Factory (virtual constructors)

The INFDEV team

Hogeschool Rotterdam
Rotterdam, Netherlands

INFDEV02-4

Introduction

Introduction

Lecture topics

- POlymorphic constructors
- The factory design pattern
- Abstract factory
- Conclusions

The problem

The problem

Introduction

- Sometimes, we know which interface to instantiate, but not its concrete class
- Interfaces specify no constructors, external code is necessary to express such mechanism
- This leads to conditionals in client code to determine which concrete class to instantiate

The problem

Introduction

- In particular, we will study the **factory design pattern** (a creational pattern)
- This moves the construction logic to a new class, thereby simulating virtual constructors
- This design pattern is going to be the topic of this lecture

The problem

Our first example

- Consider the following implementations of `Animal`

```
1  interface Animal {
2    void MakeSound();
3  }
4  class Cat : Animal {
5    public void MakeSound() {
6      ...
7    }
8  }
9  class Dog : Animal {
10   public void MakeSound() {
11     ...
12   }
13 }
14 class Dolphin : Animal {
15   public void MakeSound() {
16     ...
17   }
18 }
```

The problem

Consuming our "animals": issue with constructors

- We read the id of an animal from the console, and then want to instantiate it
- Such logic cannot be expressed inside the `Animal` interface
- Therefore, we need the client code to explicitly implement the selection mechanism

The problem

Consuming our "animals": from the client

- Our client now reads the input and uses it to instantiate a concrete animal
- The collection contains only `Animals`

```
1  List<Animal> animals = new List<Animal>();
2  int id = -1;
3  while (id != 0) {
4    id = Int32.Parse(Console.ReadLine());
5    if((id == 1)) {
6      animals.Add(new Cat());
7    }
8    if((id == 2)) {
9      animals.Add(new Dog());
10   }
11   if((id == 3)) {
12     animals.Add(new Bird());
13   }
14  }
```

The problem

Consuming our "animals": from different clients

- What about all other clients interested with consuming our animals?
- Repeating code is error prone and not maintainable
- What about adding new animals? Does it still work? How do we notify the other clients about such change?
- The manual solution just seen is neither maintainable, nor flexible

The problem

Defining instantiation logic once

- We wish to isolate instantiation logic so that it becomes reusable
- It would be ideal to add such logic in the only point that is common to all our concrete animals: the interface
- Unfortunately, interfaces do not allow constructors[a]

[a] And it actually makes sense!

The problem

Defining instantiation logic once

- We can use special-purpose classes to express such instantiation mechanism
- How?

The problem

Defining instantiation logic once

- We can use special-purpose classes to express such instantiation mechanism
- How?
- By defining special methods that create and return concrete classes belonging to some polymorphic type
- Such special-purpose classes are called abstract classes

The problem

Making our "Animal" abstract

- We can of course define our `Animal` abstract
- What follows?

The problem

Making our "Animal" abstract

- We can of course define our `Animal` abstract
- What follows?
- We can have a static method `Instantiate` that implements the instantiation mechanism, introduced at the beginning of this example, and returns a concrete animal
- `Instantiate` is static, since we cannot call it directly (`Animal` is abstract)
- We can leave `MakeSound` as a signature
- In the following we show our abstract `Animal` and we consume it

The problem


```
1  abstract class Animal {
2    public static Animal Instantiate(int id) {
3      if((id == 1)) {
4        return new Cat();
5      }
6      if((id == 2)) {
7        return new Dog();
8      }
9      if((id == 3)) {
10       return new Bird();
11     }
12   }
13   public abstract  void MakeSound();
14 }
15 ...
16 Animal an_animal = Animal.Instantiate(Int32.Parse(Console.ReadLine()));
17 an_animal.MakeSound();
```

The problem

Which in Java then becomes:

```
1  abstract class Animal {
2    public static Animal Instantiate(int id) {
3      if((id == 1)) {
4        return new Cat();
5      }
6      if((id == 2)) {
7        return new Dog();
8      }
9      if((id == 3)) {
10       return new Bird();
11     }
12   }
13   public abstract  void MakeSound();
14 }
15 ...
16 Animal an_animal = Animal.Instantiate(Integer.parseInt(new Scanner(System.in
       ).nextLine()));
17 an_animal.MakeSound();
```

The problem

Consideration

- In the last version of our `Animal` class we managed to define instantiation logic at polymorphic level, instead of carrying such task on all clients
- Now there is only one *entry point* where we can create our concrete animals: `Animal`!

The problem

Consideration

- Whenever a client wishes to instantiate an animal it has to ask `Animal`
- We now can say that `Animal` is not only the polymorphic type for our concrete animals, but also a **factory** of animals
- This instantiation mechanism belongs to the so called *simple factory method* design pattern

The problem

Consideration

- In the following we will study the formalization of such pattern together with other patterns belonging to this family
- Patterns for providing instantiation mechanisms are generally referred to as: **factory design patterns**

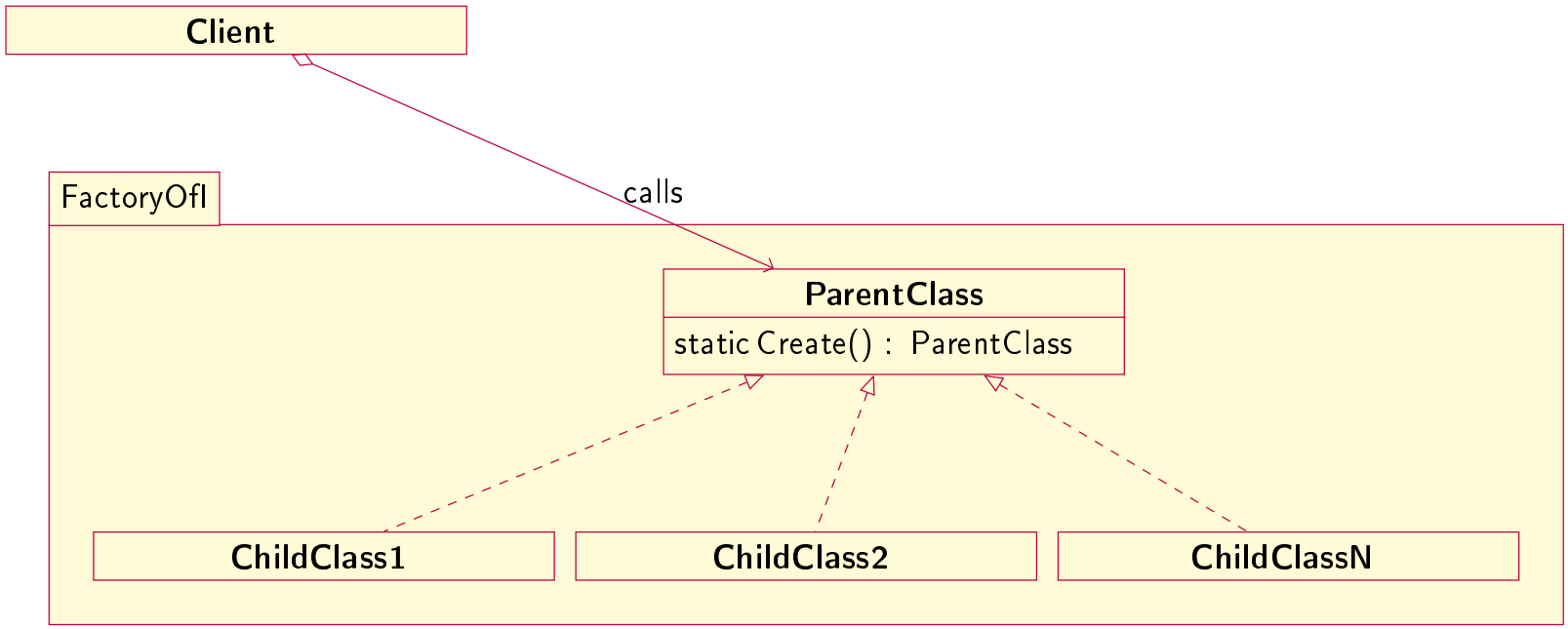
The simple factory design pattern

The simple factory design pattern

Formalization

- The solution provided for our `Animal` scenario belongs to this pattern
- A simple factory is a method that is called directly from the client
- Such method returns one of many different polymorphic classes
- Such method can be declared in the parent class (as `static`) or in a separate class
- In the following a UML of such pattern and an example are provided

The simple factory design pattern



The simple factory design pattern


```
1  abstract class Vehicle {
2    public static Vehicle Create(int id) {
3      if((id == 1)) {
4        return new Ferrari();
5      }
6      if((id == 2)) {
7        return new Lamborghini();
8      }
9    }
10   public abstract  void StartEngine();
11 }
12 ...
13 Vehicle a_vehicle = Vehicle.Create(Int32.Parse(Console.ReadLine()));
14 a_vehicle.StartEngine();
```

The simple factory design pattern

Which in Java then becomes:

```
1  abstract class Vehicle {
2    public static Vehicle Create(int id) {
3      if((id == 1)) {
4        return new Ferrari();
5      }
6      if((id == 2)) {
7        return new Lamborghini();
8      }
9    }
10   public abstract  void StartEngine();
11 }
12 ...
13 Vehicle a_vehicle = Vehicle.Create(Integer.parseInt(new Scanner(System.in).
       nextLine()));
14 a_vehicle.StartEngine();
```

The simple factory design pattern


```
1  class Ferrari : Vehicle {
2    public Ferrari() : base() {
3    }
4    public override void StartEngine() {
5      ...
6    }
7  }
8  class Lamborghini : Vehicle {
9    public Lamborghini() : base() {
10   }
11   public override void StartEngine() {
12     ...
13   }
14 }
```

The simple factory design pattern

Which in Java then becomes:

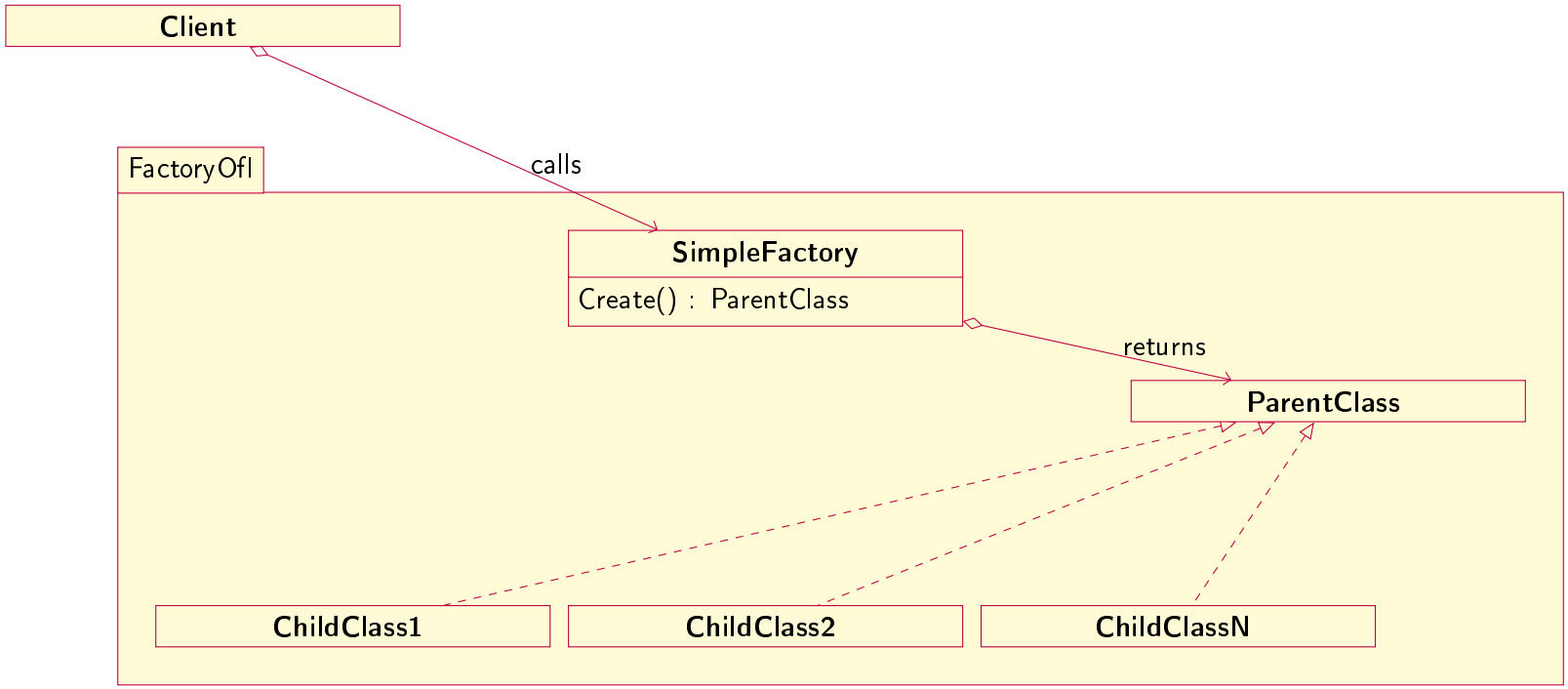
```
1  class Ferrari extends Vehicle {
2    public Ferrari() {
3      super();
4    }
5    public void StartEngine() {
6  ...
7    }
8  }
9  class Lamborghini extends Vehicle {
10   public Lamborghini() {
11     super();
12   }
13   public void StartEngine() {
14 ...
15   }
16 }
```

The simple factory design pattern

Formalization

- The `Create` method can be declared as part of a distinct factory class
- In this case an instance of such factory is necessary to call the method, unless the method is `static`
- In the following, UML of this pattern and an example are provided

The simple factory design pattern



The simple factory design pattern


```
1  interface Vehicle {
2    void StartEngine();
3  }
4  class VehicleFactory {
5    public Vehicle Create(int id) {
6      if((id == 1)) {
7        return new Ferrari();
8      }
9      if((id == 2)) {
10       return new Lamborghini();
11     }
12   }
13 }
14 ...
15 Vehicle a_vehicle = VehicleFactory.Create(Int32.Parse(Console.ReadLine()));
16 a_vehicle.StartEngine();
```

The simple factory design pattern

Which in Java then becomes:

```
1  interface Vehicle {
2    void StartEngine();
3  }
4  class VehicleFactory {
5    public Vehicle Create(int id) {
6      if((id == 1)) {
7        return new Ferrari();
8      }
9      if((id == 2)) {
10       return new Lamborghini();
11     }
12   }
13 }
14 ...
15 Vehicle a_vehicle = VehicleFactory.Create(Integer.parseInt(new Scanner(
       System.in).nextLine()));
16 a_vehicle.StartEngine();
```

The simple factory design pattern


```
1  class Ferrari : Vehicle {
2    public Ferrari() {
3    }
4    public override void StartEngine() {
5      ...
6    }
7  }
8  class Lamborghini : Vehicle {
9    public Lamborghini() {
10   }
11   public override void StartEngine() {
12     ...
13   }
14 }
```

The simple factory design pattern

Which in Java then becomes:

```
1  class Ferrari extends Vehicle {
2    public Ferrari() {
3    }
4    public void StartEngine() {
5  ...
6    }
7  }
8  class Lamborghini extends Vehicle {
9    public Lamborghini() {
10   }
11   public void StartEngine() {
12 ...
13   }
14 }
```

The simple factory design pattern

Static methods are not enough

- This method is not flexible
- We cannot redefine (part of) our factories
- We cannot to make use of polymorphism here as well

The simple factory design pattern

Static methods are not enough

- A solution would be that our simple factory method becomes virtual
- Depending on the domain, a "concrete factory" is then selected by the client that implements such virtual methods
- This mechanism of *interchangeable* factories is called the **factory method**

The factory method

The factory method

Formalization

- A factory method is: "a class which defers instantiation of an object to subclasses"[a]
- How do we achieve this? By means of polymorphism
- We make our factory polymorphic, so the instantiation becomes polymorphic, as well

[a] GOF

The factory method

Formalization

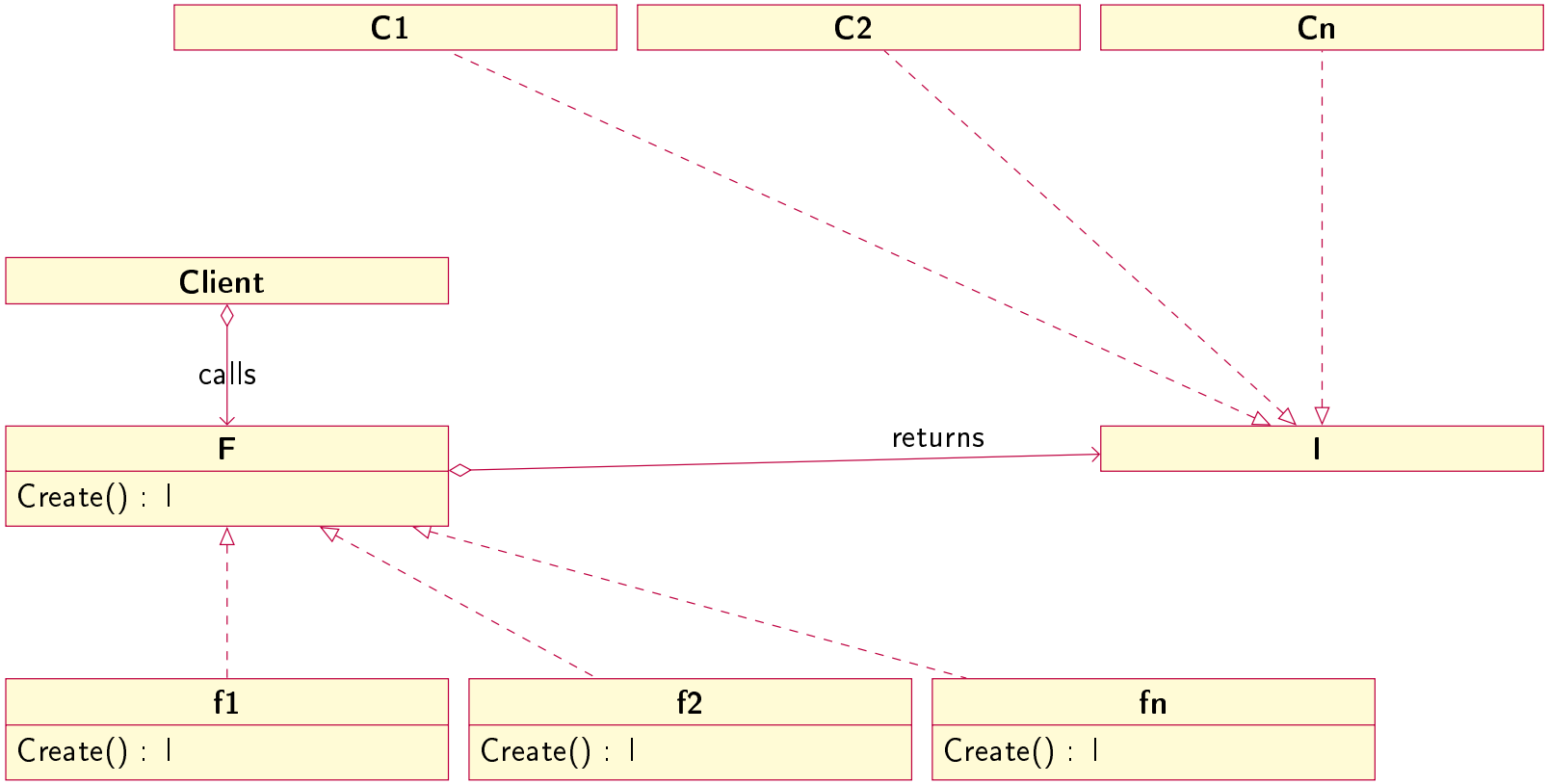
- Given a polymorphic type I (to instantiate)
- Given a series of concrete implementations of I: $C_1, \ldots, C_n$
- Factory implementation:
  - Given a polymorphic factory F_I that creates an I
  - Given a series of concrete implementations of F_I: $f_1, \ldots, f_m$

The factory method

Formalization

- By deferring instantiation of an object to subclasses a new client that has different criteria for instantiating concrete I's will provide a different concrete factory without changing the already existing relations
- Exchanging concrete factories does not affect other classes, structures, or behaviors
- In the following UML of this pattern and an example are provided

The factory method



The factory method


```
1  abstract class VehicleFactory {
2    public abstract  Vehicle Create(int id,Color color);
3  }
4  class ConcreteVehicleFactory : VehicleFactory {
5    public Vehicle Create(int id,Color color) {
6      if((id == 1)) {
7        return new Ferrari(color);
8      }
9      if((id == 2)) {
10       return new Lamborghini(color);
11     }
12   }
13 }
14 ...
15 Vehicle a_vehicle = ConcreteVehicleFactory.Create(Int32.Parse(Console.
       ReadLine());,"Color.Red");
16 a_vehicle.StartEngine();
```

The factory method

Which in Java then becomes:

```
1  abstract class VehicleFactory {
2    public abstract  Vehicle Create(int id,Color color);
3  }
4  class ConcreteVehicleFactory implements VehicleFactory {
5    public Vehicle Create(int id,Color color) {
6      if((id == 1)) {
7        return new Ferrari(color);
8      }
9      if((id == 2)) {
10       return new Lamborghini(color);
11     }
12   }
13 }
14 ...
15 Vehicle a_vehicle = ConcreteVehicleFactory.Create(Integer.parseInt(new
       Scanner(System.in).nextLine());,"Color.Red");
16 a_vehicle.StartEngine();
```

The factory method

Fixed attribute and factories

- Factories can also specialize in “cross-type” concerns
- For example, a fixed attribute

The factory method


```
1  abstract class VehicleFactory {
2    public abstract  Vehicle Create(int id);
3  }
4  class RedVehicleFactory : VehicleFactory {
5    public Vehicle Create(int id) {
6      if((id == 1)) {
7        return new Ferrari("Red");
8      }
9      if((id == 2)) {
10       return new Lamborghini("Red");
11     }
12   }
13 }
14 class YellowVehicleFactory : VehicleFactory {
15   public Vehicle Create(int id) {
16     if((id == 1)) {
17       return new Ferrari("Yellow");
18     }
19     if((id == 2)) {
20       return new Lamborghini("Yellow");
21     }
22   }
23 }
24 ...
25 Vehicle a_vehicle = YellowVehicleFactory.Create(Int32.Parse(Console.ReadLine
       ()));
26 a_vehicle.StartEngine();
```

The factory method

Which in Java then becomes:

```
1  abstract class VehicleFactory {
2    public abstract  Vehicle Create(int id);
3  }
4  class RedVehicleFactory implements VehicleFactory {
5    public Vehicle Create(int id) {
6      if((id == 1)) {
7        return new Ferrari("Red");
8      }
9      if((id == 2)) {
10       return new Lamborghini("Red");
11     }
12   }
13 }
14 class YellowVehicleFactory implements VehicleFactory {
15   public Vehicle Create(int id) {
16     if((id == 1)) {
17       return new Ferrari("Yellow");
18     }
19     if((id == 2)) {
20       return new Lamborghini("Yellow");
21     }
22   }
23 }
24 ...
25 Vehicle a_vehicle = YellowVehicleFactory.Create(Integer.parseInt(new Scanner
       (System.in).nextLine()));
26 a_vehicle.StartEngine();
```

The factory method


```
1  interface Vehicle {
2    void StartEngine();
3  }
4  class Ferrari : Vehicle {
5    public Ferrari(color Color) {
6    }
7    public override void StartEngine() {
8      ...
9    }
10 }
11 class Lamborghini : Vehicle {
12   public Lamborghini(color Color) {
13   }
14   public override void StartEngine() {
15     ...
16   }
17 }
```

The factory method

Which in Java then becomes:

```
1  interface Vehicle {
2    void StartEngine();
3  }
4  class Ferrari extends Vehicle {
5    public Ferrari(color Color) {
6    }
7    public void StartEngine() {
8  ...
9    }
10 }
11 class Lamborghini extends Vehicle {
12   public Lamborghini(color Color) {
13   }
14   public void StartEngine() {
15 ...
16   }
17 }
```

Conclusions

Conclusions

Conclusions

- Sometime we need interfaces to implement virtual constructors
- Why? Because sometimes we know the polymorphic type to instantiate first and later the concrete one
- A naive solution would see the client code implement such instantiation mechanism, but this yields repetition and makes code less maintainable
- Factories solve this issue elegantly by promoting polymorphic constructors, by means of class polymorphism, or static methods

Conclusions

Conclusions

- Static methods are less flexible when compared to polymorphic classes, since abstract classes allow both virtual and non virtual methods
- Moreover, polymorphic classes allow the definition of multiple interchangeable concrete factories, each shaped for a specific domain

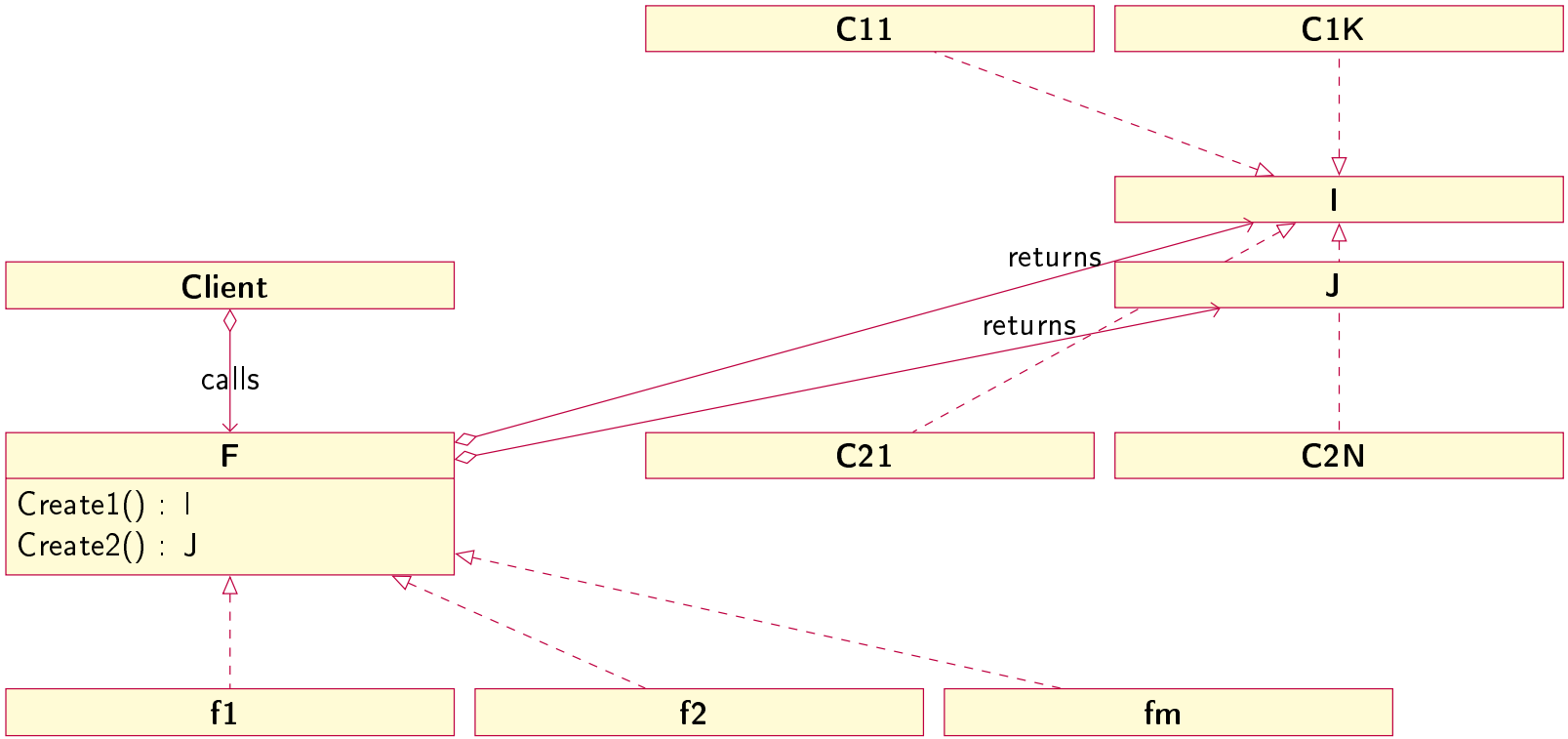
Appendix

Appendix

The abstract factory method - formalization

- The biggest pattern of the factories seen so far
- Is acts the same as the factory method, except for the fact that it might contain more than one virtual instantiation method
- Each of them returning a different but “related” polymorphic object
- In the following a UML of such pattern and an example are provided

Appendix



Appendix


```
1  abstract class VehicleComponentsFactory {
2    public abstract  Tire CreateTire();
3    public abstract  Seat CreateSeat();
4    ...
5  }
6  class FerraryComponents : VehicleComponentsFactory {
7    ...
8    public Seat CreateSeat() {
9      ...
10   }
11   public Tire CreateTire() {
12     ...
13   }
14 }
15 class LamborghiniComponents : VehicleComponentsFactory {
16   ...
17   public Seat CreateSeat() {
18     ...
19   }
20   public Tire CreateTire() {
21     ...
22   }
23 }
24 ...
```

Appendix

Which in Java then becomes:

```
1  abstract class VehicleComponentsFactory {
2    public abstract  Tire CreateTire();
3    public abstract  Seat CreateSeat();
4  ...
5  }
6  class FerraryComponents implements VehicleComponentsFactory {
7  ...
8    public Seat CreateSeat() {
9  ...
10   }
11   public Tire CreateTire() {
12 ...
13   }
14 }
15 class LamborghiniComponents implements VehicleComponentsFactory {
16 ...
17   public Seat CreateSeat() {
18 ...
19   }
20   public Tire CreateTire() {
21 ...
22   }
23 }
24 ...
```

Appendix


```
1  class Garage {
2    public VehicleComponentsFactory ferrariComponentsFactory;
3    public VehicleComponentsFactory lamborghiniComponentsFactory;
4    public Garage() {
5      this.ferrariComponentsFactory = new FerraryComponents();
6      this.lamborghiniComponentsFactory = new LamborghiniComponents();
7    }
8    public void RepearTires(vehicle Vehicle) {
9      vehicle.Visit(ferrari => ferrari.Tire   = ferrariComponentsFactory.
           CreateTire(),lamborghini => ferrari.lamborghini   =
           lamborghiniComponentsFactory.CreateTire());
10   }
11 }
12 ...
```

Appendix

Which in Java then becomes:

```
1  class Garage {
2    public VehicleComponentsFactory ferrariComponentsFactory;
3    public VehicleComponentsFactory lamborghiniComponentsFactory;
4    public Garage() {
5      this.ferrariComponentsFactory = new FerraryComponents();
6      this.lamborghiniComponentsFactory = new LamborghiniComponents();
7    }
8    public void RepearTires(vehicle Vehicle) {
9      vehicle.Visit(Error: unsupported lambda functions in Java pretty printer
           ,Error: unsupported lambda functions in Java pretty printer);
10   }
11 }
12 ...
```

This is it!

The best of luck, and thanks for the attention!