Tactical Design Patterns in .NET: Creating Objects

UNDERSTANDING CONSTRUCTORS AND THEIR ROLES

Zoran Horvat
OWNER AT CODING HELMET CONSULTANCY
@zoranh75 www.codinghelmet.com

The Tactical Design Patterns in .NET Series

Managing Responsibilities

March 2015

Control Flow

December 2015

Creating Objects

June 2016

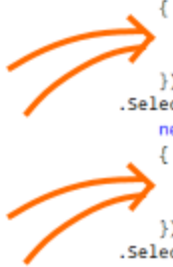
Patterns by the Book

In Theory	In Practice
Division is formal and precise	Division rarely corresponds to usage
Motivation is to produce a good design (designing phase)	Motivation is to improve existing design (refactoring phase)

a complicated method

```
public class PaintingCompany: IPainter
{

    private readonly IEnumerable<IPainter> painters;

    public PaintingCompany(IEnumerable<IPainter> painters)
    {
        this.painters = new List<IPainter>(painters);
    }

    public double Paint(double houses)
    {
        double totalVelocity = this.GetOverallVelocity();

        double totalDays =
            this.painters
            .Select(painter =>
                new
                {
                    Painter = painter,
                    Velocity = 1 / (double)painter.EstimateDays(1)
                })
            .Select(record =>
                new
                {
                    Painter = record.Painter,
                    HousesToPaint = houses * record.Velocity / totalVelocity
                })
            .Select(record => record.Painter.Paint(record.HousesToPaint))
            .Max();

        return totalDays;

    }

    private double GetOverallVelocity()
    {
        return
            this.painters
            .Select(painter => painter.EstimateDays(1))
            .Select(daysPerHouse => 1 / (double)daysPerHouse)
            .Sum();
    }

    public double EstimateDays(double houses)
    {
        return houses / this.GetOverallVelocity();
    }
}
```

```
public class PaintingCompany: IPainter
{

    private readonly IEnumerable<IPainter> painters;

    public PaintingCompany(IEnumerable<IPainter> painters)
    {
        this.painters = new List<IPainter>(painters);
    }

    public double Paint(double houses)
    {
        double totalVelocity = this.GetOverallVelocity();

        double totalDays =
            this.painters
            .Select(painter =>
                new
                {
                    Painter = painter,
                    Velocity = 1 / (double)painter.EstimateDays(1)
                })
            .Select(record =>
                new
                {
                    Painter = record.Painter,
                    HousesToPaint = houses * record.Velocity / totalVelocity
                })
            .Select(record => record.Painter.Paint(record.HousesToPaint))
            .Max();

        return totalDays;

    }

    private double GetOverallVelocity()
    {
        return
            this.painters
            .Select(painter => painter.EstimateDays(1))
            .Select(daysPerHouse => 1 / (double)daysPerHouse)
            .Sum();
    }

    public double EstimateDays(double houses)
    {
        return houses / this.GetOverallVelocity();
    }
}
```



nested calculations hide added responsibilities!

```
public class PaintingCompany: IPainter
{

    private readonly IEnumerable<IPainter> painters;
    private readonly IPaintingScheduler scheduler;

    public PaintingCompany(IEnumerable<IPainter> painters,
                           IPaintingScheduler scheduler)
    {
        this.painters = new List<IPainter>(painters);
        this.scheduler = scheduler;
    }

    public double Paint(double houses)
    {

        double totalDays =
            this.scheduler
            .Organize(this.painters, houses)
            .Select(record => record.Item1.Paint(record.Item2))
            .Max();

        return totalDays;

    }

    public double EstimateDays(double houses)
    {
        return
            this.scheduler
            .Organize(this.painters, houses)
            .Select(pair => pair.Item1.EstimateDays(pair.Item2))
            .Max();
    }
}
```

complexity is now here

```
public class ProportionalScheduler: IPaintingScheduler
{
    public IEnumerable<Tuple<IPainter, double>> Organize(
                                IEnumerable<IPainter> painters, double houses)
    {

        double totalVelocity = GetOverallVelocity(painters);

        IEnumerable<Tuple<IPainter, double>> result =
            painters
            .Select(painter =>
                new
                {
                    Painter = painter,
                    Velocity = 1 / (double)painter.EstimateDays(1)
                })
            .Select(record =>
                Tuple.Create(
                    record.Painter,
                    houses * record.Velocity / totalVelocity))
            .ToList();

        return result;

    }

    private double GetOverallVelocity(IEnumerable<IPainter> painters)
    {
        return
            painters
            .Select(painter => painter.EstimateDays(1))
            .Select(daysPerHouse => 1 / (double)daysPerHouse)
            .Sum();
    }
}
```

```
public class PaintingCompany: IPainter
{

    private readonly IEnumerable<IPainter> painters;
    private readonly IPaintingScheduler scheduler;

    public PaintingCompany(IEnumerable<IPainter> painters,
                           IPaintingScheduler scheduler)
    {
        this.painters = new List<IPainter>(painters);
        this.scheduler = scheduler;
    }

    public double Paint(double houses)
    {

        double totalDays =
            this.scheduler
            .Organize(this.painters, houses)
            .Select(record => record.Item1.Paint(record.Item2))
            .Max();

        return totalDays;

    }

    public double EstimateDays(double houses)
    {
        return
            this.scheduler
            .Organize(this.painters, houses)
            .Select(pair => pair.Item1.EstimateDays(pair.Item2))
            .Max();
    }
}
```

calling the strategy

complexity is now here

```
public class ProportionalScheduler: IPaintingScheduler
{
    public IEnumerable<Tuple<IPainter, double>> Organize(
                                IEnumerable<IPainter> painters, double houses)
    {

        double totalVelocity = GetOverallVelocity(painters);

        IEnumerable<Tuple<IPainter, double>> result =
            painters
            .Select(painter =>
                new
                {
                    Painter = painter,
                    Velocity = 1 / (double)painter.EstimateDays(1)
                })
            .Select(record =>
                Tuple.Create(
                    record.Painter,
                    houses * record.Velocity / totalVelocity))
            .ToList();

        return result;

    }

    private double GetOverallVelocity(IEnumerable<IPainter> painters)
    {
        return
            painters
            .Select(painter => painter.EstimateDays(1))
            .Select(daysPerHouse => 1 / (double)daysPerHouse)
            .Sum();
    }
}
```

strategy

refer to Managing Responsibilities course for full example

```
public class PaintingCompany: IPainter
{

    private readonly IEnumerable<IPainter> painters;
    private readonly IPaintingScheduler scheduler;

    public PaintingCompany(IEnumerable<IPainter> painters,
                           IPaintingScheduler scheduler)
    {
        this.painters = new List<IPainter>(painters);
        this.scheduler = scheduler;
    }

    public double Paint(double houses)
    {

        double totalDays =
            this.scheduler
            .Organize(this.painters, houses)
            .Select(record => record.Item1.Paint(record.Item2))
            .Max();

        return totalDays;

    }

    public double EstimateDays(double houses)
    {
        return
            this.scheduler
            .Organize(this.painters, houses)
            .Select(pair => pair.Item1.EstimateDays(pair.Item2))
            .Max();
    }
}
```

strategy object is a dependency

calling the strategy

complexity is now here

```
public class ProportionalScheduler: IPaintingScheduler
{
    public IEnumerable<Tuple<IPainter, double>> Organize(
                                IEnumerable<IPainter> painters, double houses)
    {

        double totalVelocity = GetOverallVelocity(painters);

        IEnumerable<Tuple<IPainter, double>> result =
            painters
            .Select(painter =>
                new
                {
                    Painter = painter,
                    Velocity = 1 / (double)painter.EstimateDays(1)
                })
            .Select(record =>
                Tuple.Create(
                    record.Painter,
                    houses * record.Velocity / totalVelocity))
            .ToList();

        return result;

    }

    private double GetOverallVelocity(IEnumerable<IPainter> painters)
    {
        return
            painters
            .Select(painter => painter.EstimateDays(1))
            .Select(daysPerHouse => 1 / (double)daysPerHouse)
            .Sum();
    }
}
```

strategy

Tactical Design Patterns in .NET

Managing Responsibilities

Helps decide which responsibility to put into which class.

Composite, Chain of Responsibility, Visitor, etc.

Control Flow

Simplify control flow, remove branching, nulls, loops.

Null Object, Special Case, Iterator, Map-Reduce, etc.

Creating Objects

Helps decide how to create objects and object graphs.

Abstract Factory, Factory Method, Builder, Specification.

Division of Creational Patterns

Common Literature	This Course
Abstract Factory	Abstract Factory
Builder	Builder
Factory Method	Factory Method (with lambdas)
Prototype	~~Prototype~~
Singleton	~~Singleton~~
	Specification (not a creational pattern)

Introducing the Specification Pattern

Specification by the Book

Describes an expression (Boolean, arithmetic)

Validates an object

Specification as Builder

Describes object and its parts

Specializes that knowledge to create an object

Morphing Creational Patterns

ntations can be connected and combined

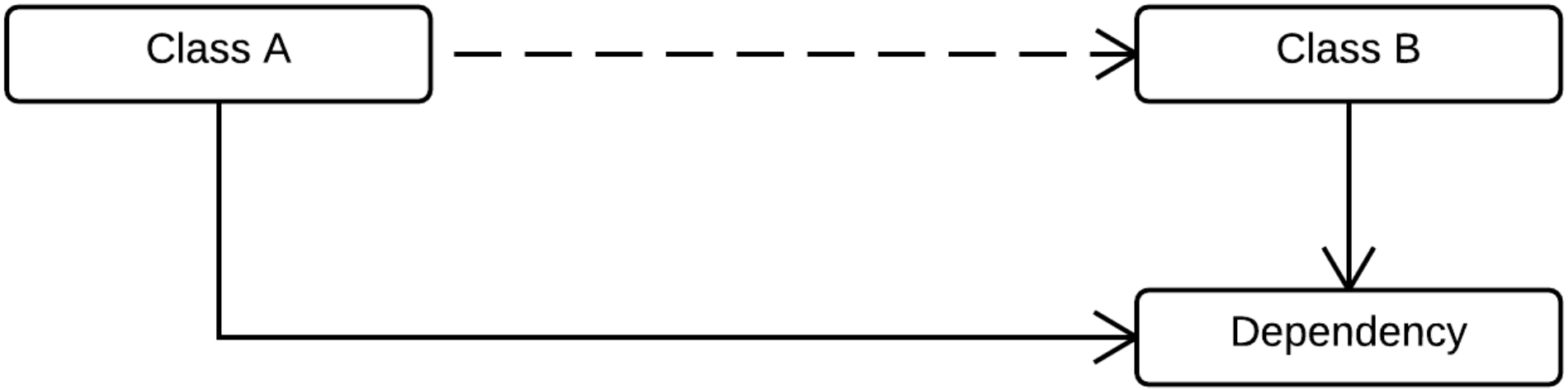
Do’s and Don’ts of Patterns

Abstract Factory

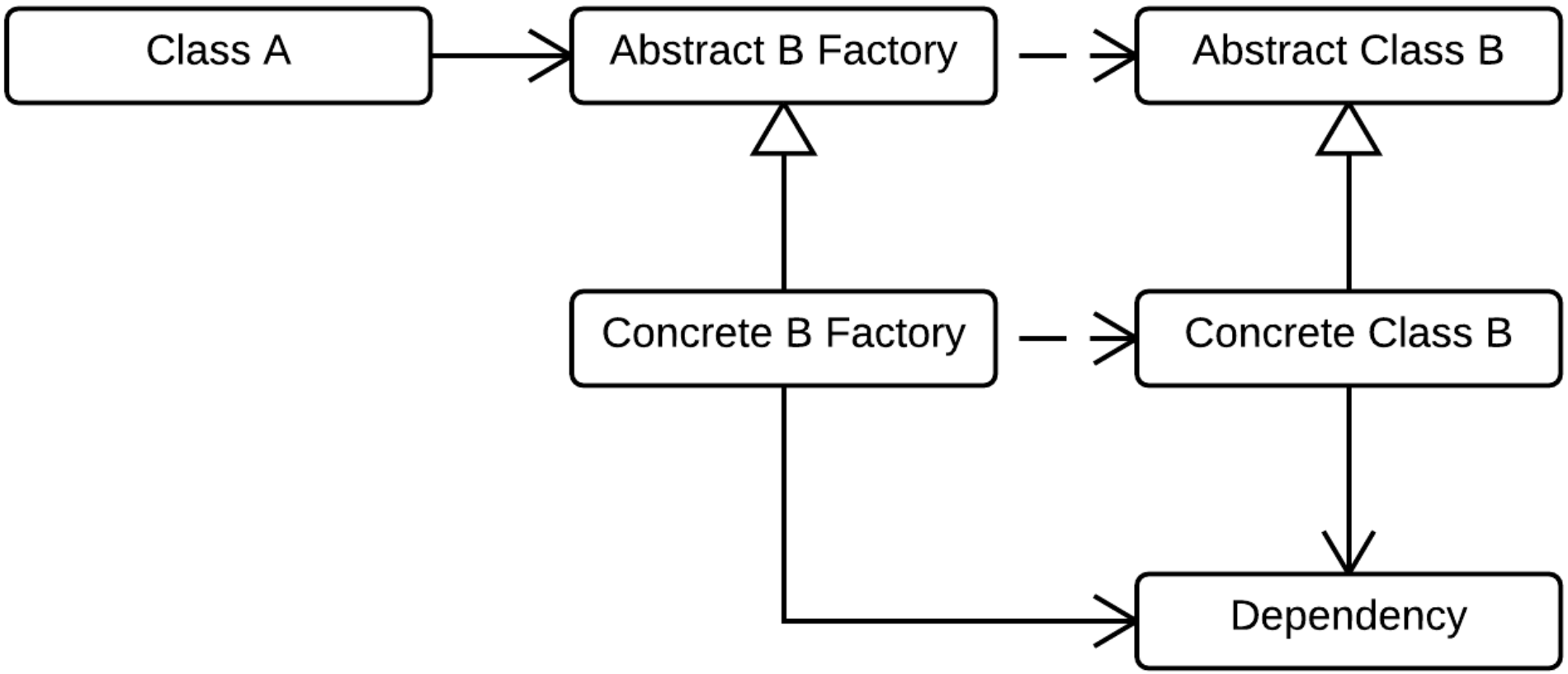
ıcrete interface implementation

dependencies issue

Stopping the Leaky Dependencies



Stopping the Leaky Dependencies



Knowing the Path ≠ Walking the Path

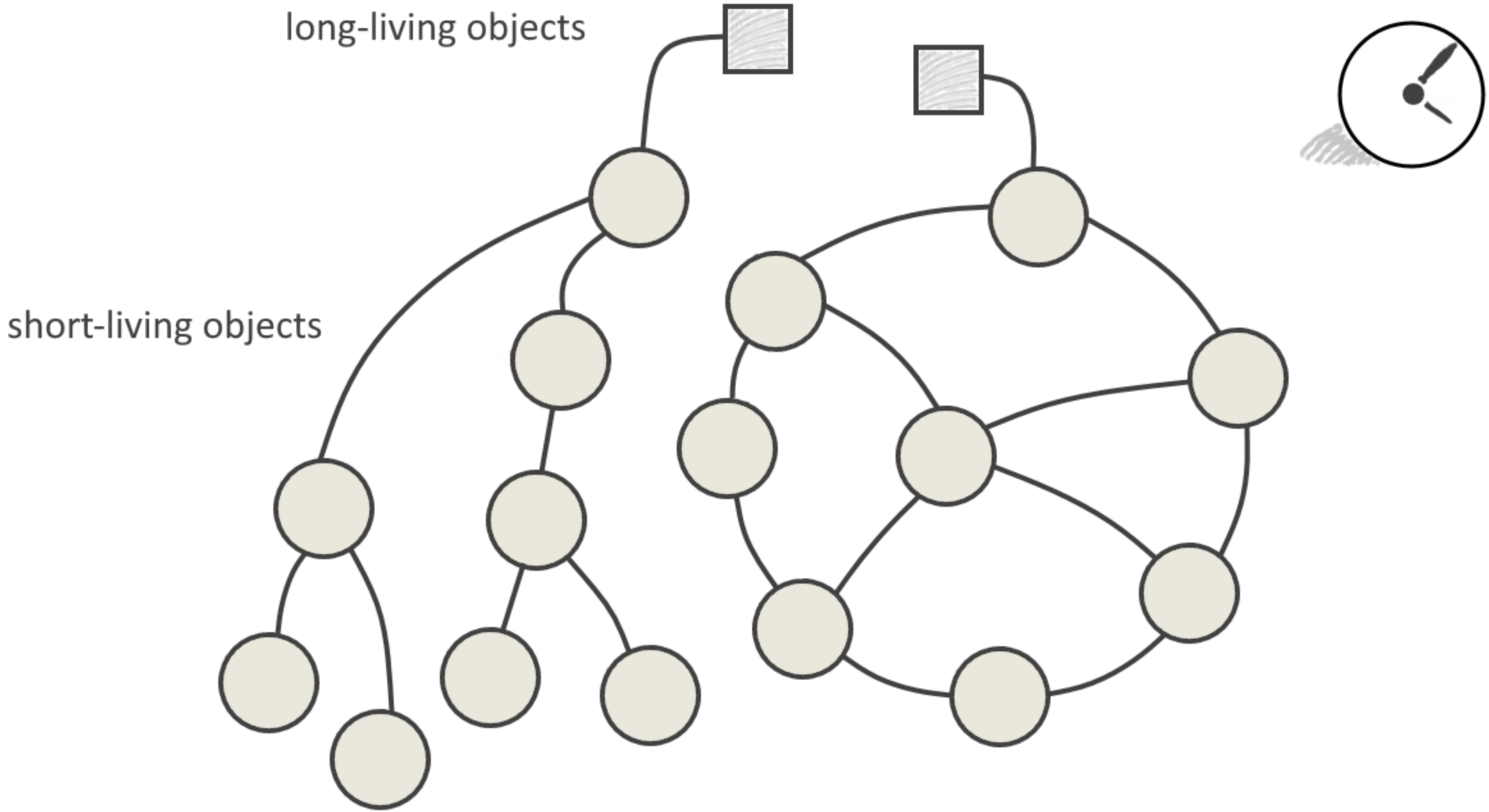
Step 1: Understand the design issue.

ep 2: Apply a pattern to remove it.

Q: What does it mean
to create a new object?

What does it take to create a brand new object?

an object


long-living objects
short-living objects

response serializer

application layer

response stream

HTTP request

response object

DTOs

request deserializer

domain layer

request

request parameters

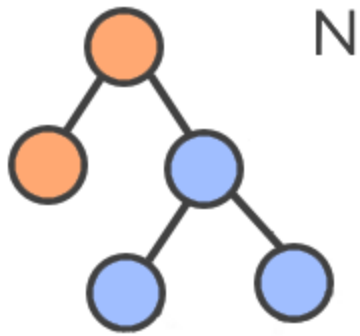
model

controller

repository

to database

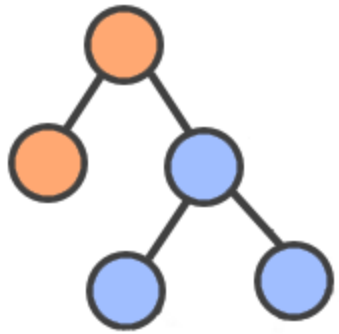
```
new DateTime(2016, 3, 8)

        new Painter(managingDirector, workScheduler)
```



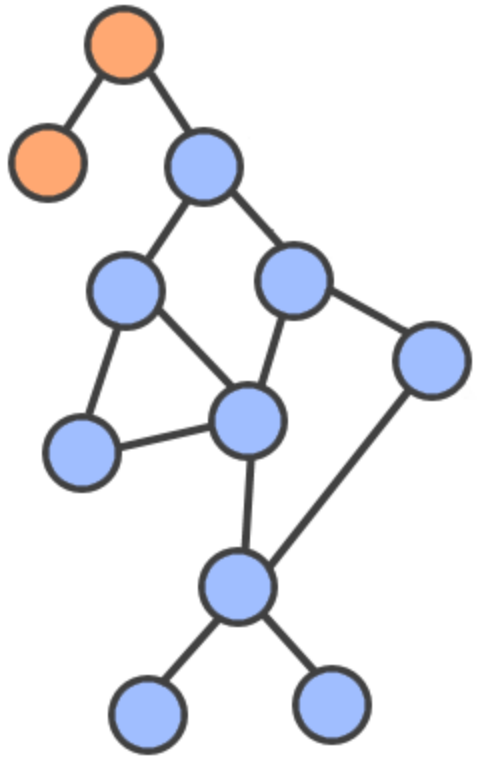
New object

Dependencies

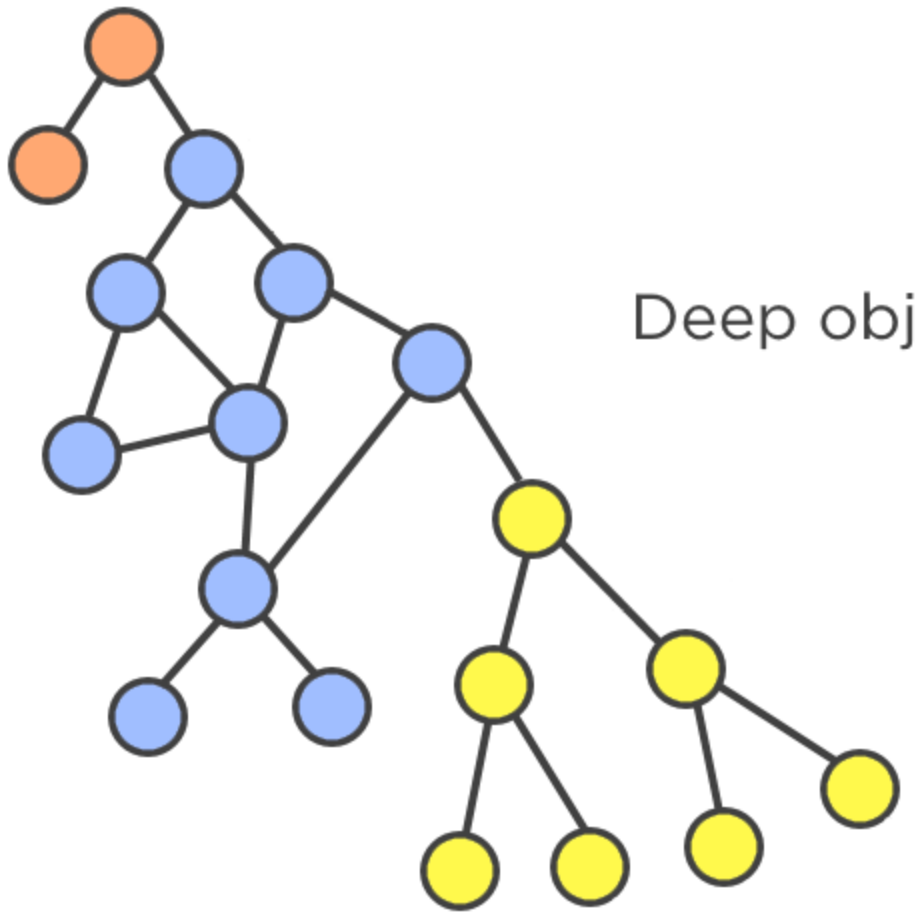
Dependencies of dependencies



Object graph

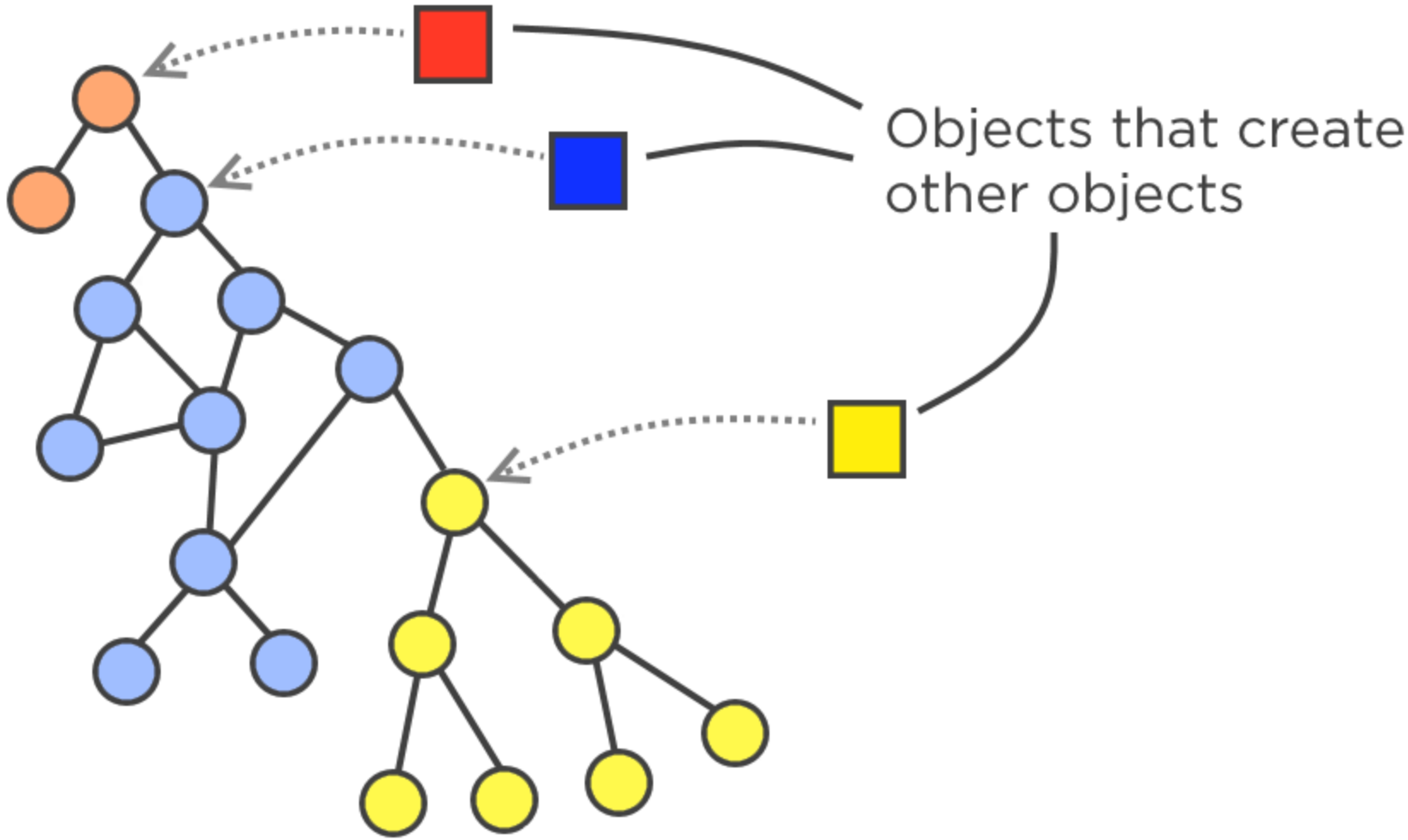


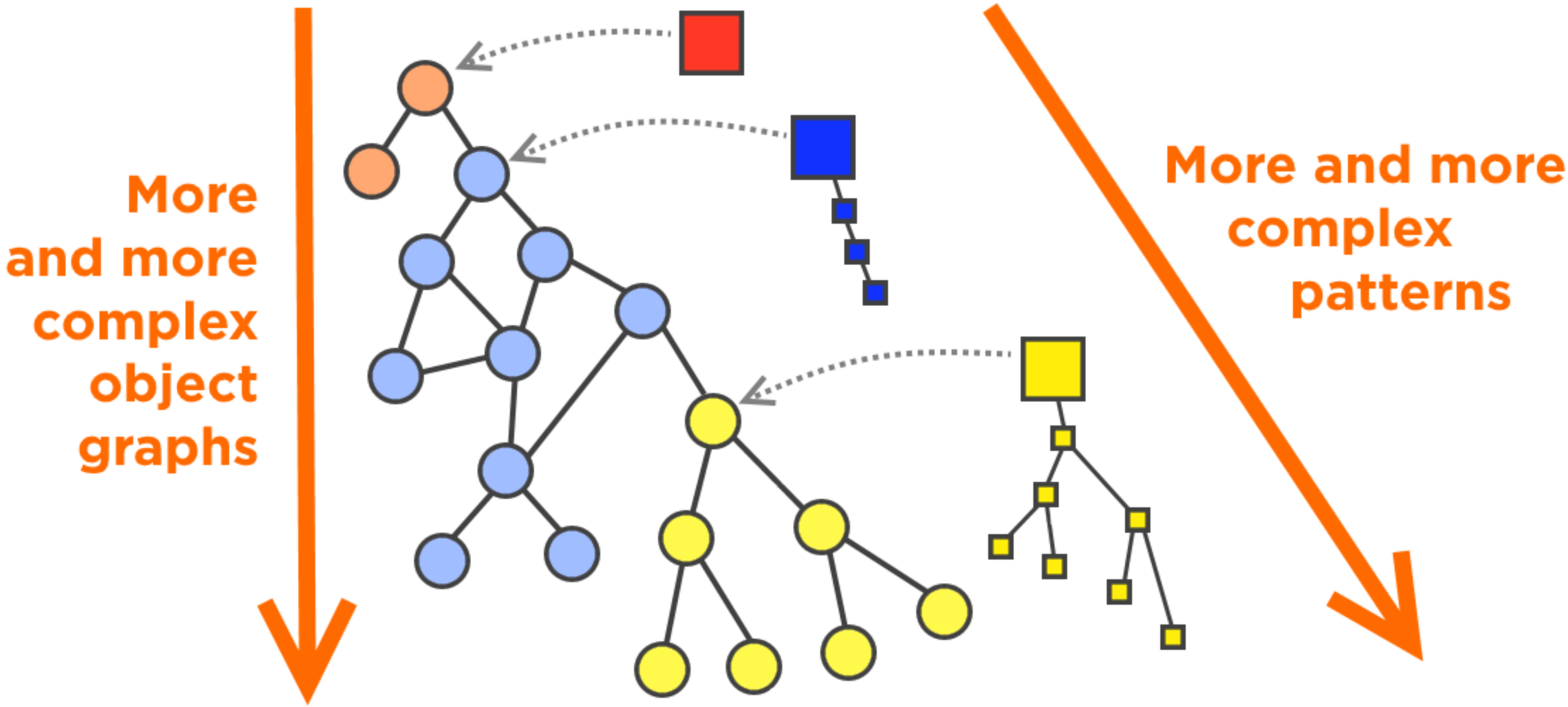
Deep object graph

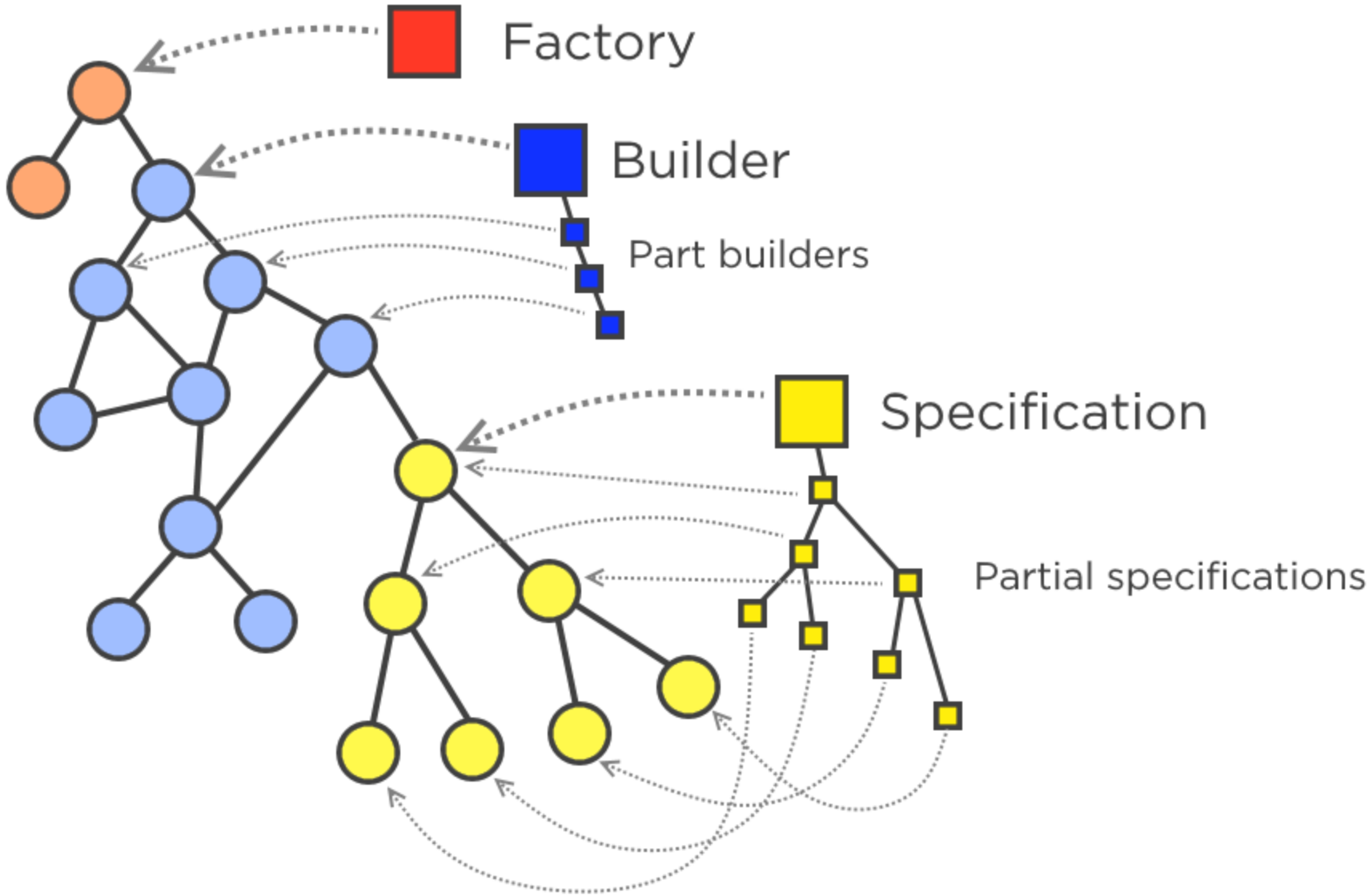


Deep object graph

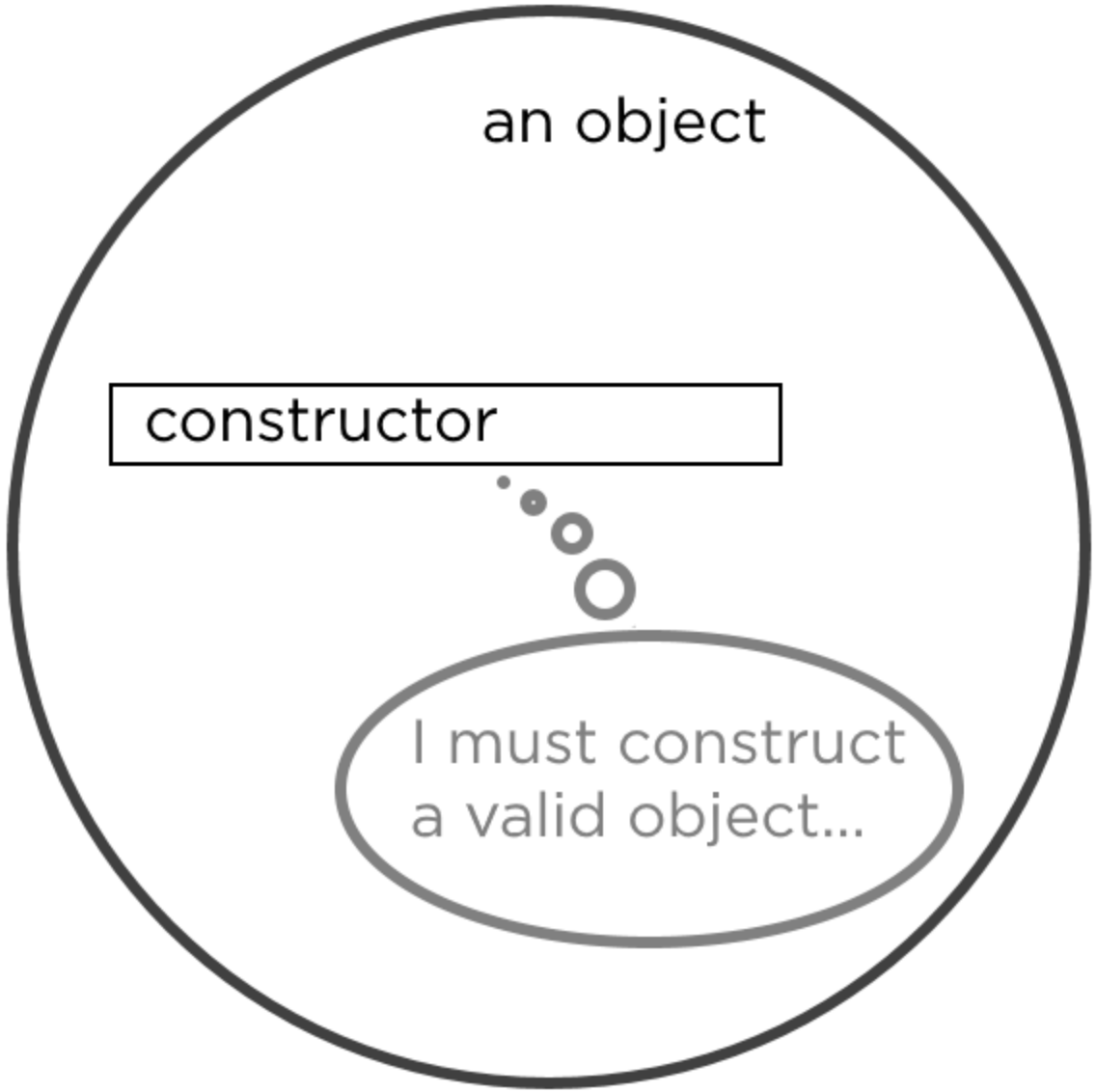
Sub-graph
with own
root object

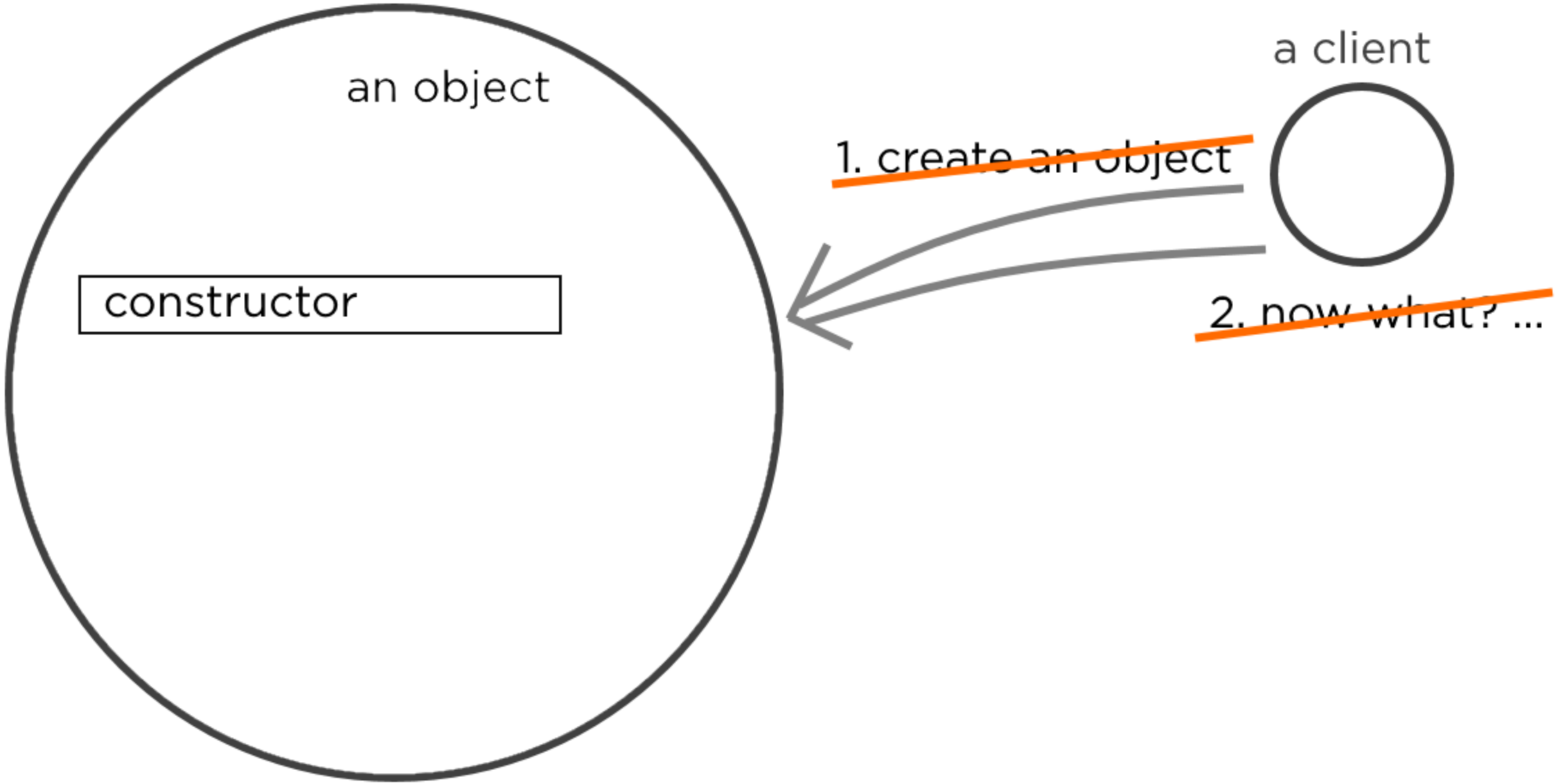

Objects that create
other objects

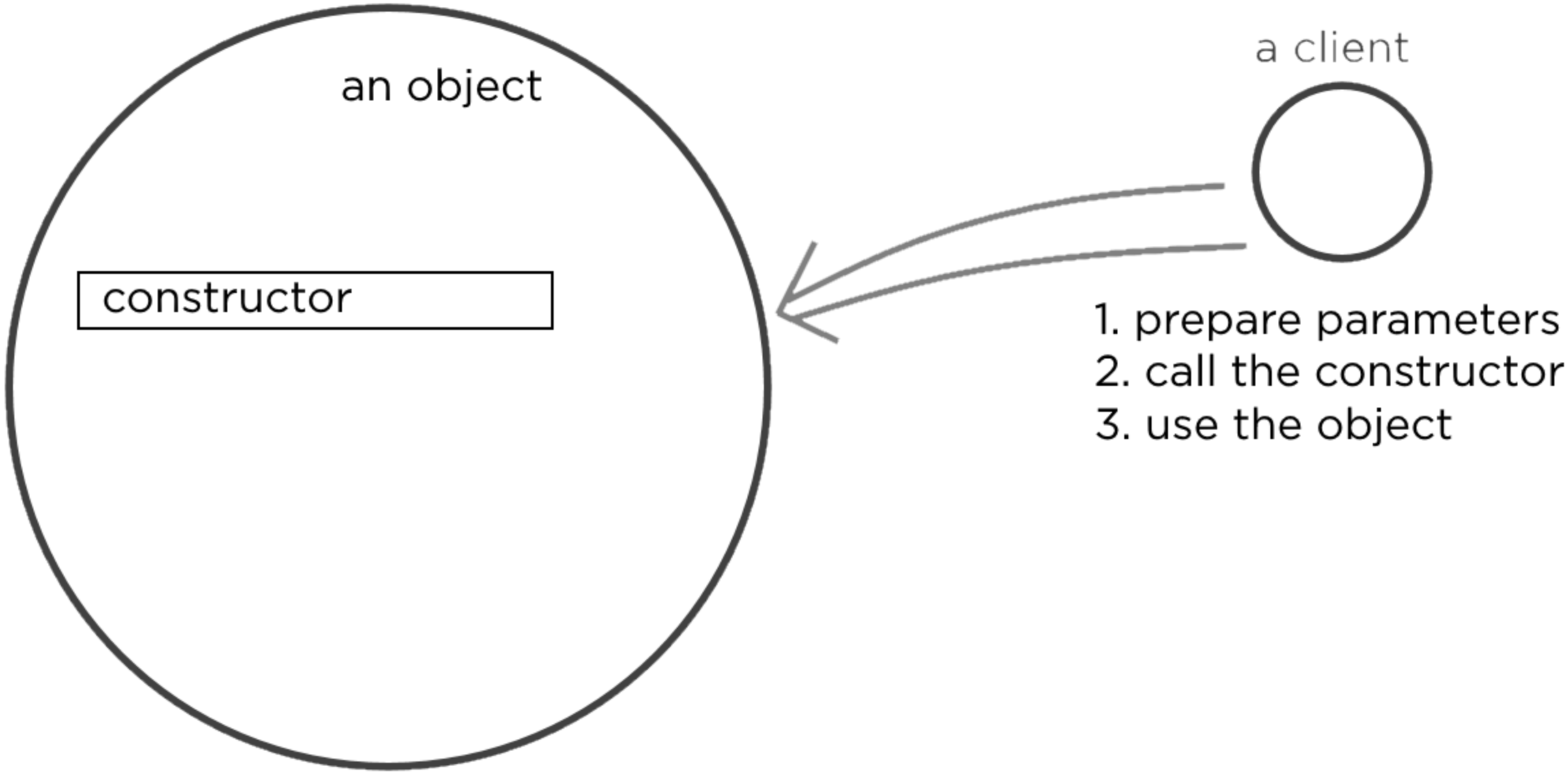

More and more complex object graphs
More and more complex patterns

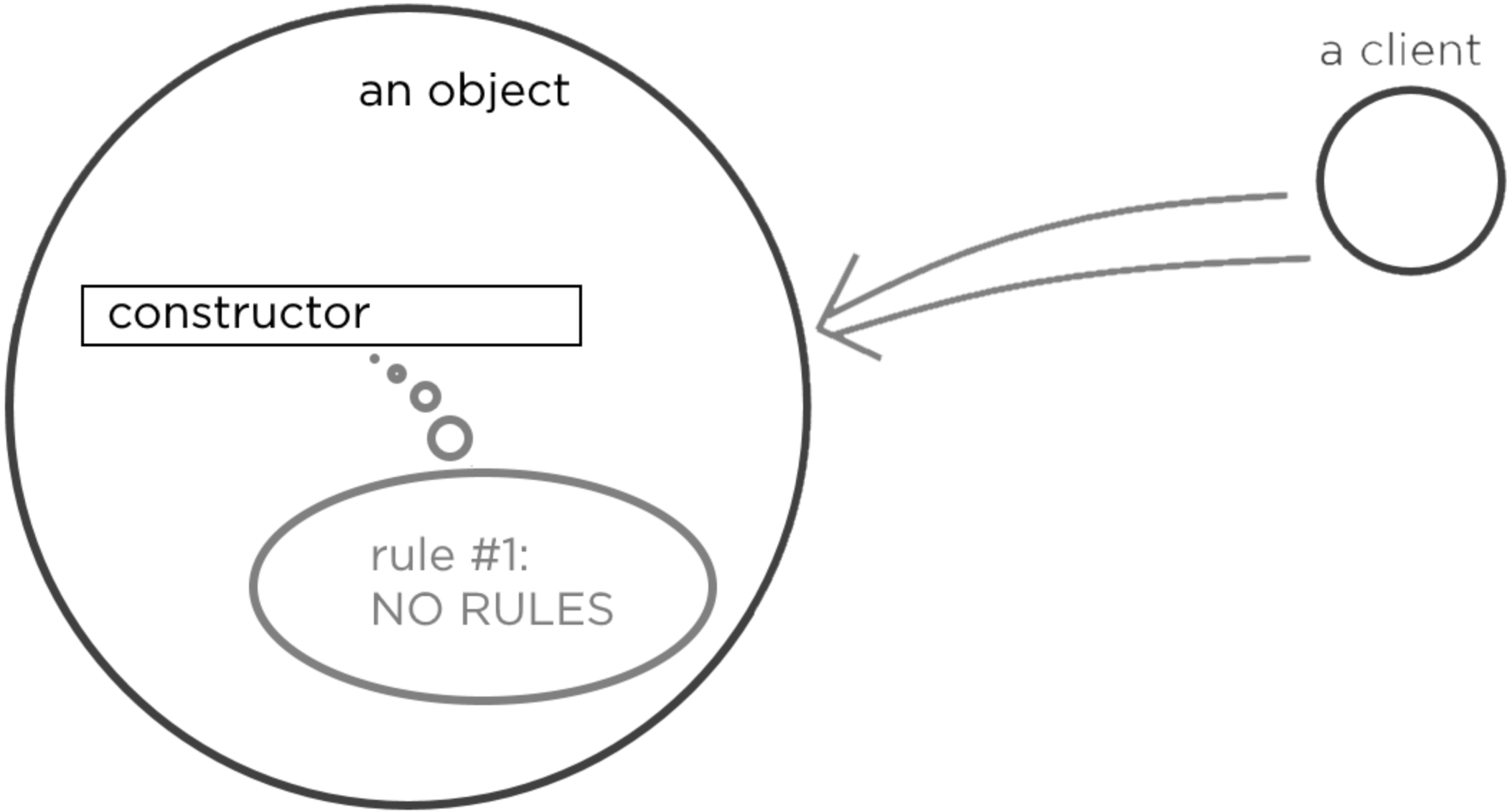

Factory
Builder
Part builders
Specification
Partial specifications

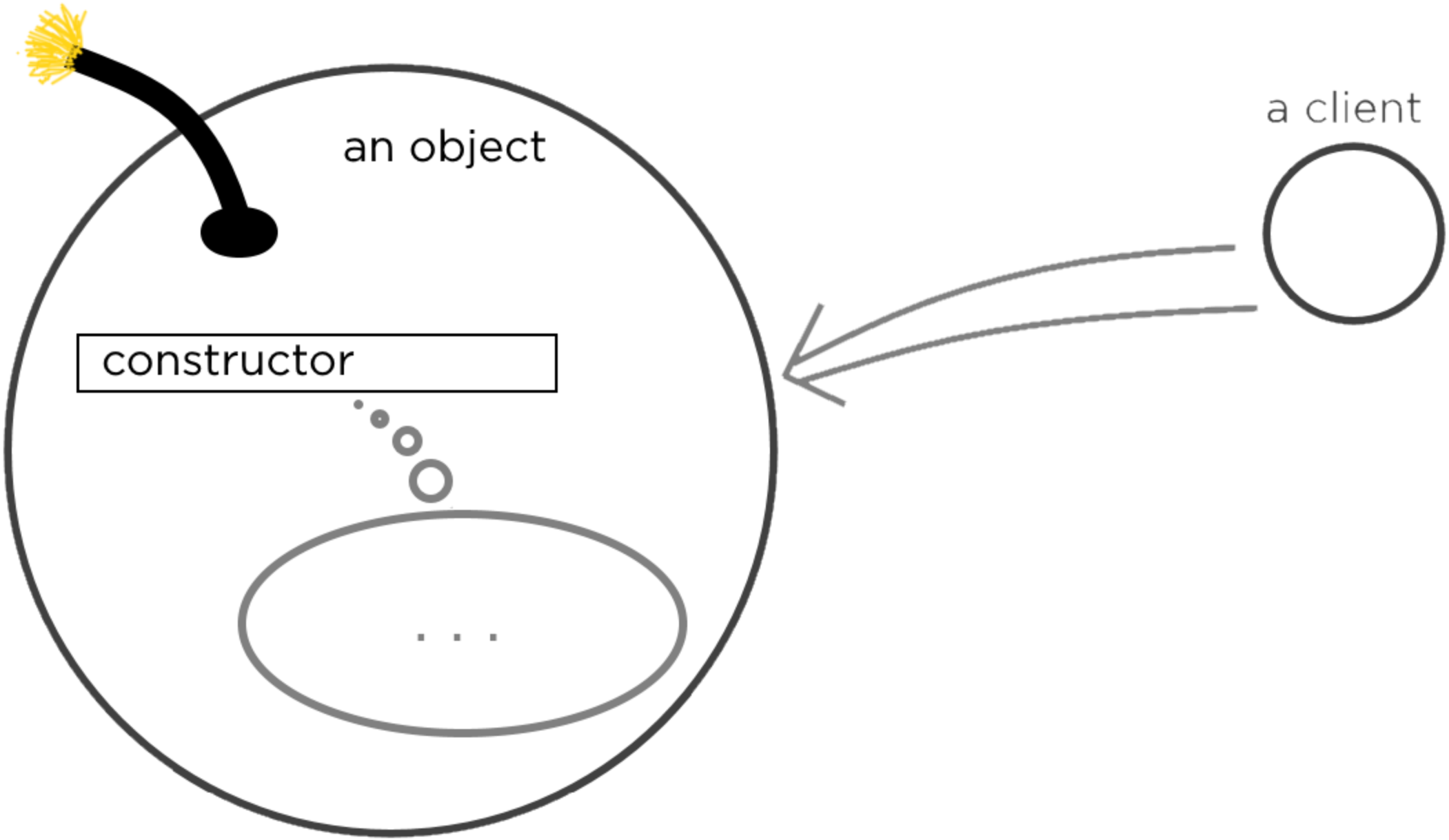
an object

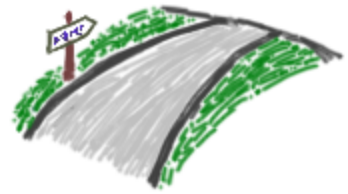

an object
constructor
I must construct
a valid object...


an object
constructor
a client
1. create an object
2. now what? ...

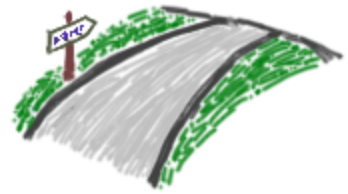

an object
constructor
a client
1. prepare parameters
2. call the constructor
3. use the object


an object
constructor
rule #1:
NO RULES
a client


an object
constructor
. . .
a client

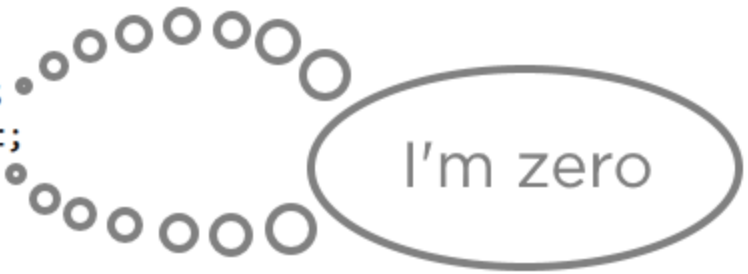


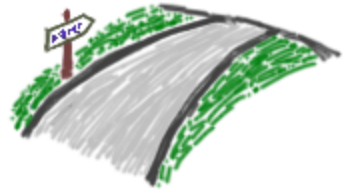
```
class RoadSegment
{
    private double distanceKm;
    private TimeSpan timeSpent;

    public double DistanceKm
    {
        get { return this.distanceKm; }
        set
        {
            if (value <= 0) throw new ArgumentException();
            this.distanceKm = value;
        }
    }

    public TimeSpan TimeSpent
    {
        get { return this.timeSpent; }
        set
        {
            if (value <= new TimeSpan(0)) throw new ArgumentException();
            this.timeSpent = value;
        }
    }

    public double VelocityKph => this.DistanceKm / this.TimeSpent.TotalHours;
}
```



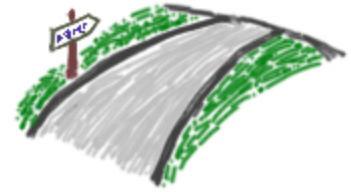
```
class RoadSegment
{
    private double distanceKm;
    private TimeSpan timeSpent;

    public double DistanceKm
    {
        get { return this.distanceKm; }
        set
        {
            if (value <= 0) throw new ArgumentException();
            this.distanceKm = value;
        }
    }

    public TimeSpan TimeSpent
    {
        get { return this.timeSpent; }
        set
        {
            if (value <= new TimeSpan(0)) throw new ArgumentException();
            this.timeSpent = value;
        }
    }

    public double VelocityKph => this.DistanceKm / this.TimeSpent.TotalHours;
}
```

```
class RoadSegment
{
    private double distanceKm;
    private TimeSpan timeSpent;

    public double DistanceKm
    {
        get { return this.distanceKm; }
        set
        {
            if (value <= 0) throw new ArgumentException();
            this.distanceKm = value;
        }
    }

    public TimeSpan TimeSpent
    {
        get { return this.timeSpent; }
        set
        {
            if (value <= new TimeSpan(0)) throw new ArgumentException();
            this.timeSpent = value;
        }
    }

    public double VelocityKph => this.DistanceKm / this.TimeSpent.TotalHours;
}
```





```
class RoadSegment
{
    private double distanceKm;
    private TimeSpan timeSpent;

    public double DistanceKm
    {
        get { return this.distanceKm; }
        set
        {
            if (value <= 0) throw new ArgumentException();
            this.distanceKm = value;
        }
    }

    public TimeSpan TimeSpent
    {
        get { return this.timeSpent; }
        set
        {
            if (value <= new TimeSpan(0)) throw new ArgumentException();
            this.timeSpent = value;
        }
    }

    public double VelocityKph => this.DistanceKm / this.TimeSpent.TotalHours;
}
```



```
RoadSegment segment = new RoadSegment();

Console.WriteLine("Your velocity was {0}km/h",
                  segment.VelocityKph);
```

```
Your velocity was NaNkm/h
```

Make invalid states
non-representable.

Make Invalid States Non-representable

Types must convey rules of the world

A nonexistent concept cannot be represented by an existing type

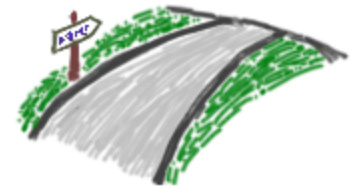
Objects must represent valid states only

If there is an object,
then the object is fine.

If There is an Object, Then the Object is Fine

ondition was violated ⇒ Constructor will fail

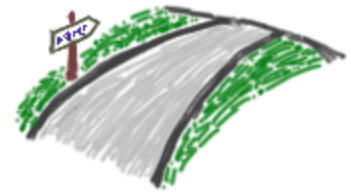
are satisfied ⇒ Constructor will execute



```
void Report(IEnumerable<RoadSegment> segments)
{
    double totalDistanceKm = 0;
    TimeSpan totalTimeSpent = new TimeSpan();

    foreach (RoadSegment segment in segments)
    {
        Console.WriteLine("{0}km at {1}km/h",
                          segment.DistanceKm, segment.VelocityKph);
        totalDistanceKm += segment.DistanceKm;
        totalTimeSpent += segment.TimeSpent;
    }

    Console.WriteLine("TOTAL {0}km at {1}km/h average",
                      totalDistanceKm,
                      totalDistanceKm / totalTimeSpent.TotalHours);
}
```

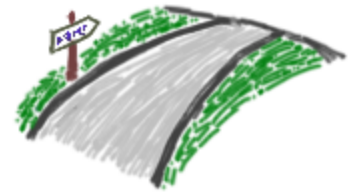
```
Report(
    new[]
    {
        new RoadSegment(),
        new RoadSegment()
    });
```

```
0km at NaNkm/h
0km at NaNkm/h
TOTAL 0km at NaNkm/h average
```



```
class RoadSegment
{
    private double distanceKm;
    private TimeSpan timeSpent;

    public RoadSegment(double distanceKm, TimeSpan timeSpent)
    {
        this.DistanceKm = distanceKm;
        this.TimeSpent = timeSpent;
    }

    public double DistanceKm...

    public TimeSpan TimeSpent...

    public double VelocityKph => this.DistanceKm / this.TimeSpent.TotalHours;

}
```

both are throwing



```
class RoadSegment
{
    private double distanceKm;
    private TimeSpan timeSpent;

    public RoadSegment(double distanceKm, TimeSpan timeSpent)
    {
        this.DistanceKm = distanceKm;
        this.TimeSpent = timeSpent;
    }

    public double DistanceKm...

    public TimeSpan TimeSpent...

    public double VelocityKph => this.DistanceKm / this.TimeSpent.TotalHours;

}
```

```
Report(
    new[]
    {
        new RoadSegment(1.5, TimeSpan.FromMinutes(62)),
        new RoadSegment(2.25, TimeSpan.FromMinutes(0))
    });
```

causes ArgumentException

Summary

Why the design pattern?

- A tool to solve issues in existing design
- Good for refactoring

The question of creating new objects

- It all ends in the class constructor
- Constructor guarantees new objects are created in consistent state

Further investigation

- How to ensure consistency of new objects?

What Follows

The problem of constructing new objects

- We will start with simple cases
- Then progress to more complex ones

The goal

- Build entire object graphs in one call

There is a lot of theory involved

- Some modules will only cover parts of the programming theory

Module 2 Abstract Factory

- Abstract Factory pattern by the book
- There will be challenges with basic implementation
- The problem is in “Abstract”

Module 2 Abstract Factory

Module 3 Avoiding Excess Abstractness

- One common way of reducing abstractness of Abstract Factory
- Abstract Factory for a single concrete product
- ASP.NET MVC Framework uses this tactic
- The problem of too much abstractness will always be there

Module 2	Abstract Factory
Module 3	Avoiding Excess Abstractness
Module 4	Covariance and Contravariance

- The problem of understanding interactions between abstract data types (ADTs)
- We will cover covariance and contravariance of generic types
- We must understand variance in order to understand ADTs

Module 2	Abstract Factory
Module 3	Avoiding Excess Abstractness
Module 4	Covariance and Contravariance
Module 5	Substitution and Liskov Substitution Principles

- Capitalize on understanding the type variance
- Substitution principle is about structural subtyping
- LSP is about behavioral subtyping

Module 2	Abstract Factory
Module 3	Avoiding Excess Abstractness
Module 4	Covariance and Contravariance
Module 5	Substitution and Liskov Substitution Principles
Module 6	Builder

- Outline the limitations of Abstract Factory
- Break out in direction of concrete types
- ... which will cause troubles with too much concreteness

- The issue of calling protocols
- Creating the object may come with a lot of rules
- Introduce a proper calling protocol
- Builder is ideal to hold the protocol implementation

- Client doesn't want to know too much
- We need to step back to lower level of complexity
- Factory Method as a thin wrapper around the Builder
- Keep the power of Builder
- Use the simplicity of Factory Method

Module 2	Abstract Factory
Module 3	Avoiding Excess Abstractness
Module 4	Covariance and Contravariance
Module 5	Substitution and Liskov Substitution Principles
Module 6	Builder
Module 7	Calling Protocols and Builder
Module 8	Factory Method with Lambdas
Module 9	Building Complex Objects with Specification

- There are limitations inherent to Builders
- Hoist the Builder up to the level of Specification
- Specify future object, but don't build it yet
- Wrap the Specification inside a thin Builder to build the object
- This approach is the paradigm shift
- Lets us specify objects and graphs of extreme complexity
- Lets us specify objects with complex rules and protocols

- Specification pattern in full swing
- Describes entire object graphs
- Enforces all rules on the graph
- Enforces all calling protocols
- Verifies all interactions between parts of the graph
- Easily turned into a plain Builder
- Builder easily turned into Factory

Module 2	Abstract Factory
Module 3	Avoiding Excess Abstractness
Module 4	Covariance and Contravariance
Module 5	Substitution and Liskov Substitution Principles
Module 6	Builder
Module 7	Calling Protocols and Builder
Module 8	Factory Method with Lambdas
Module 9	Building Complex Objects with Specification
Module 10	Building Object Graphs with Specification

- All creational patterns are connected
- Factory can be upgraded to Builder
- Builder can be upgraded to Specification
- Specification can be reduced to Builder
- Builder can be reduced to Factory
- Pick the pattern which corresponds with complexity of the target product

Module 2	Abstract Factory
Module 3	Avoiding Excess Abstractness
Module 4	Covariance and Contravariance
Module 5	Substitution and Liskov Substitution Principles
Module 6	Builder
Module 7	Calling Protocols and Builder
Module 8	Factory Method with Lambdas
Module 9	Building Complex Objects with Specification
Module 10	Building Object Graphs with Specification

Next module –

Advancing from Constructor to Abstract Factory