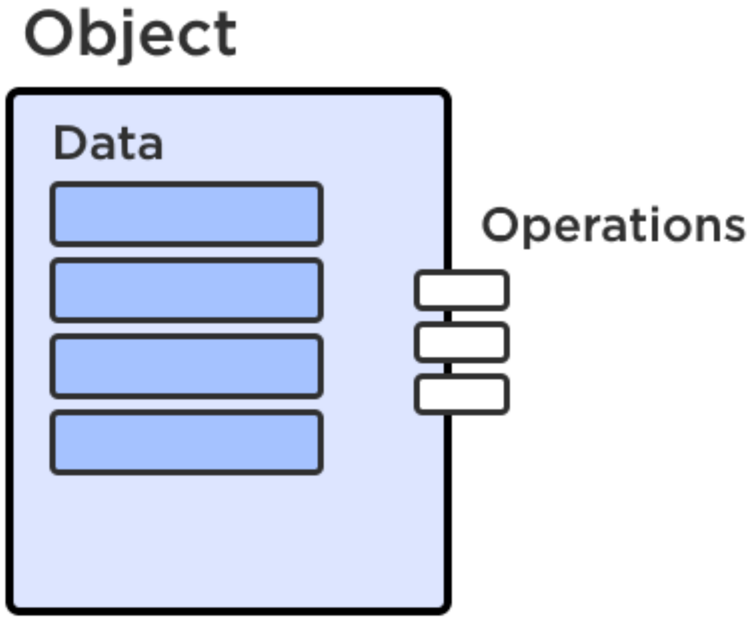
Removing Corruption by Only Creating Consistent Objects

Zoran Horvat
PRINCIPAL CONSULTANT AT CODING HELMET

@zoranh75 csharpmentor.com



Correctness

Stability

Consistency

Error conditions

Defense

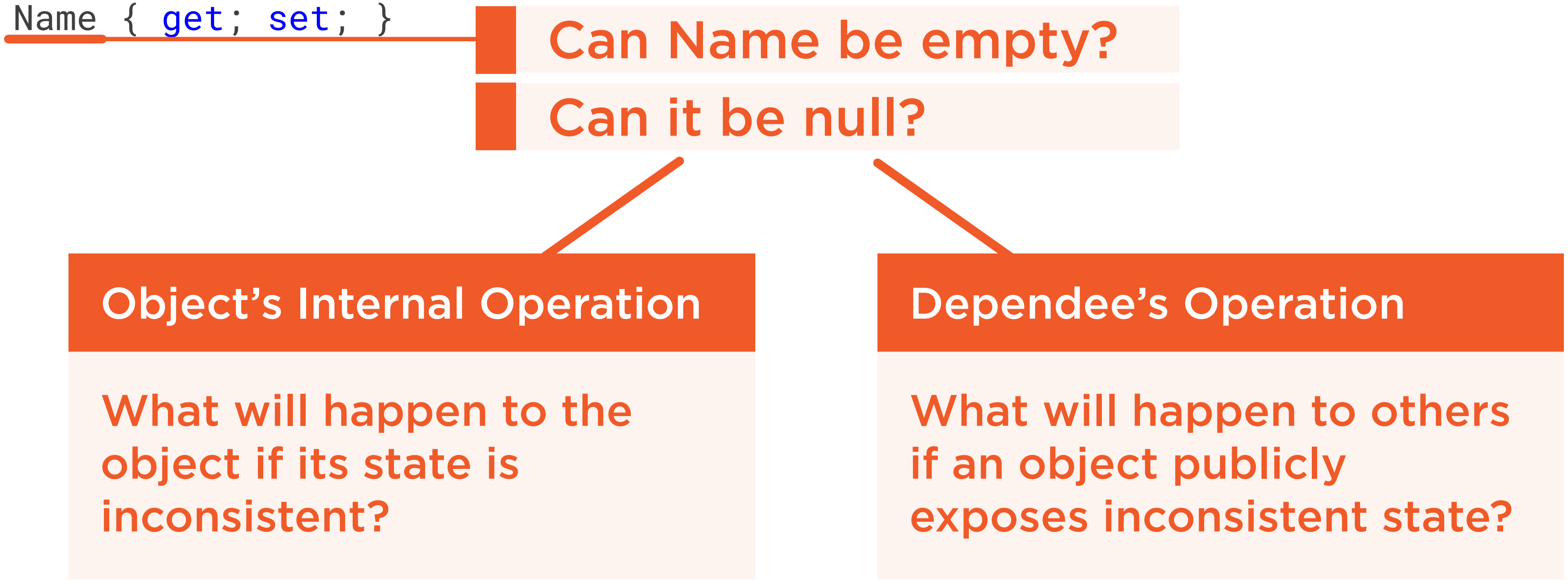
Demo

College Management Application

Support:

- Students,
- Teachers,
- Subjects,
- Exams,
- Grades, etc.

```
public class Student
{
  private string Name { get; set; }
}
```



Can Name be empty?

Can it be null?

Object’s Internal Operation

What will happen to the object if its state is inconsistent?

Dependee’s Operation

What will happen to others if an object publicly exposes inconsistent state?

```
public class Student
{
  private string Name { get; set; }
}
```

Can Name be empty?

Can it be null?

Object’s Internal Operation

What will happen to the object if its state is inconsistent?

State error will cause execution error

Defensive code is mandatory

```
public class Student
{
  private string Name { get; set; }
  public int NameLength
  {
    get
    {
      return this.Name.Length;
    }
  }
  public char NameInitialLetter
  {
    get
    {
      return this.Name[0];
    }
  }
}
```

Fails if Name is null

Fails if Name is null or empty

```
public class Student
{
  private string Name { get; set; }
  public int NameLength
  {
    get
    {
      if (this.Name != null)
        return this.Name.Length;
      else
        return 0;
    }
  }
  public char NameInitialLetter
  {
    get
    {
      if (this.Name != null && this.Name.Length > 0)
        return this.Name[0];
      else
        return ‘A’;
    }
  }
}
```

There is plenty of defensive code here

It makes solution a couple of times longer

```
public class Student
{
  private string Name { get; set; }
}
```

If only the Name could never be null or empty...

Then we would have nothing to defend from

Introduce factory function for a stateful object

```
public class Student
{
  private string Name { get; set; }

  public Student()
  {
  }
}
```

Implicit parameterless constructor is the built-in factory function

Sets numeric fields to zero, Booleans to False, references to null

```
public class Student
{
  private string Name { get; set; }

  public Student()
  {
    this.Name = null;
  }
}
```

Implicit constructor will set the Name property to null

This null reference incurs defensive code

```
public class Student
{
  private string Name { get; }

  public Student(string name)
  {
    this.Name = name;
  }
}
```

No more Name setter

Custom parameterized constructor sets Name to non-default values

```
public class Student
{
  private string Name { get; }

  public Student(string name)
  {
    if (string.IsNullOrEmpty(name))
      throw new ArgumentException();

    this.Name = name;
  }
}
```

When invalid data are received, new object is *not* constructed

```
public class Student
{
  private string Name { get; }
  public Student(string name)
  {
    if (string.IsNullOrEmpty(name))
      throw new ArgumentException();
    this.Name = name;
  }
  public int NameLength
  {
    get
    {
      return this.Name.Length;
    }
  }
  public char NameInitialLetter
  {
    get
    {
      return this.Name[0];
    }
  }
}
```

No need to defend when accessing internal state

```
public class Student
{
  private string Name { get; }
  public Student(string name)
  {
    if (string.IsNullOrEmpty(name))
      throw new ArgumentException();
    this.Name = name;
  }
  public int NameLength => this.Name.Length;
  public char NameInitialLetter => this.Name[0];
}
```

Expression-bodied syntax can be used when there is no defensive code

```
public class Student
{
  private string Name { get; }
  public Student(string name)
  {
    if (string.IsNullOrEmpty(name))
      throw new ArgumentException();
    this.Name = name;
  }
  public int NameLength => this.Name.Length;
  public char NameInitialLetter => this.Name[0];
}
```

Separation of responsibilities

Constructor ensures that only valid objects can be created

The caller will never be able to obtain an inconsistent object

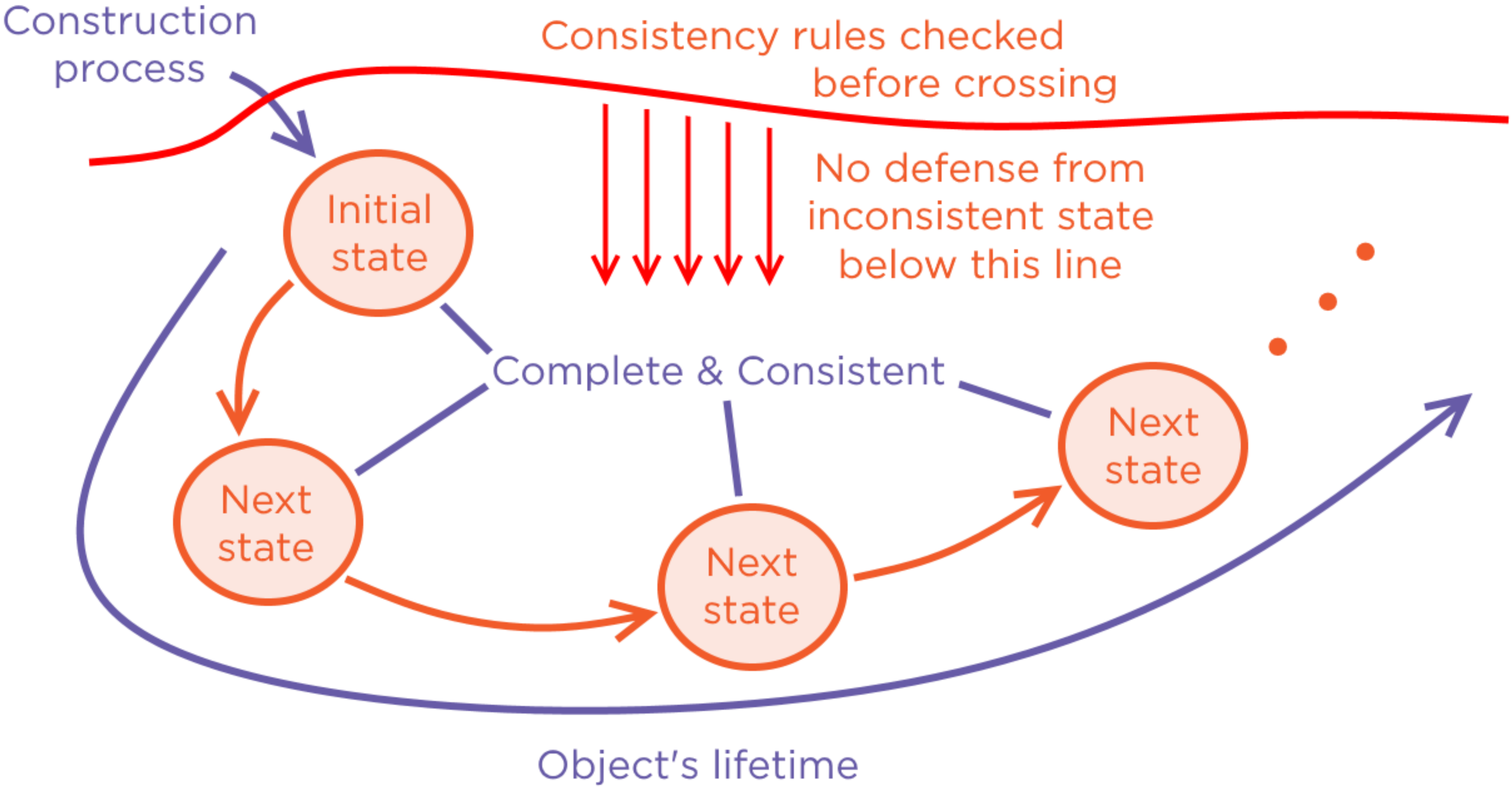
Example: Never accept null

Provide your own factory function for every stateful object

Rules of Thumb:

Define one factory function per class

Have no discrete parameters
(no enums, Booleans, etc.)


Construction process
Consistency rules checked before crossing
No defense from inconsistent state below this line
Initial state
Complete & Consistent
Next state
Next state
Next state
Object's lifetime

THE OBJECT RULE

If you have the object, then it is fine.

Demo

Constructing `ExamApplication` **object**

- Student - takes the exam
- Subject - in which the exam is taken
- Professor - administers the exam

Rules
`student != null`
`subject != null`
`professor != null`
`student` enrolled semester of `subject`
`student` has not passed exam on `subject`
`subject` taught by `professor`

```
class ExamApplication
{
  public ExamApplication(Student candidate, Subject subject, Professor admin)
  {
    if (<any argument null>)
      throw new ArgumentNullException();
    if (<any other rule violated>)
      throw new ArgumentException();
    ...
  }
}

ExamApplication appl = new ExamApplication(student, subject, professor);
```

We want an object, not an exception!

```
class ExamApplication
{
  public ExamApplication(Student candidate, Subject subject, Professor admin)
  {
    if (<any argument null>)
      throw new ArgumentNullException();
    if (<any other rule violated>)
      throw new ArgumentException();
    ...
  }
}

ExamApplication appl = new ExamApplication(student, subject, professor);
DealWith(appl);
```

Did we reach this point, or did we not?

```
class ExamApplication
{
  public ExamApplication(Student candidate, Subject subject, Professor admin)
  {
    if (<any argument null>)
      throw new ArgumentNullException();
    if (<any other rule violated>)
      throw new ArgumentException();
    ...
  }
}

try
{
  ExamApplication appl = new ExamApplication(student, subject, professor);
  DealWith(appl);
}
catch (Exception ex)
{
  DisplayWarning(ex.Message);
}
```

`try-catch` is probably the heaviest `if-then-else` you could ever think of

```
if (Alright(professor, subject, student))
{
    ExamApplication appl = new ExamApplication(student, subject, professor);
    DealWith(appl);
}
else
{
    DisplayWarning();
}
```

Validation function and constructor must contain the same validation logic

Code duplication, unless…

We can wrap validation and construction into an object!

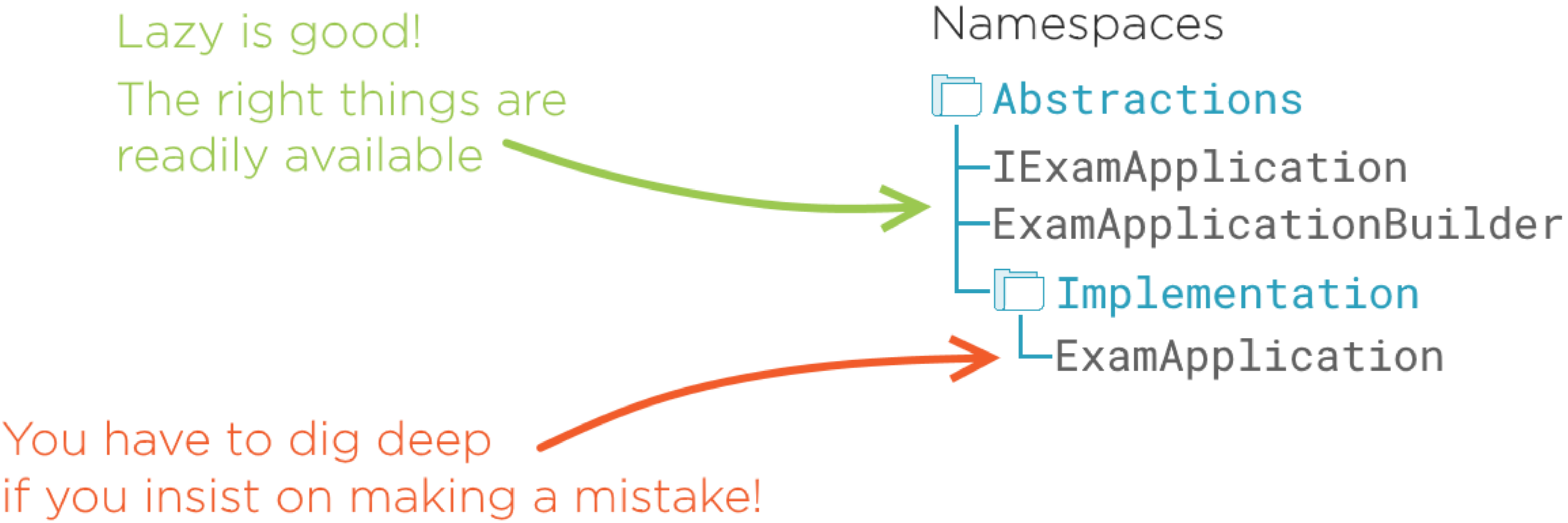
Existential Precondition

A rule which must be satisfied before an object can be constructed.

Reinforcing the Object Rule

If you have an object, then it’s fine.

Defensive Trick: Nesting Namespaces



Lazy is good!

The right things are readily available

You have to dig deep if you insist on making a mistake!

Defensive Trick: Nesting Namespaces

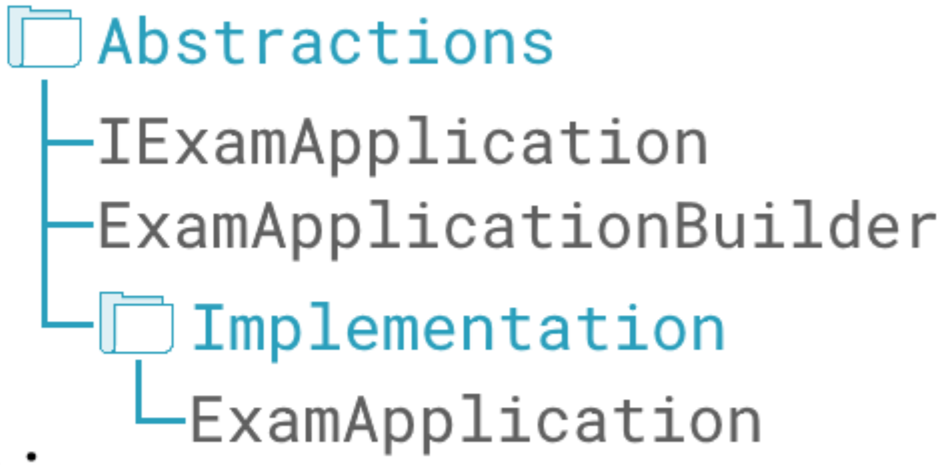
Easy and right:

```
new ExamApplicationBuilder();
```

Hard to do it wrong:

```
new Implementation.ExamApplication();
```

Namespaces



Abstractions
IExamApplication
ExamApplicationBuilder
Implementation
ExamApplication

Summary

Data-centric defense

- Data may be corrupt
- Hence need to defend

An object approach

- Wrap data inside an object
- Make the object guarantee the data are complete and consistent

Summary

Factory function

- Constructor is a proper factory
- Validates input, constructs an object

THE OBJECT RULE

IF YOU HAVE AN OBJECT, IT’S FINE

No defense after construction

No multiple constructors for a class

- They indicate multiple responsibilities
- They invite lots of defensive code
- Split them into multiple classes
- Instantiate each class in one way

Summary

Addressing complex validation rules

- Abandon constructor validation
- Introduce a Builder

The Builder concept

- Wrap validation and construction into an object itself
- Builder implementation can vary in complexity

Refer to

Tactical Design Patterns in .NET: Creating Objects

Summary

Reconsidering exceptions

- Their value is questionable, as will be seen

Object creation

- Exception instead of an object means the request was not completed

Defensive code

- Exception requires explicit defense
- Defensive design with no exceptions promises no explicit defensive code

Next module

Making Valid State Transitions Only