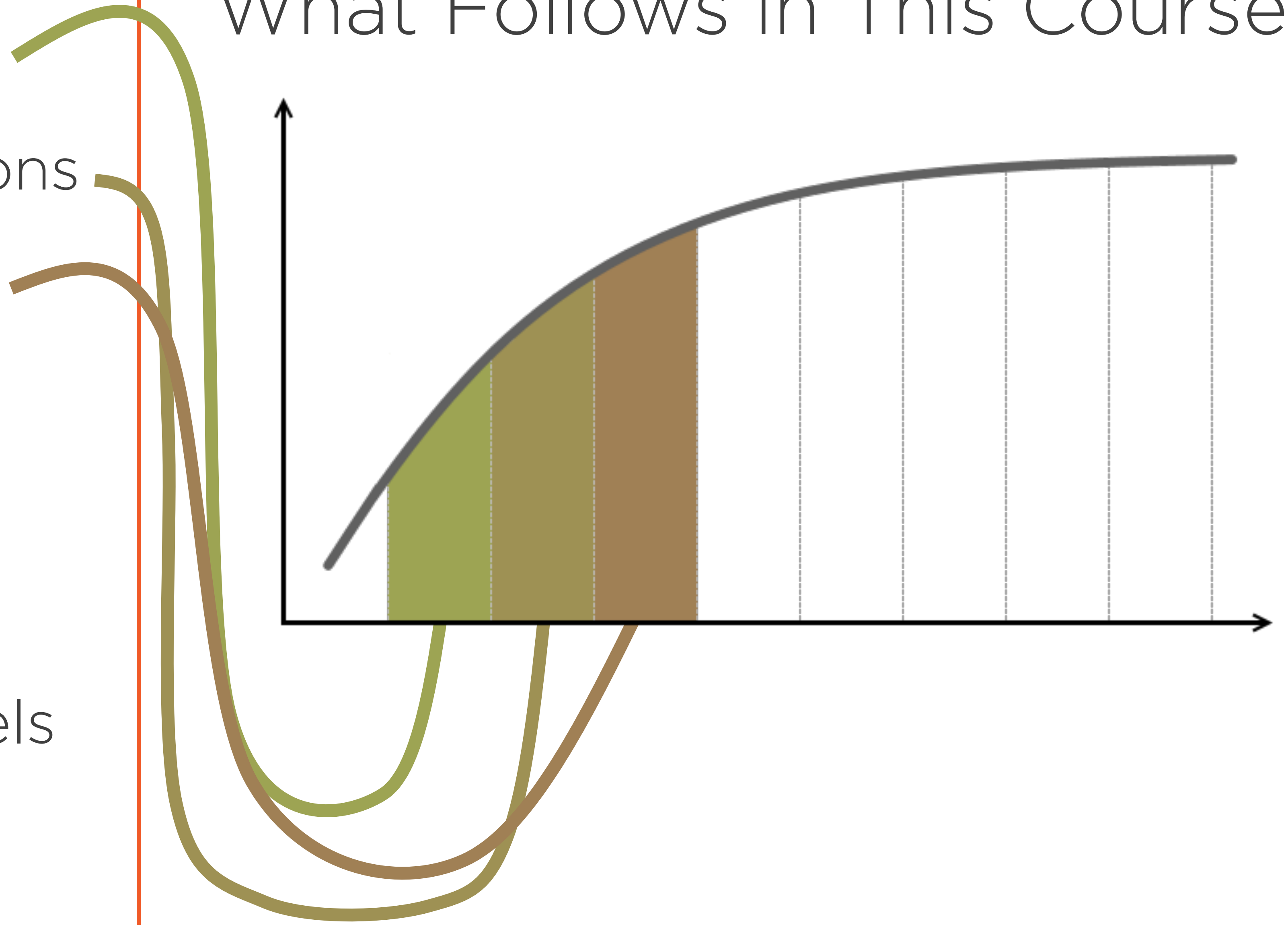
Function domains

Defensive design

Objects, not nulls

Rich domain models

No exceptions



What Follows in This Course

Consistent objects

Consistent mutations

No primitive types

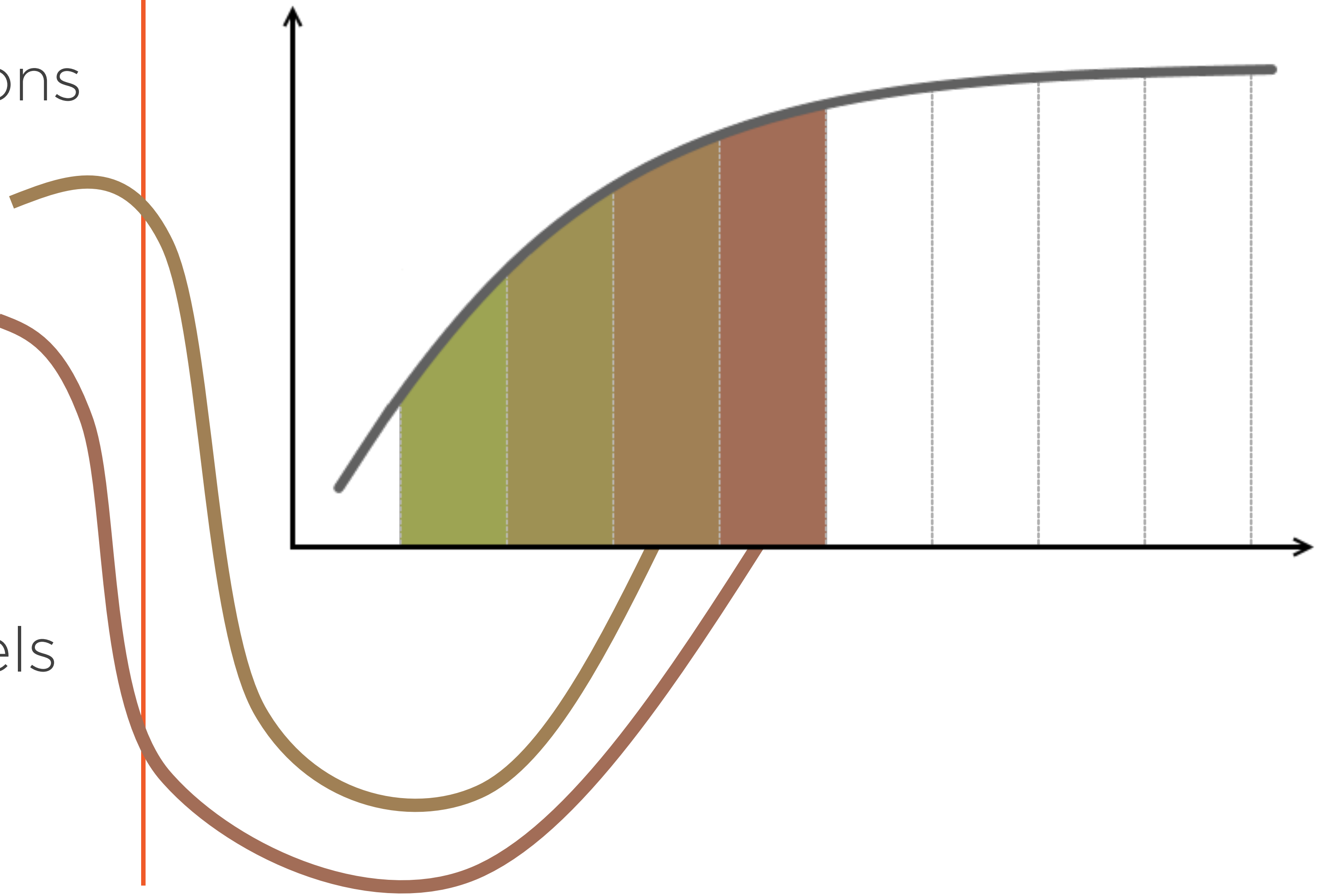
Function domains

Defensive design

Objects, not nulls

Rich domain models

No exceptions



What Follows in This Course

Consistent objects

Consistent mutations

No primitive types

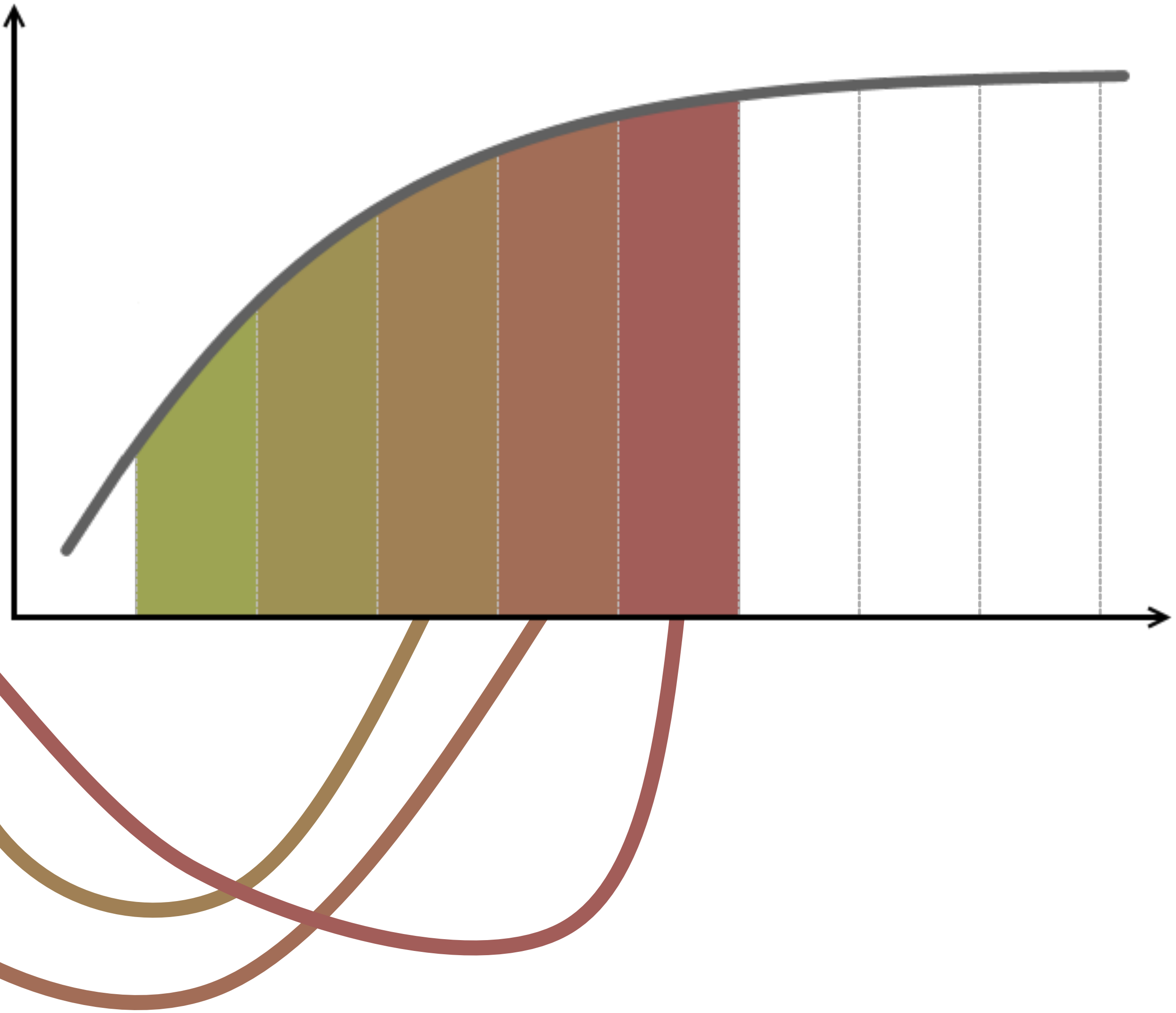
Function domains

Defensive design

Objects, not nulls

Rich domain models

No exceptions



Consistent objects

Consistent mutations

No primitive types

Function domains

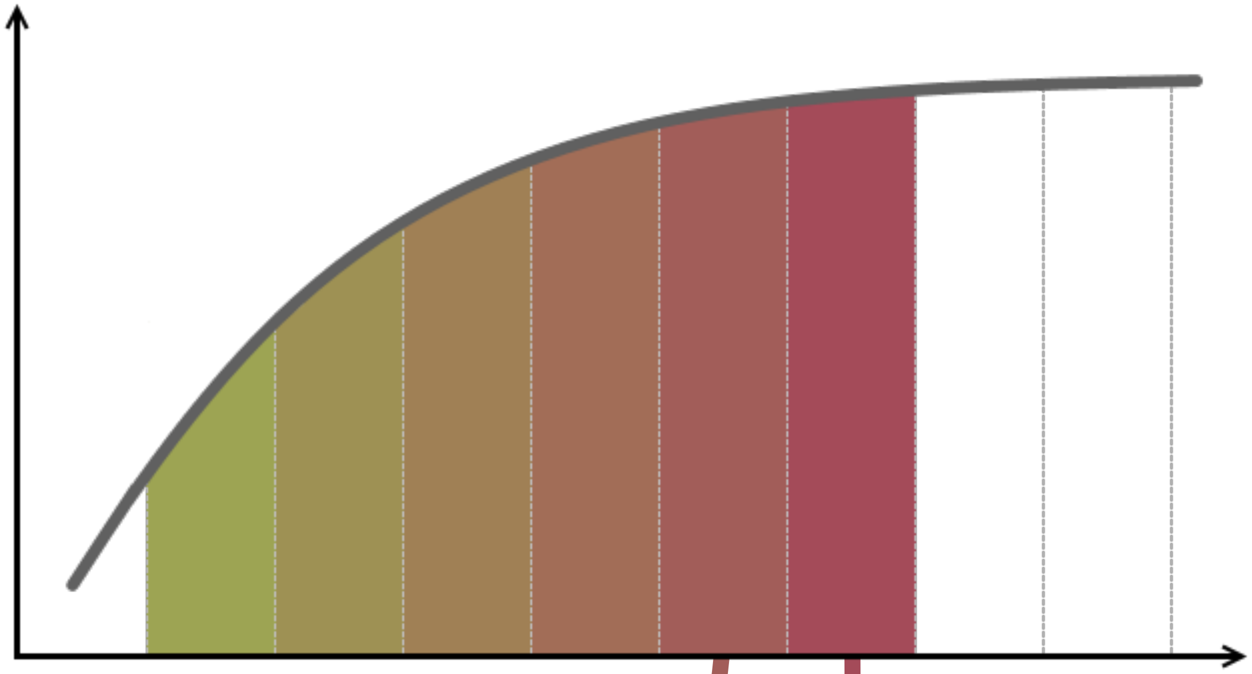
Defensive design

Objects, not nulls

Rich domain models

No exceptions

What Follows in This Course



What Follows in This Course

Consistent objects

Consistent mutations

No primitive types

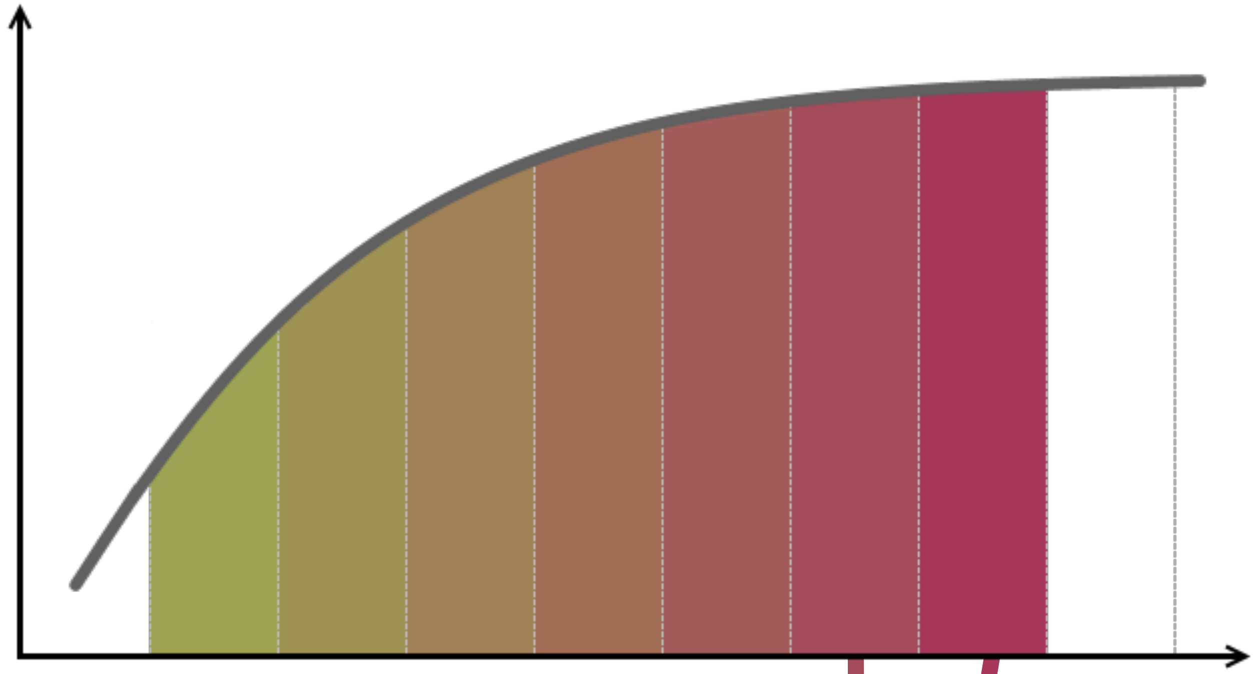
Function domains

Defensive design

Objects, not nulls

Rich domain models

No exceptions



What Follows in This Course

Consistent objects

Consistent mutations

No primitive types

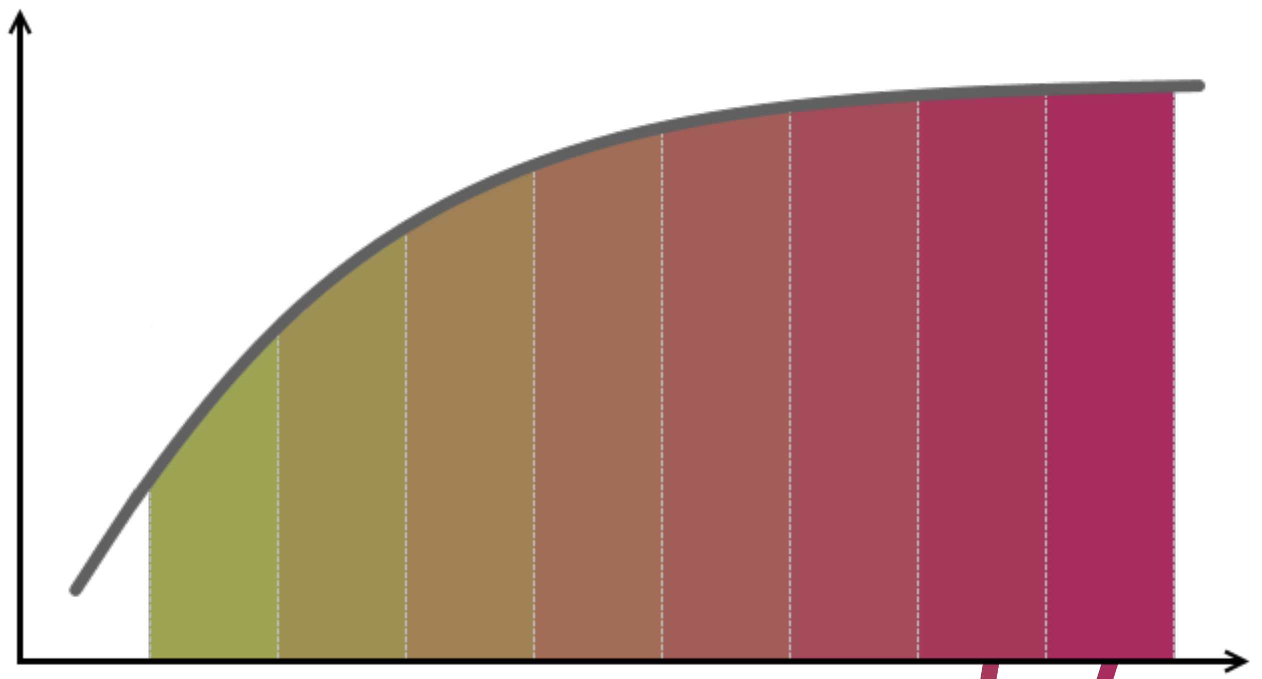
Function domains

Defensive design

Objects, not nulls

Rich domain models

No exceptions



Deborah Kurata

Defensive Coding in C#

Advanced Defensive Programming Techniques

This course presumes understanding of basic defensive coding techniques

This course covers more than just coding

- ✓ **Design**
- ✓ **Coding**
- ✓ **Analysis**

Deborah Kurata

Defensive Coding in C#

Start from defensive coding

Let the design principles grow naturally

Defensive design will emerge

Traditional Defensive Techniques

Predictability

Certainty

Simplicity

Completeness

Failing fast

Handling errors

Reveal intention

- ✓ Method name
- ✓ Argument names
- ✓ Return value

Example:

```
public MyType MyMethod();
```

- ☑ Return an object of `MyType`
- ☒ Do not return null
- ☒ Do not throw an exception

Predictability

Certainty

Simplicity

Completeness

Failing fast

Handling errors

Traditional Defensive Techniques

Do not pass switches to a method

- ☒ No Boolean arguments
- ☒ No enum arguments

Switches introduce uncertainty

Method should do one thing with absolute certainty

Only accept valid argument values

Predictability

Certainty

Simplicity

Completeness

Failing fast

Handling errors

Traditional Defensive Techniques

Keep small number of method arguments

Keep the code short

Avoid branching when possible

Avoid code repetition

Predictability

Certainty

Simplicity

Completeness

Failing fast

Handling errors

Traditional Defensive Techniques

`if` instruction should have a reasonable **`else`** branch

`switch` instruction should have a reasonable **`default`** case

Predictability

Certainty

Simplicity

Completeness

Failing fast

Handling errors

Traditional Defensive Techniques

Do not let bad data propagate

- ☑ Fail right away, or
- ☑ Provide a reasonable default and keep going

Predictability

Certainty

Simplicity

Completeness

Failing fast

Handling errors

Traditional Defensive Techniques

Handle errors when they happen

Assert preconditions, postconditions and invariants

IF SOMETHING **CANNOT** HAPPEN, ASSERT IT AND IT WILL **NOT** HAPPEN

Add global exception handler to keep the application running

Predictability

Certainty

Simplicity

Completeness

Failing fast

Handling errors

These were the traditional *defensive coding* techniques

We can start discovering *defensive designing* techniques

An Experiment

```
int[] data
```

Defensive Code

```
if (data.Any())
{
  int min = data.Min();
  ReportSuccess(min);
}
else
{
  ReportFailure();
}
```

Regular flow

Defensive part

An idea:

Merge `if` and `else` into one flow

Defensive Design

Sequential Search Improvement

Sequential Search

```
bool Contains(int[] ar, int search)
{
  for (int i = 0; i < ar.Length; i++)
  {
    if (ar[i] == search)
    {
        return true;
    }
  }
  return false;
}
```

Two branching instructions

Quick Sequential Search

Sequential Search Improvement

Sequential Search

```
bool Contains(int[] ar, int search)
{
  for (int i = 0; i < ar.Length; i++)
  {
    if (ar[i] == search)
    {
        return true;
    }
  }
  return false;
}
```

Quick Sequential Search

```
bool Contains(int[] ar, int search)
{
  int end = ar.Length - 1;
  int last = ar[end];
  ar[end] = search;

  int index = 0;
  while (ar[index] != search)
  {
    index = index + 1;
  }

  ar[end] = last;
  return index < end || last == search;
}
```

Put the last element aside

Sequential Search Improvement

Sequential Search

```
bool Contains(int[] ar, int search)
{
  for (int i = 0; i < ar.Length; i++)
  {
    if (ar[i] == search)
    {
        return true;
    }
  }
  return false;
}
```

Put the search value at the end of the array

Quick Sequential Search

```
bool Contains(int[] ar, int search)
{
  int end = ar.Length - 1;
  int last = ar[end];
  ar[end] = search;

  int index = 0;
  while (ar[index] != search)
  {
    index = index + 1;
  }

  ar[end] = last;
  return index < end || last == search;
}
```

Put the last element aside

Sequential Search Improvement

Sequential Search

```
bool Contains(int[] ar, int search)
{
  for (int i = 0; i < ar.Length; i++)
  {
    if (ar[i] == search)
    {
        return true;
    }
  }
  return false;
}
```

Search the array

Quick Sequential Search

```
bool Contains(int[] ar, int search)
{
  int end = ar.Length - 1;
  int last = ar[end];
  ar[end] = search;

  int index = 0;
  while (ar[index] != search)
  {
    index = index + 1;
  }

  ar[end] = last;
  return index < end || last == search;
}
```

Use only one branching instruction

Sequential Search Improvement

Sequential Search

```
bool Contains(int[] ar, int search)
{
  for (int i = 0; i < ar.Length; i++)
  {
    if (ar[i] == search)
    {
        return true;
    }
  }
  return false;
}
```

Reconstruct the array

Was it “our” or “their” search value?

Quick Sequential Search

```
bool Contains(int[] ar, int search)
{
  int end = ar.Length – 1;
  int last = ar[end];
  ar[end] = search;

  int index = 0;
  while (ar[index] != search)
  {
    index = index + 1;
  }

  ar[end] = last;
  return index < end || last == search;
}
```

An Experiment

Defensive Code

```
if (data.Any())
{
  int min = data.Min();
  ReportSuccess(min);
}
else
{
  ReportFailure();
}
```

Call one function or the other

Defensive Design

```
int potentialMin =
  data.DefaultIfEmpty(0).Min();

Action todo =
  data.Take(1).Select(_ =>
    () => ReportSuccess(potentialMin))
```

Take at most one element...

and map it into a lambda...

which calls the **ReportSuccess** custom function

An Experiment

Defensive Code

```
if (data.Any())
{
  int min = data.Min();
  ReportSuccess(min);
}
else
{
  ReportFailure();
}
```

Call one function or the other

Defensive Design

```
int potentialMin =
  data.DefaultIfEmpty(0).Min();

Action todo =
  data.Take(1).Select(_ =>
    () => ReportSuccess(potentialMin))
  .DefaultIfEmpty(ReportFailure)
  .Single();

todo();
```

An Experiment

Defensive Code

```
if (data.Any())
{
  int min = data.Min();
  ReportSuccess(min);
}
else
{
  ReportFailure();
}
```

Defensive call:

Acknowledges that there might be no data

Defensive Design

```
int potentialMin =
  data.DefaultIfEmpty(0).Min();

Action todo =
  data.Take(1).Select(_ =>
    () => ReportSuccess(potentialMin))
  .DefaultIfEmpty(ReportFailure)
  .Single();

todo();
```

Defensive call:

Acknowledges that there might be no action

An Experiment

Defensive Code

```
if (data.Any())
{
  int min = data.Min();
  ReportSuccess(min);
}
else
{
  ReportFailure();
}
```

Compiles fine without this

Defensive Design

```
int potentialMin =
  data.DefaultIfEmpty(0).Min();

Action todo =
  data.Take(1).Select(_ =>
    () => ReportSuccess(potentialMin))
  .DefaultIfEmpty(ReportFailure)
  .Single();

todo();
```

Will not compile without this

An Experiment

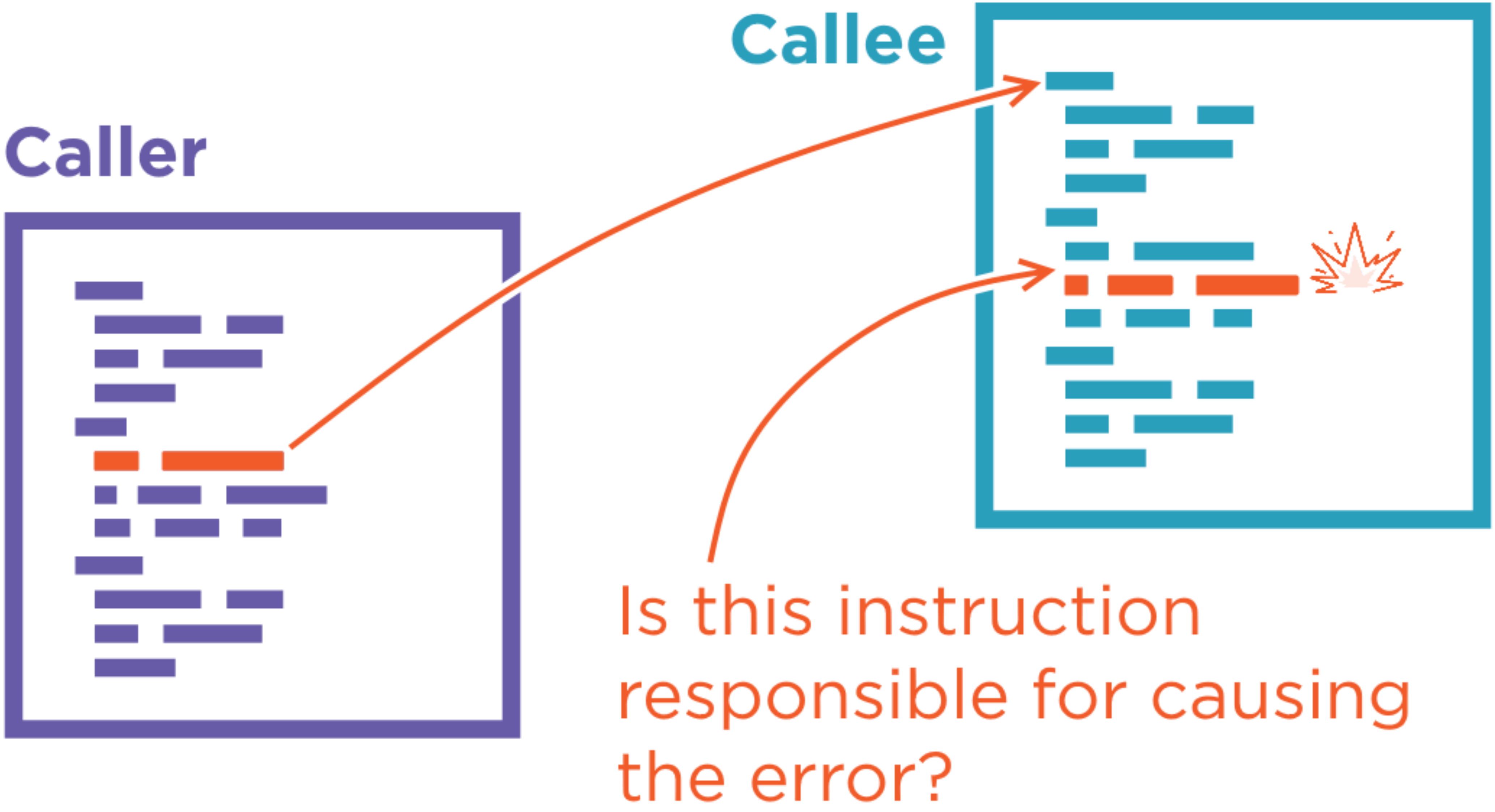
Defensive Code

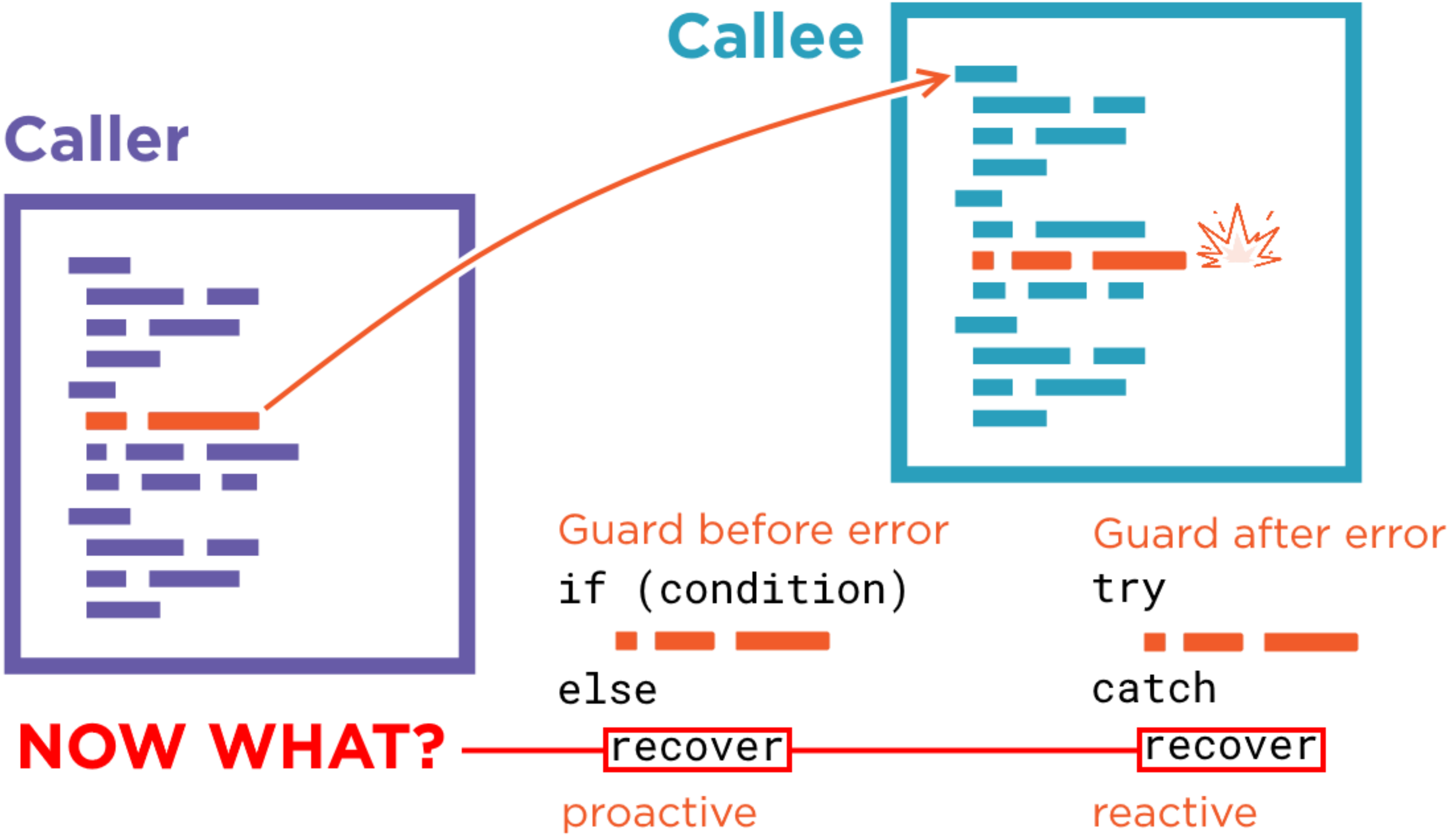
```
if (data.Any())
{
  int min = data.Min();
  ReportSuccess(min);
}
else
{
  ReportFailure();
}
```

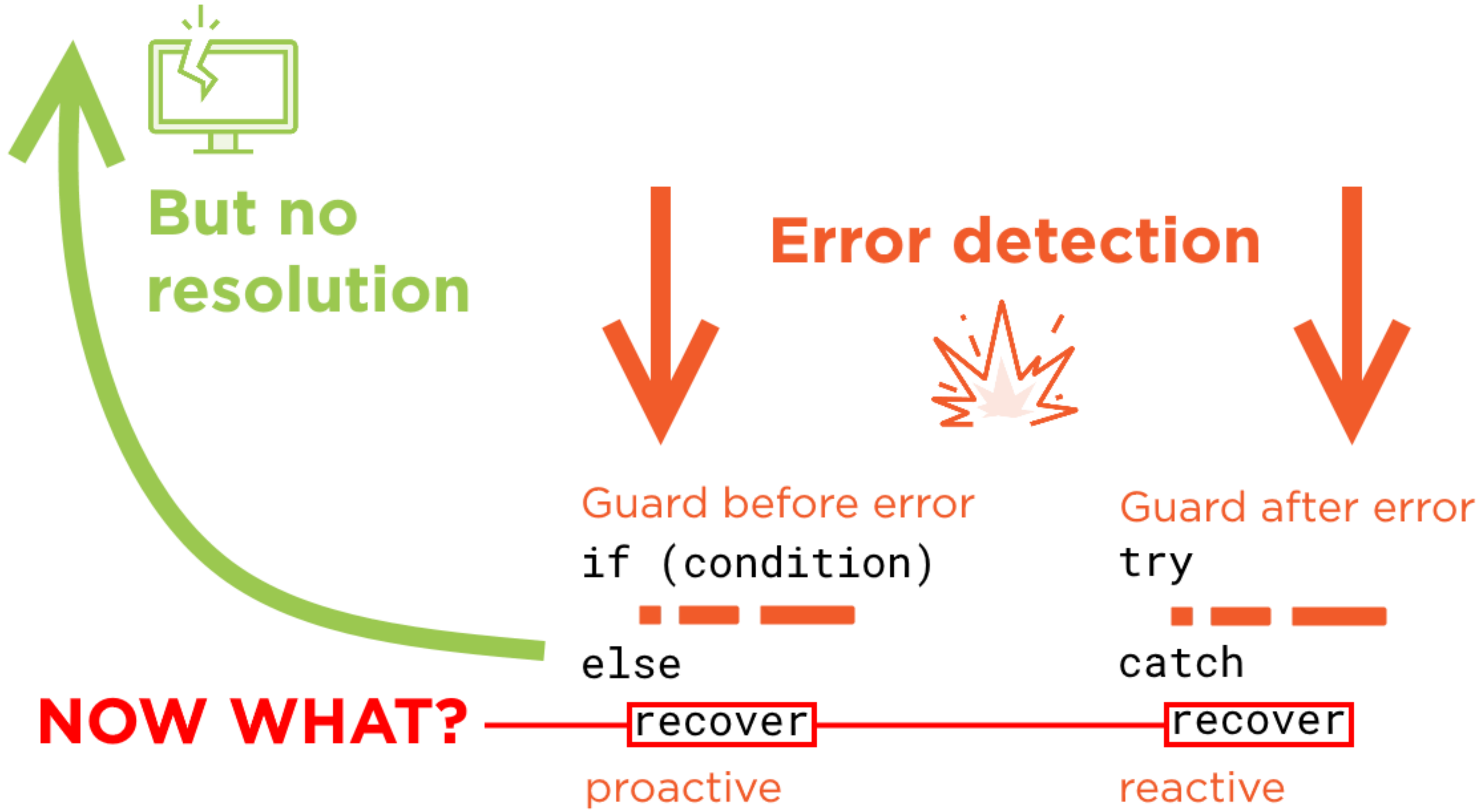
Defensive Design

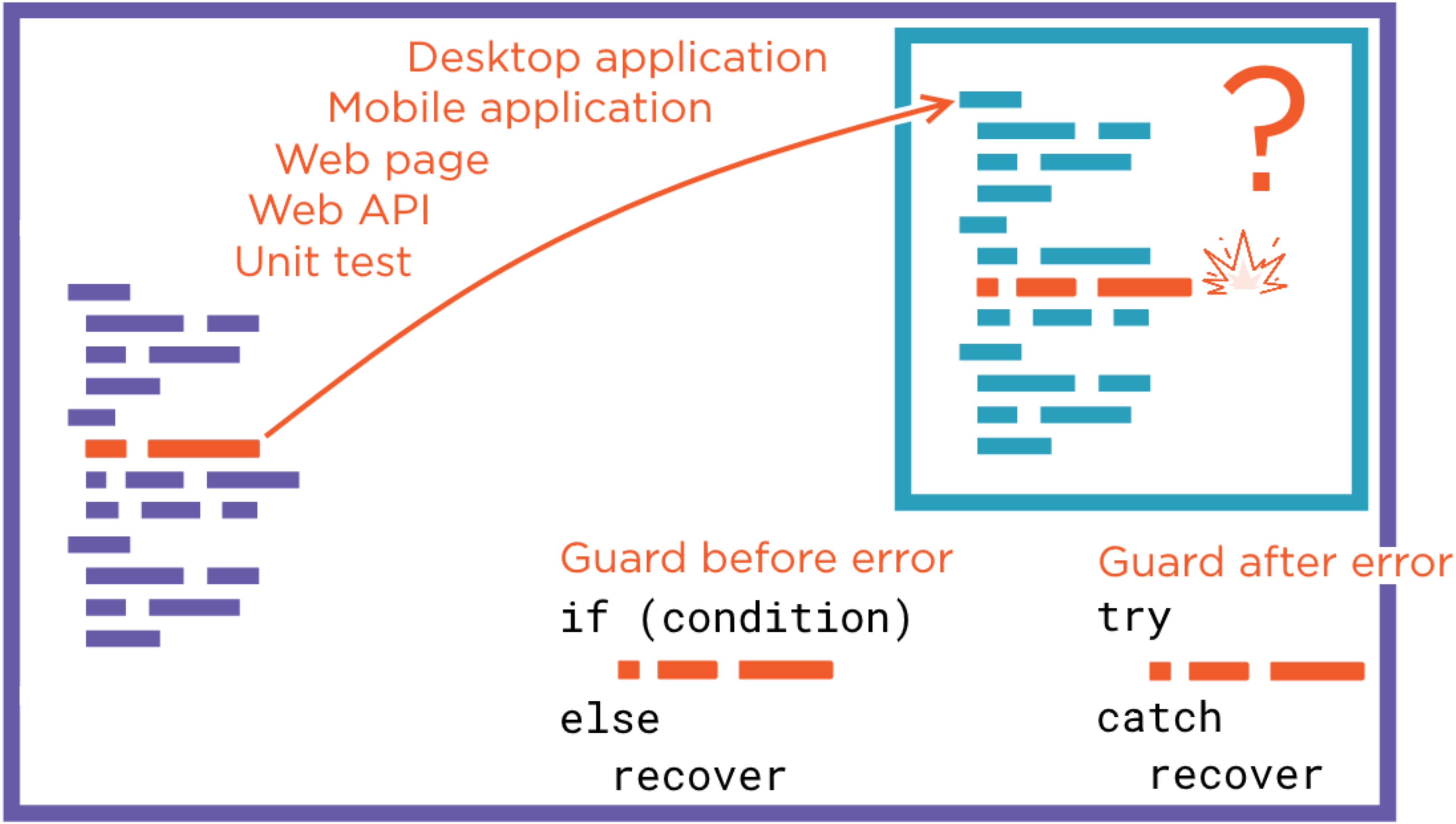
```
Action todo =
  data.Take(1).Select(_ =>
    () => ReportSuccess(data.Min()))
  .DefaultIfEmpty(ReportFailure)
  .Single();

todo();
```


Callee
Caller
Is this instruction responsible for causing the error?


Caller
Callee
Guard before error
if (condition)
else
recover
proactive
Guard after error
try
catch
recover
reactive
NOW WHAT?

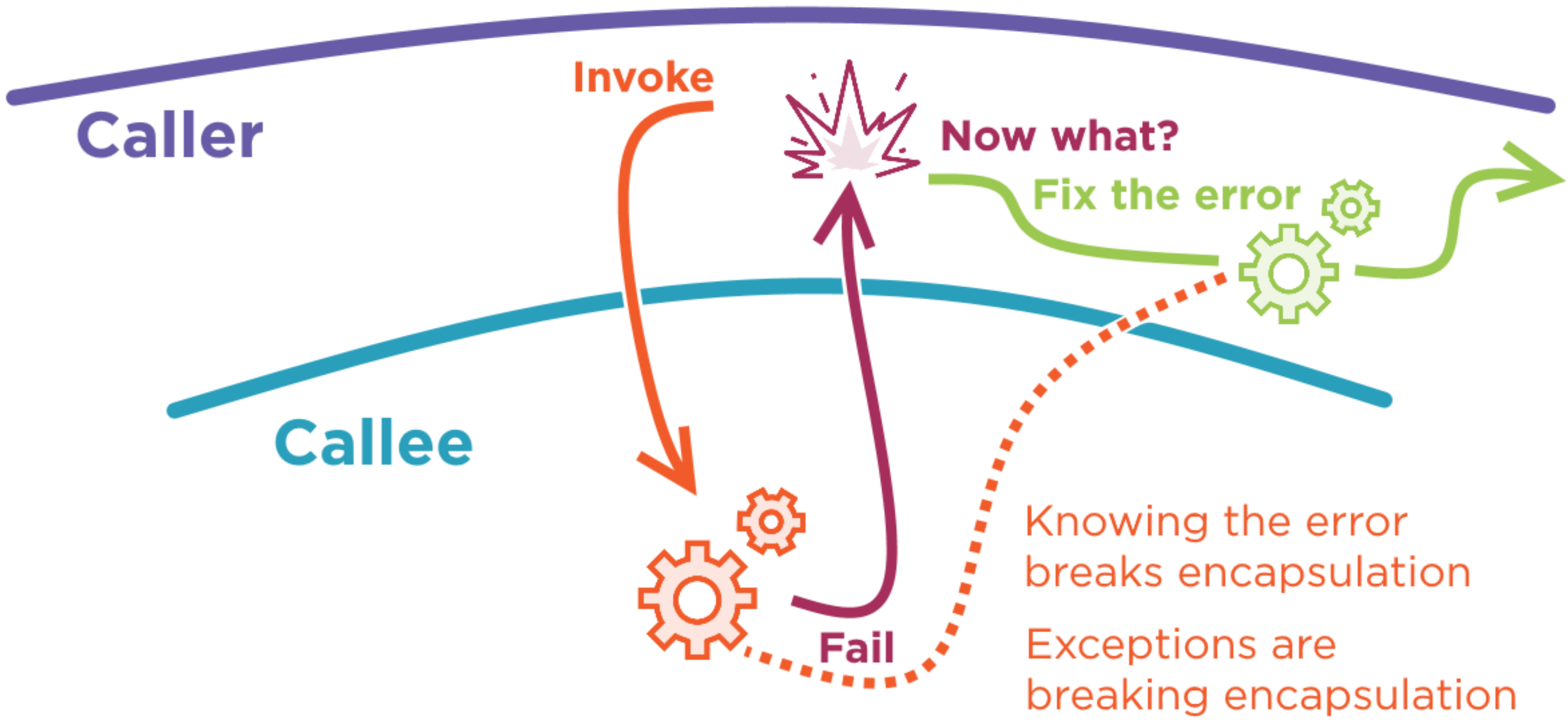

But no resolution
Error detection
Guard before error
if (condition)
else
recover
proactive
Guard after error
try
catch
recover
reactive
NOW WHAT?

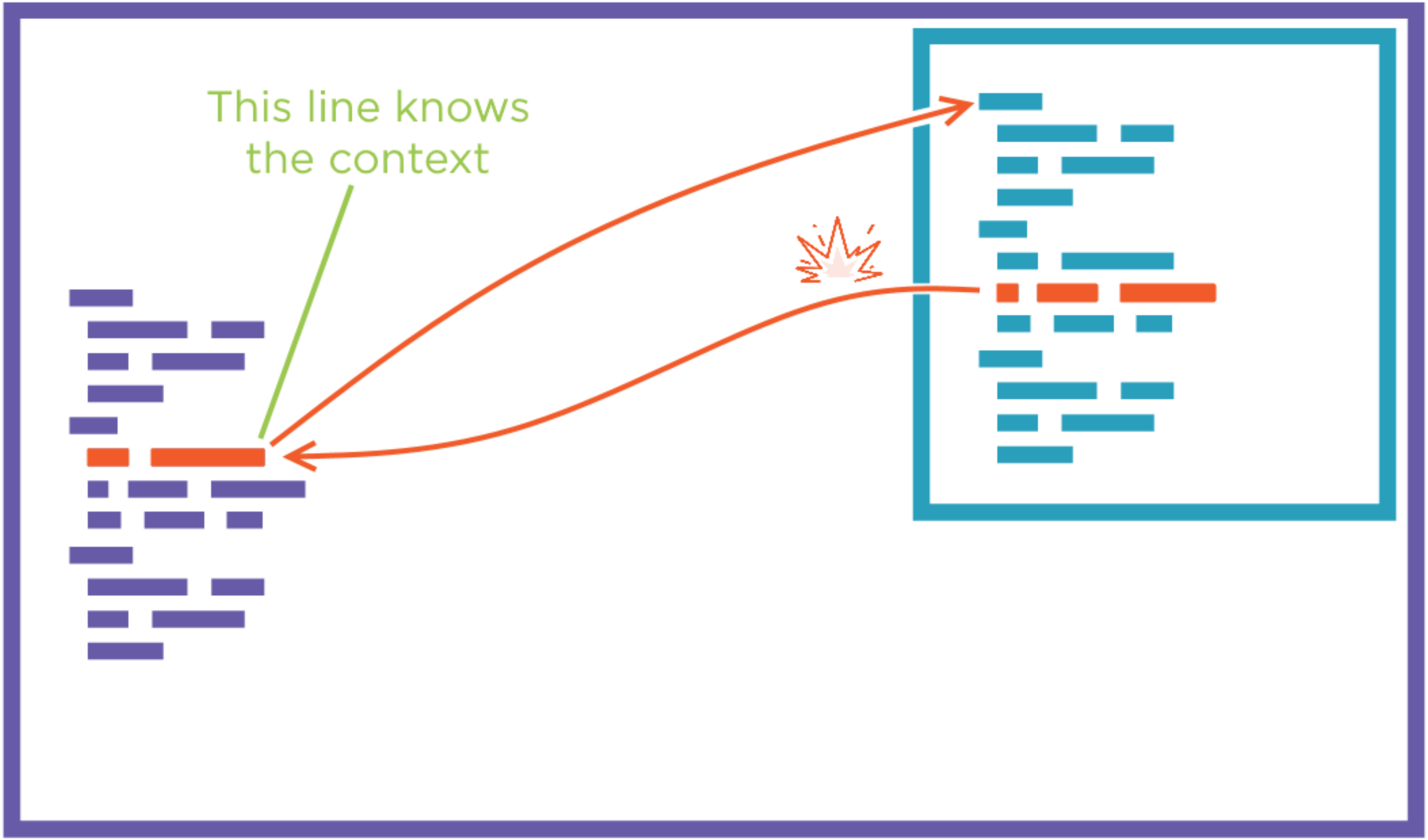


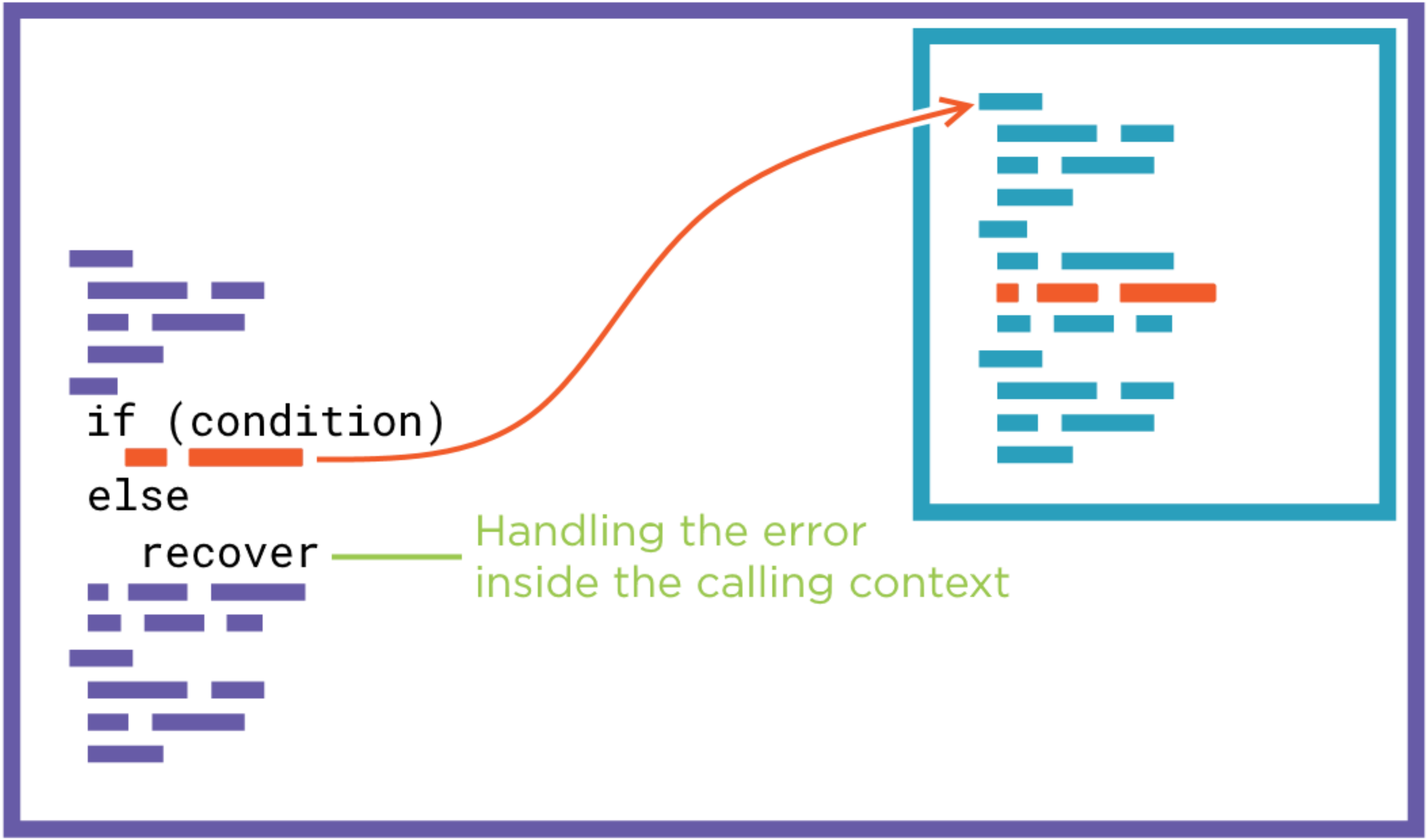
Desktop application
Mobile application
Web page
Web API
Unit test
?
Guard before error
if (condition)
else
recover
Guard after error
try
catch
recover

Called function cannot defend.

Called function can detect an error, but it cannot handle the error


Caller
Invoke
Now what?
Fix the error
Callee
Fail
Knowing the error breaks encapsulation
Exceptions are breaking encapsulation


This line knows
the context


if (condition)
else
recover
Handling the error
inside the calling context

Method Argument Validation

Method Caller

```
if (valid(x)) // proactive caller
{
  obj.SomeMethod(x);
}

try            // reactive caller
{
  obj.SomeMethod(x);
}
catch
{
  BeRobust();
}
KeepGoing();
...
```

Called Method

```
void SomeMethod(arg)
{
  if (!valid(arg))
    throw
  ...
}
```

Want to avoid error conditions?

Produce design in which they *cannot* appear!

```
class MoneyAccount
{
  void Deposit(decimal amount)
  {
    if (amount <= 0)
    {
      throw new
        ArgumentException();
    }
    // put amount onto a pile
  }
}
```

- **Amount must be positive**
- **Guard against invalid input**
  Invariant logic can be hard-coded
- **Throwing an exception may lead to instability**
  Called method doesn’t know if anyone handles this exception

```
class PositiveAmount
{
}

class MoneyAccount
{
  void Deposit(
    PositiveAmount amt)
  {
    if (amount <= 0)
    {
      throw new
        ArgumentException();
    }
    // put amount onto a pile
  }
}
```

- **Recognize positive amount as an explicit concept**
- **Request positive amount object**
- **No reason to check anymore**
  We know the amount is positive

```
class PositiveAmount
{
}

class MoneyAccount
{
  void Deposit(
    PositiveAmount amt)
  {
    // put amount onto a pile
  }
}
```

- **Recognize positive amount as an explicit concept**
- **Request positive amount object**
- **No reason to check anymore**
  We know the amount is positive

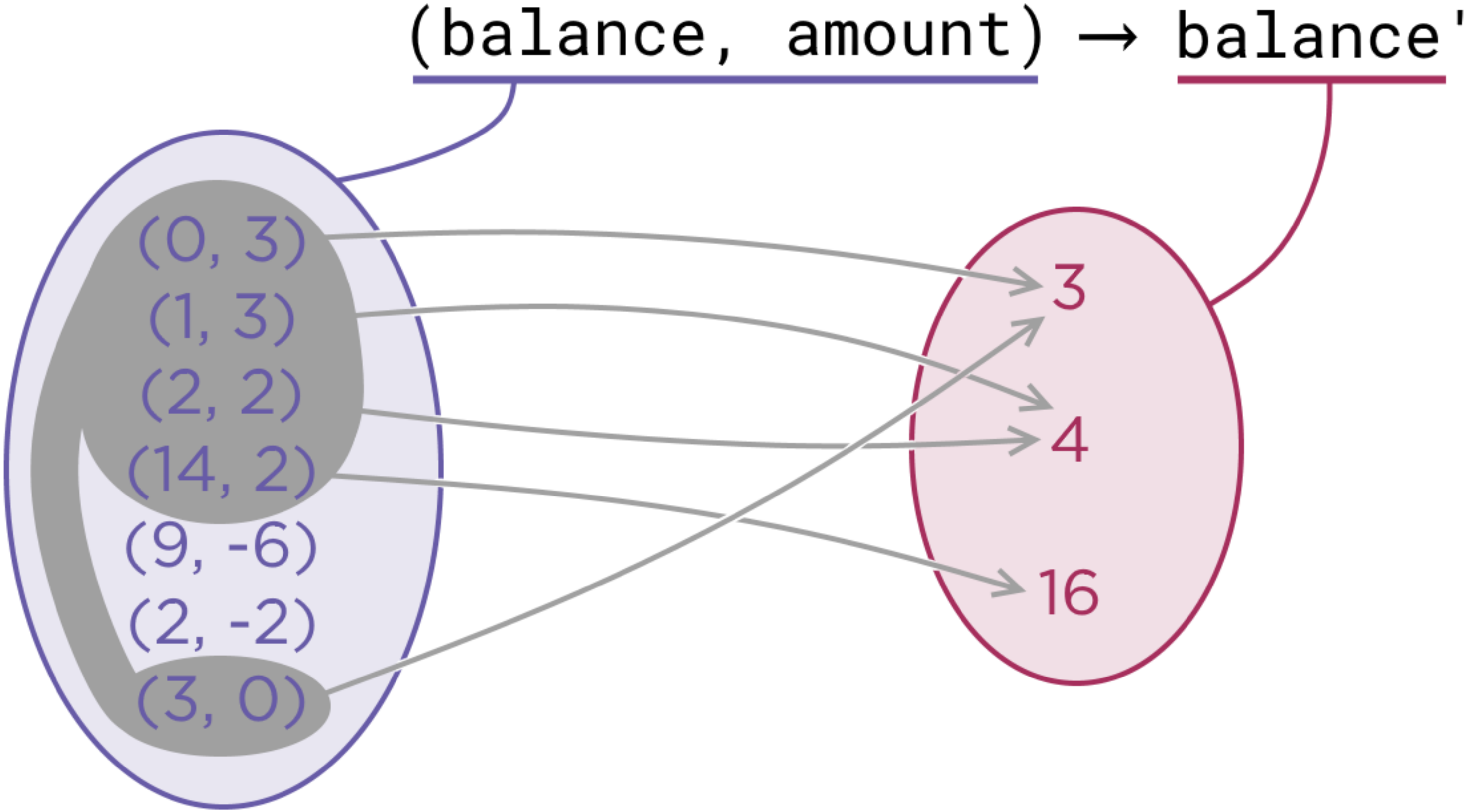
```
this.Deposit(decimal amount) → this'
```

`(this, amount) → this'`

Definition of a Program

Software which repeatedly transforms one system state into the next system state.

(a.k.a. The Depressing Definition of a Program)


(balance, amount) → balance'
(0, 3)
(1, 3)
(2, 2)
(14, 2)
(9, -6)
(2, -2)
(3, 0)
3
4
16

Summary

Traditional defensive techniques

- Based on if-else, try-catch, etc.

Other defensive techniques

- Known function arguments
- Predictable function returns
- Preconditions, postconditions, invariants

Defensive design

- Defense is built into public interface
- Unsafe use is syntactically incorrect

Ultimate goal: Nothing to defend from

Summary

The First Law of Defense

When you have to defend,
you have already lost

Summary

Explicit defense in code

- Detect the error and fail

Defense by design

- Steer the operation so that it never gets stuck

Next module

Creating Consistent Objects