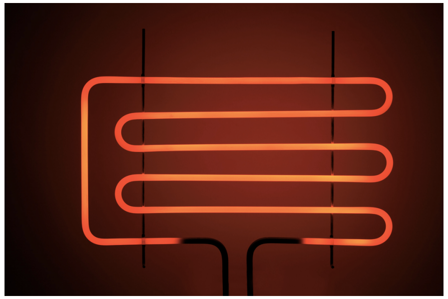
ADDITIONAL MATERIAL FOR INFSEN01-2

DR ANNA V. KONONOVA

LECTURE 5. ABSTRACT FACTORY AND FACTORY METHOD. MAY 30, 2017

- Reminder: **Read all official slides as well** (your exam is prepared based on them)
- Photo opportunity!
- Last week, we have revisited the “Apples and Bears” example, studied Adapter pattern and applied it to the ”heating element” example
- **Today** we will: quickly revise last lesson’s example, look at abstract factory and factory method patterns and work on a large new example problem.



ADAPTER: EXAMPLE

"**Heating element**". Our department makes software for remote house climate control. It uses *commercial* modelling package which calculates the *current expected time* (in seconds) required for heating up the room up to a given temperature (based on model parameters, i.e. some physical properties of the house and the heating system).

Our software is so streamlined that the investors decided to reuse it in two other departments: one dealing with the same systems for American market and another working with temperature control units for science labs. American customers are used to the Fahrenheit scale and scientific labs use the Kelvin scale (we are used to the Celsius scale).

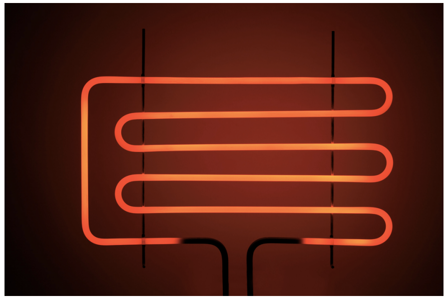
- *Assume some modelling package that for input tempC returns the value tempC*10 seconds (black box)*
- Write remote house climate control application, simulate some calls to the modelling interface (0°C, 10°C, 23°C)
- Extend the interface of the heating element controller for easy reuse (via Adapter pattern)
- Remember to use only Celsius inside the interface, and only equipment scale outside the interface
- **Write a short program simulating calls to new interface from all three types of equipment and prinint model output (try inputting 36.8°C, 98.24°F, 309.95K)**

(If you have problems imagining different scales think of how height can be given in meters, feet, etc.)

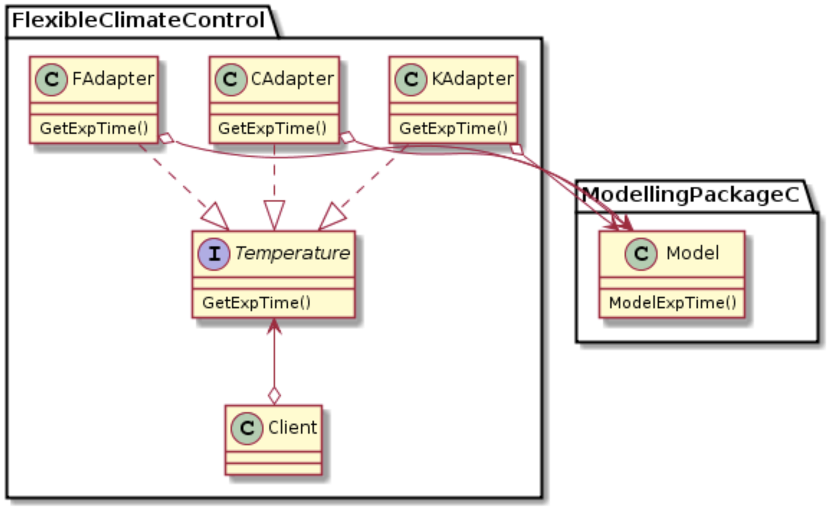
tC = **tK** - 273.15 **tF** = (**tC** * 9/5 + 32) **tK** = (**tF** + 459.67) * 5/9

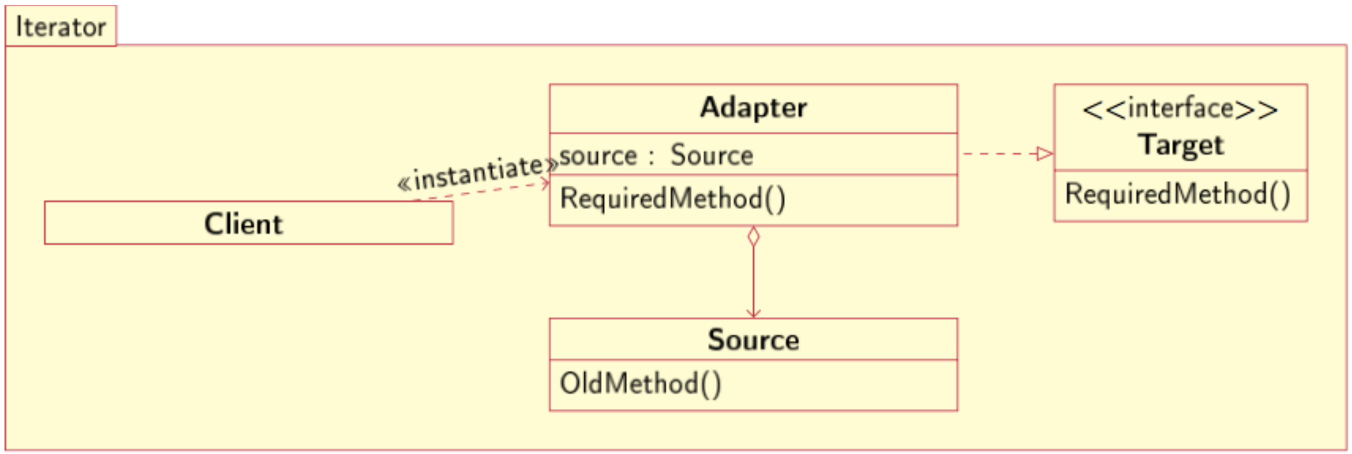
tK = **tC** + 273.15 **tC** = (**tF** - 32) * 5/9 **tF** = **tK** * 9/5 - 459.67

Water freezes: 0°C = 32°F = 273.15K
Weather today: 18°C = 64.4°F = 291.15K
Normal human: 36.8°C = 98.24°F = 309.95K
Water boils: 99.98°C = 211.96°F = 373.13K



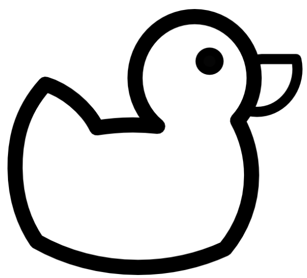
ADAPTER: COMPARE EXAMPLE DESIGN





ABSTRACT CLASS (OFFICIAL SLIDES)

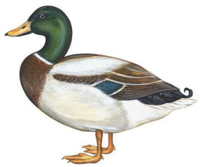
- Interfaces: fully abstract; classes: fully concrete. We want something in between ➢
- In OOP special classes are allowed which contain some methods with the body and some without ➢ abstract classes
- It is not possible to instantiate them directly
- They have to be inherited to use their functionalities
- All abstract methods must eventually get implementation
- Abstract classes allow to combine polymorphism with concrete implementations



ABSTRACT CLASS PATTERN

- “*Depend on abstractions. Do not depend on concrete classes*” (OO principle)

Why is this important?



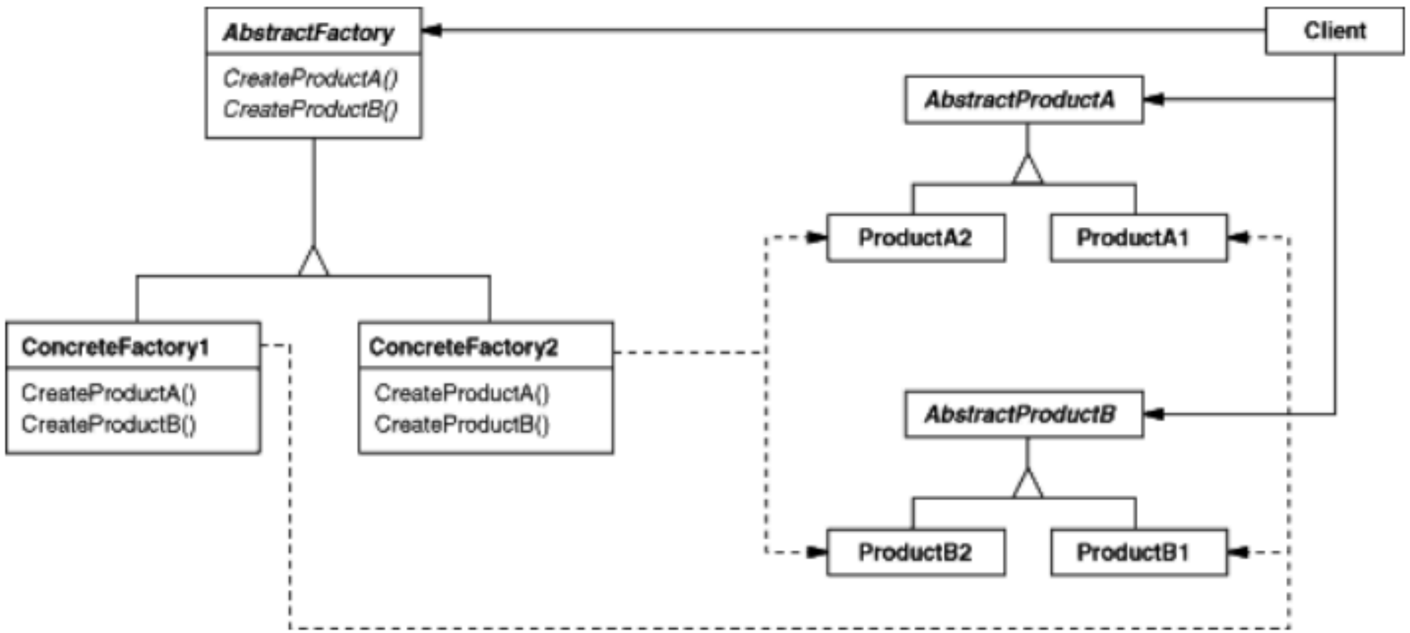
```
Duck duck;
duck = new MallardDuck();
duck = new RedHeadDuck();
duck = new DecoyDuck(); // ducks will be spread all over the code
```

- Code is not closed for modification! ➤ need to encapsulate what varies! ➤ how?
- ➤ A handy design pattern called Abstract Class pattern
- **Intent**: provide an interface for creating families of related or dependent objects without specifying their concrete classes
- **Applicability**:

- A system should be independent on how its products are created, composed and represented
- A system is configured with one of multiple families of products
- A family of related product objects is designed to be used together
- A class library of products is required and their implementation needs to stay hidden

ABSTRACT FACTORY PATTERN: PARTICIPANTS, STRUCTURE

- **AbstractFactory** – declares an interface for operations that create abstract product objects
- **ConcreteFactory** – implements the operations to create concrete product objects
- **AbstractProduct** – declares an interface for a type of product object
- **ConcreteProduct** – defines a product object to be created by the corresponding concrete factory; implements the AbstractProduct interface
- **Client** – uses only interfaces declared by AbstractFactory and AbstractProduct classes



Consequences:

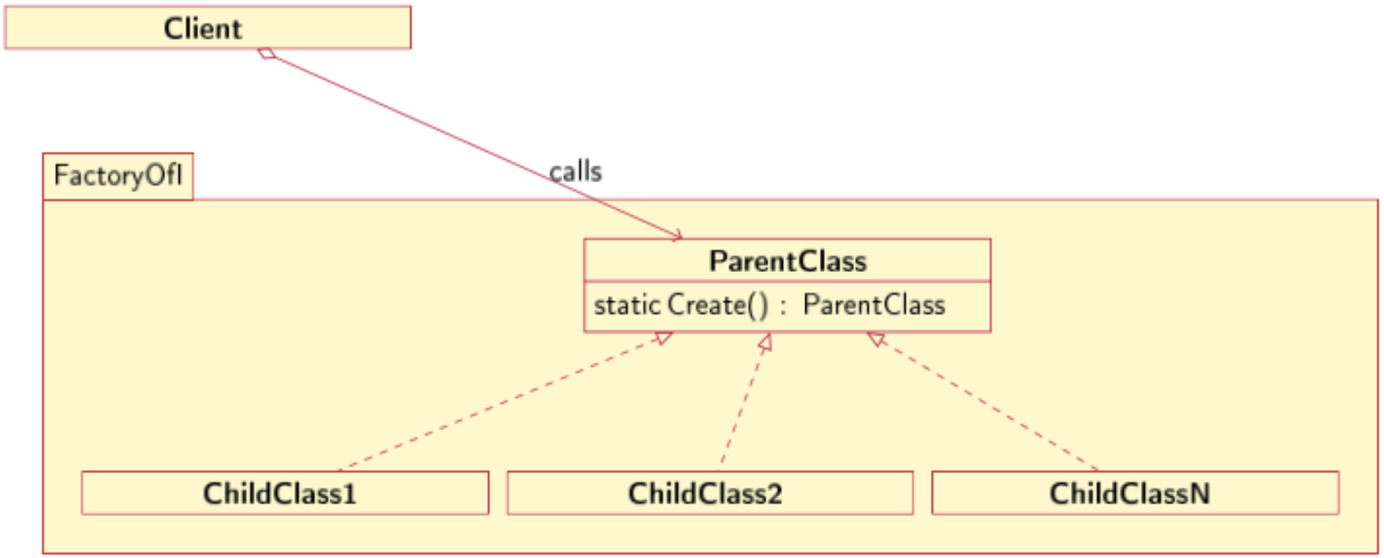
- Isolates concrete class: clients manipulate instances through their abstract interfaces
- Makes exchanging product families easy: since the class of concrete factory appears only during instantiation ➢ easy to change!
- Promotes/enforces consistency among products
- Makes supporting new kinds of products difficult: AbstractFactory interface fixes the set of products that can be created ➢ changes in AbstractFactory and all its subclasses ☹

Implementation:

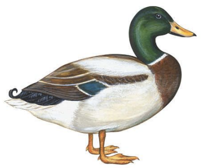
- AbstractFactory only declares an interface; it is concrete subclasses that actually create products. The most common way to do this is to define a *factory method* for each product
- AbstractFactory usually defines operations for each kind of the product it produces. Thus adding new kind of product is even more tricky ➢ more flexible (but *less* safe) design is to add a parameter to operations that create objects to specify the kind ➢ still single make operation with a parameter indicating which object to create (well-suited for dynamically-typed languages) ➢ careful if client needs to perform subclass-specific operations that are not accessible via abstract interface (cant differentiate product's class)

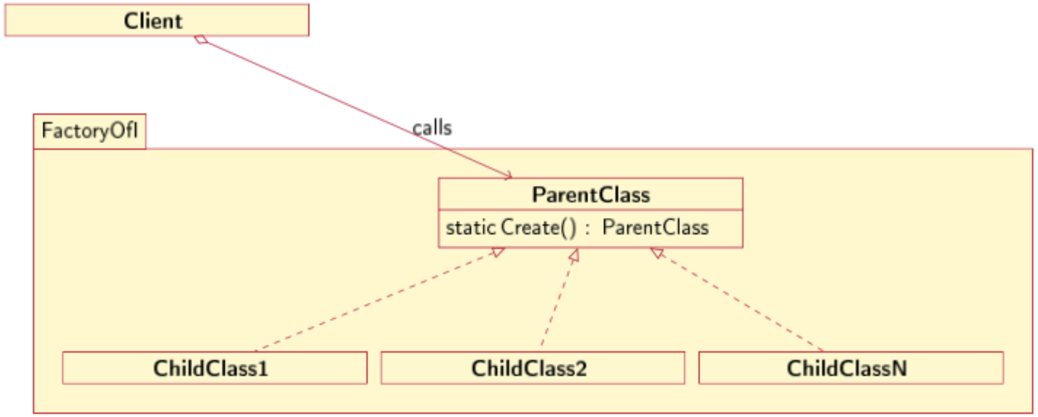
SIMPLE FACTORY: OFFICIAL SLIDES

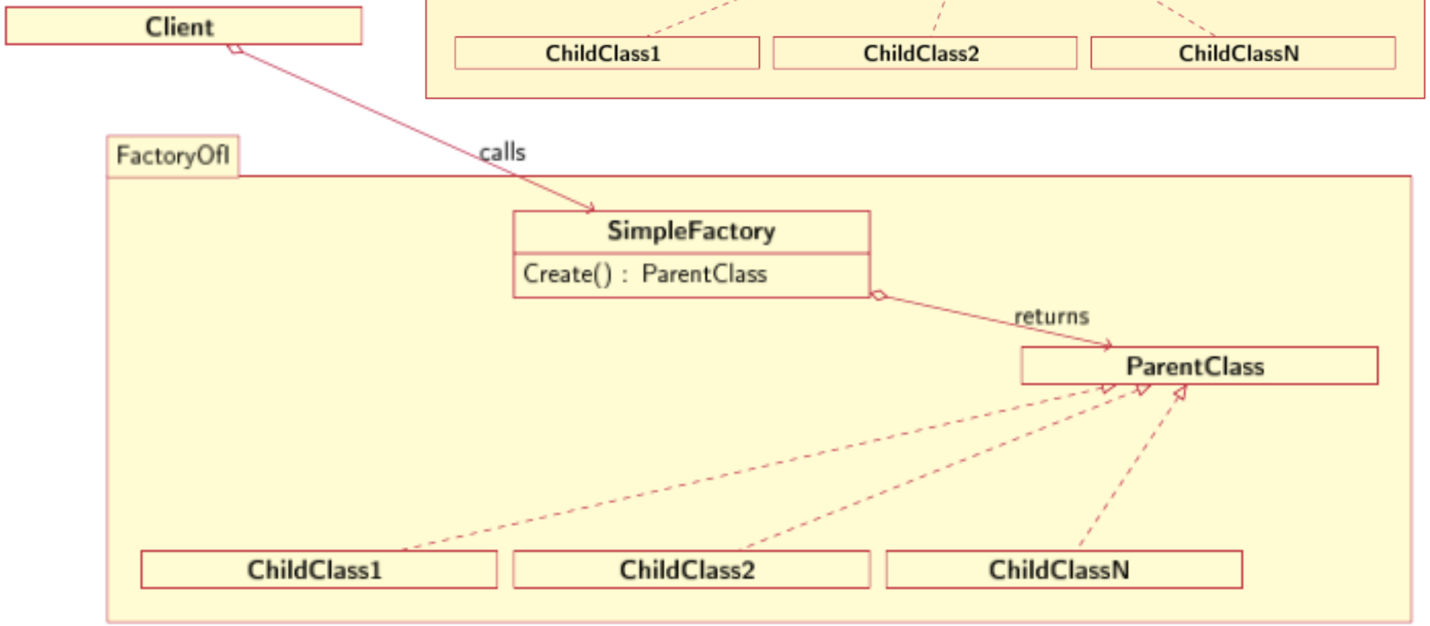
- Patterns providing instantiation mechanisms are generally referred to as factory design patterns
- Factory create method is called directly by the client; it is declared in the parent class (as static) or in a separate class; it returns one of many polymorphic classes;



SIMPLE FACTORY: OFFICIAL SLIDES







FACTORY METHOD PATTERN

- **Intent**: define an interface for creating an object but the subclasses decide which class to instantiate
- **Applicability:**

- A class can't anticipate the class of objects it must create
- A class wants its subclasses to specify the objects it creates
- Classes delegate responsibility to one of several helper subclasses and knowledge needs to be localised the knowledge of which helper subclass is the delegate

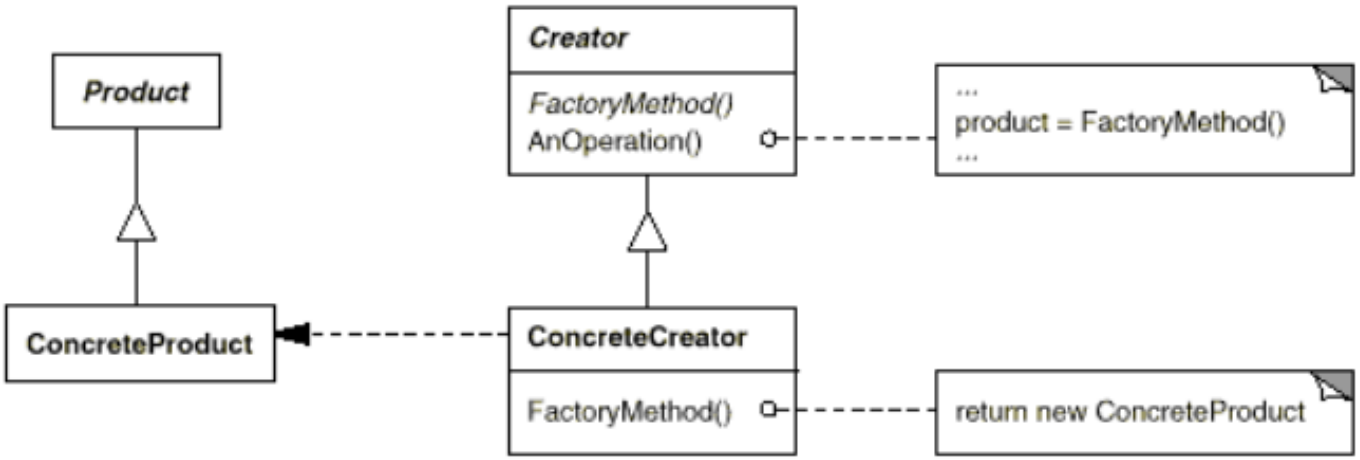
Remember Abstract Factory pattern:

- ***Intent****: provide an interface for creating families of related or dependent objects without specifying their concrete classes*
- ***Applicability:***

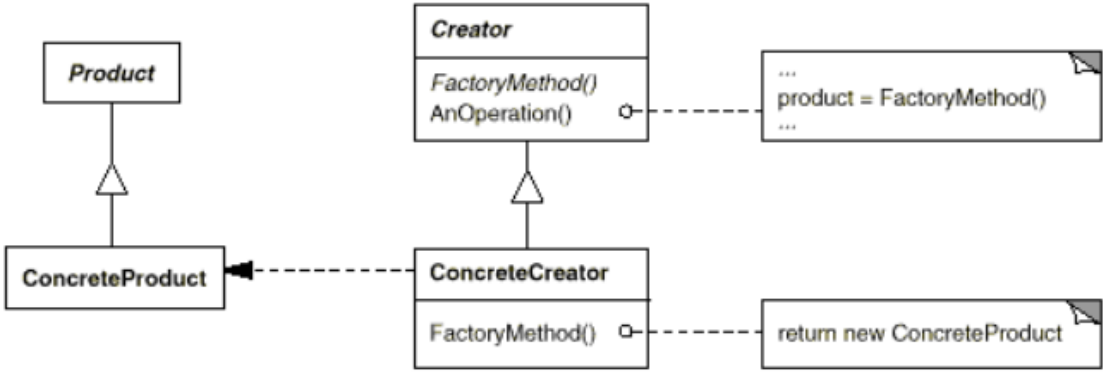
- *A system should be independent on how its products are created, composed and represented*
- *A system is configured with one of multiple families of products*
- *A family of related product objects is designed to be used together*
- *A class library of products is required and their implementation needs to stay hidden*

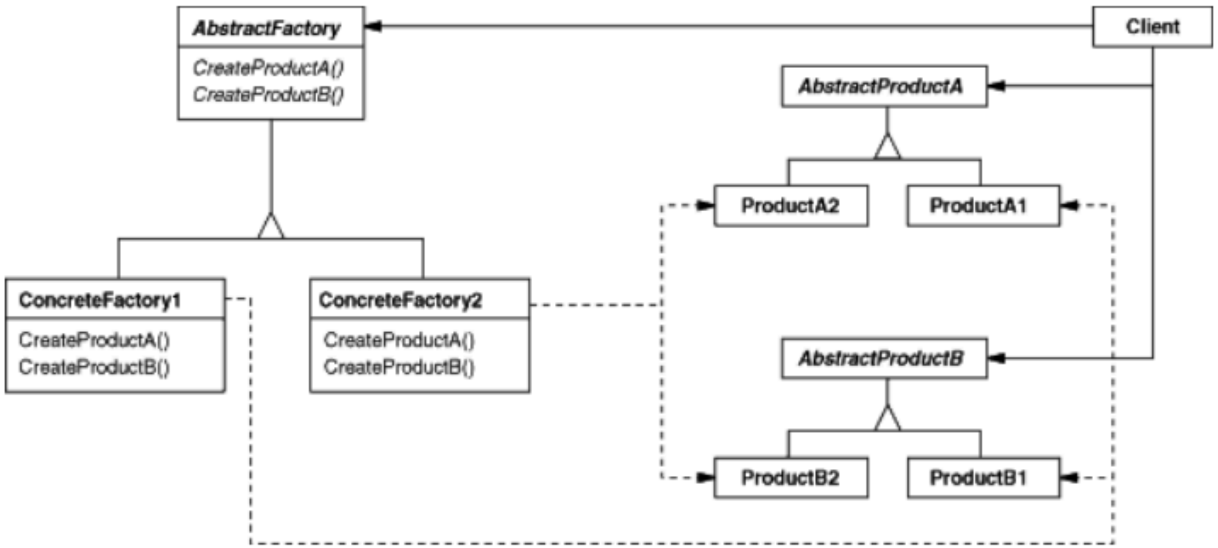
FACTORY METHOD PATTERN: PARTICIPANTS, STRUCTURE

- **Product** – defines the interface of objects the factory method creates
- **ConcreteProduct** – implements the Product interface
- **Creator** - declares the factory method which returns an object of type Product; may call the factory method to create a Product object
- **ConcreteCreator** – overrides the factory metho to return an instance of a ConcreteProduct



FACTORY METHOD AND ABSTRACT FACTORY PATTERNS



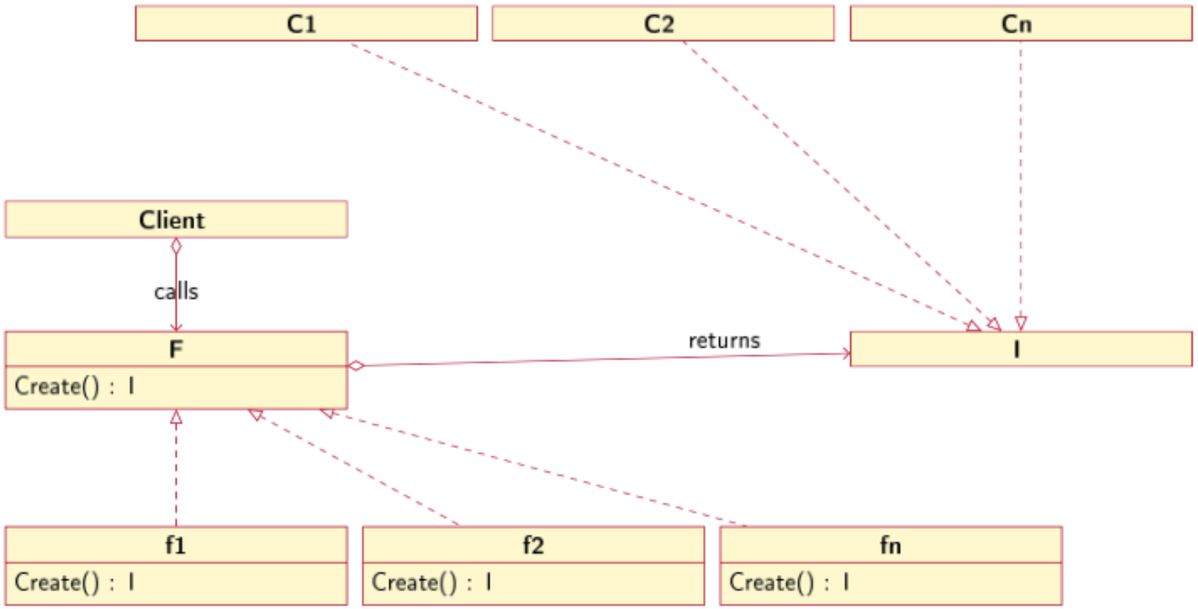


Consequences:

- Eliminated the need to bind application-specific classes into the code
- Clients might need to subclass the Creator class just to create a particular ConcreteProduct object
- Creating objects inside a class with factory method is always more flexible than creating an object directly
- Connects parallel hierarchies: if a class delegates some of its responsibility to a separate class ➢clients can also call corresponding factory methods

FACTORY METHOD PATTERN: OFFICIAL SLIDES

- A factory method: a class that defers instantiation of an object to a subclass
- Via polymorphism ➢ polymorphic factory ➢ polymorphic instantiation
- Given a polymorphic type *I* and a series of concrete implementations of *I*: C_l, …, C_n, Factory implementation is polymorphic factory F_l that creates *I* given concrete implementations of F_l: f_l,…, f_m
- 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 classes does not affect other classes, structures or behaviours



EXAMPLE: SANTA'S WORKSHOP

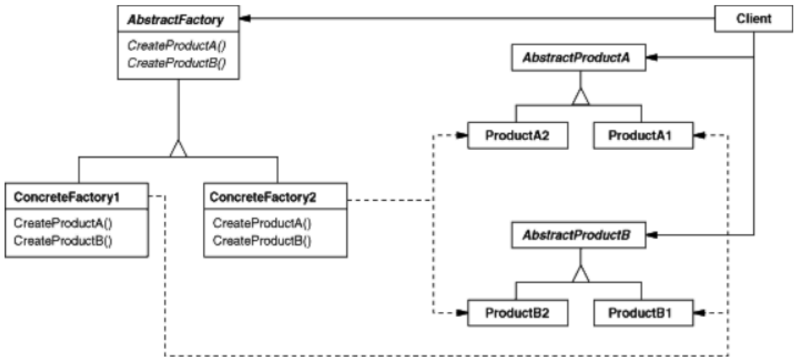
- "**Santa's Workshop**": due to a rapid growth of the Earth's population, Santa Claus had to work multiple shifts. To reduce his workload, an extra Santa has been hired. Unfortunately only Bad Santa was available. Both Santas prepare two kinds of presents: one for well-behaved kids and one for the less-well-behaved kids. However, clearly their presents differ.. a little
- *Good Santa* chooses to give every *well-behaved* kid a hand-made toy of their choice. *Less-well-behaved* kids receive a freshly printed "How to be a good child" book.
- *Bad Santa* prepares presents for *well-behaved* kids packing random parts of various Lego sets. Meanwhile the *less-well-behaved* kids receive his real autograph
- Both Santas work simultaneously and choice of a Santa is done runtime based on each Santa's availability

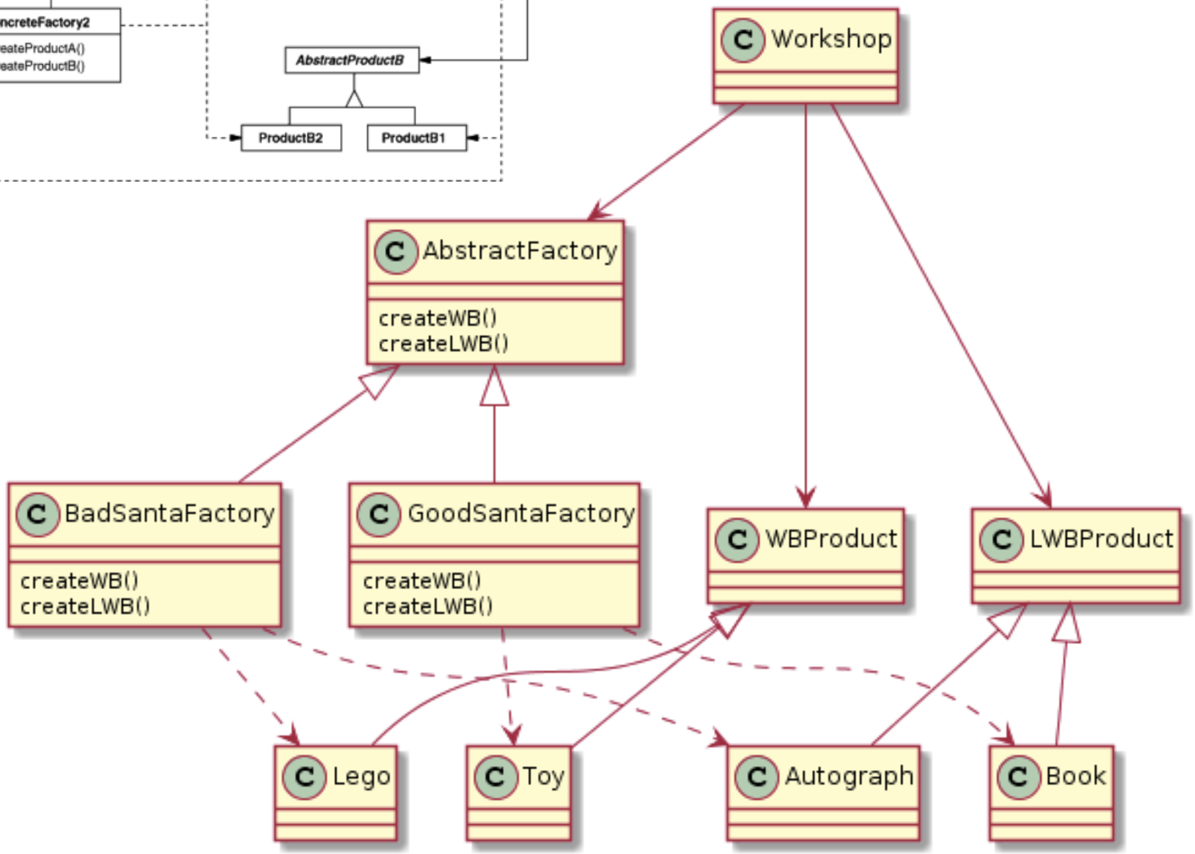
➢ design and draw the class diagram to implement the application running the joint Santa's workshop

➢ run the workshop with requests for 5 kids: kids 1,2, 4 are well-behaved and kids 3, 5 are less-well-behaved. You can choose which Santa is available for each kid

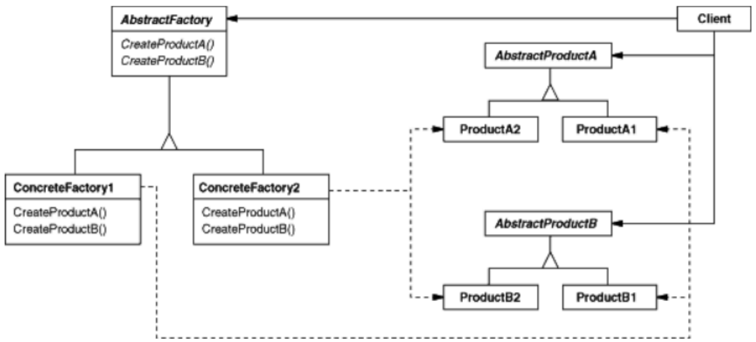
* several designs are possible

EXAMPLE: SANTA’S WORKSHOP





EXAMPLE: SANTA'S WORKSHOP

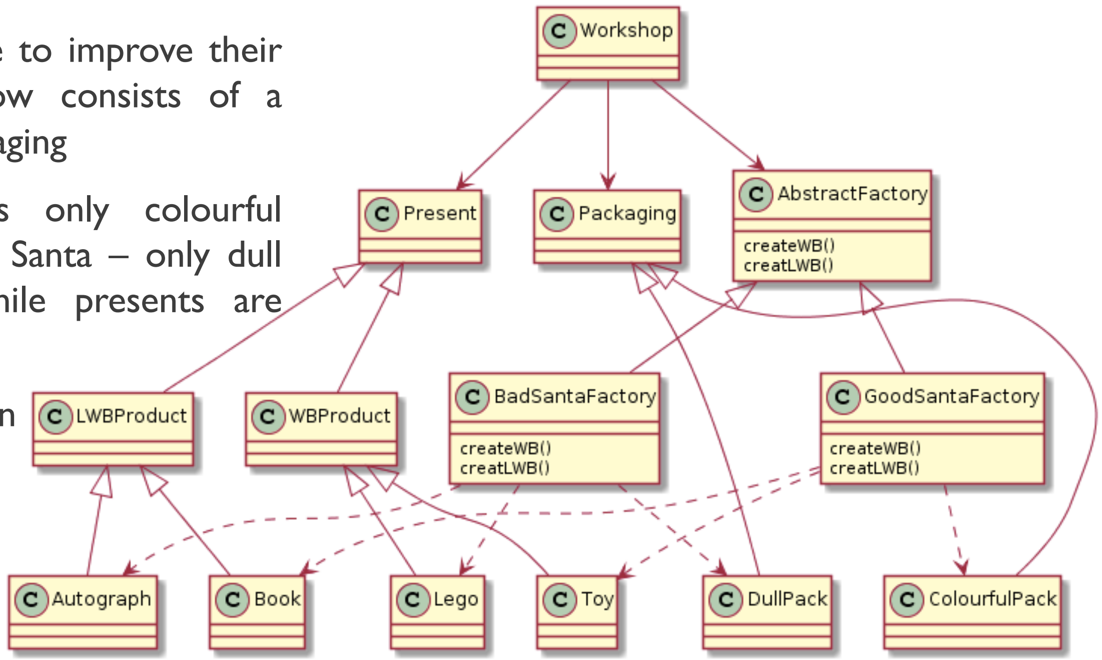


Both Santas decide to improve their gifts: each gift now consists of a present and a packaging

Good Santa uses only colourful packaging and Bad Santa – only dull packaging. Meanwhile presents are chosen as before

➢redraw the design

* several designs are possible



CONCLUSION

- **Today** we have revised last lesson's example, studied two new design patterns and applied them to a new example problem
- **You should be half way through with your assignment by now**

✓ Lesson 1: Introduction

✓ Lesson 2: Visitor pattern

✓ Lesson 3: Iterator pattern

✓ Lesson 4: Adapter pattern

✓ Lesson 5: Abstract factory, Factory Method

⏳ Lesson 6: Decorator pattern

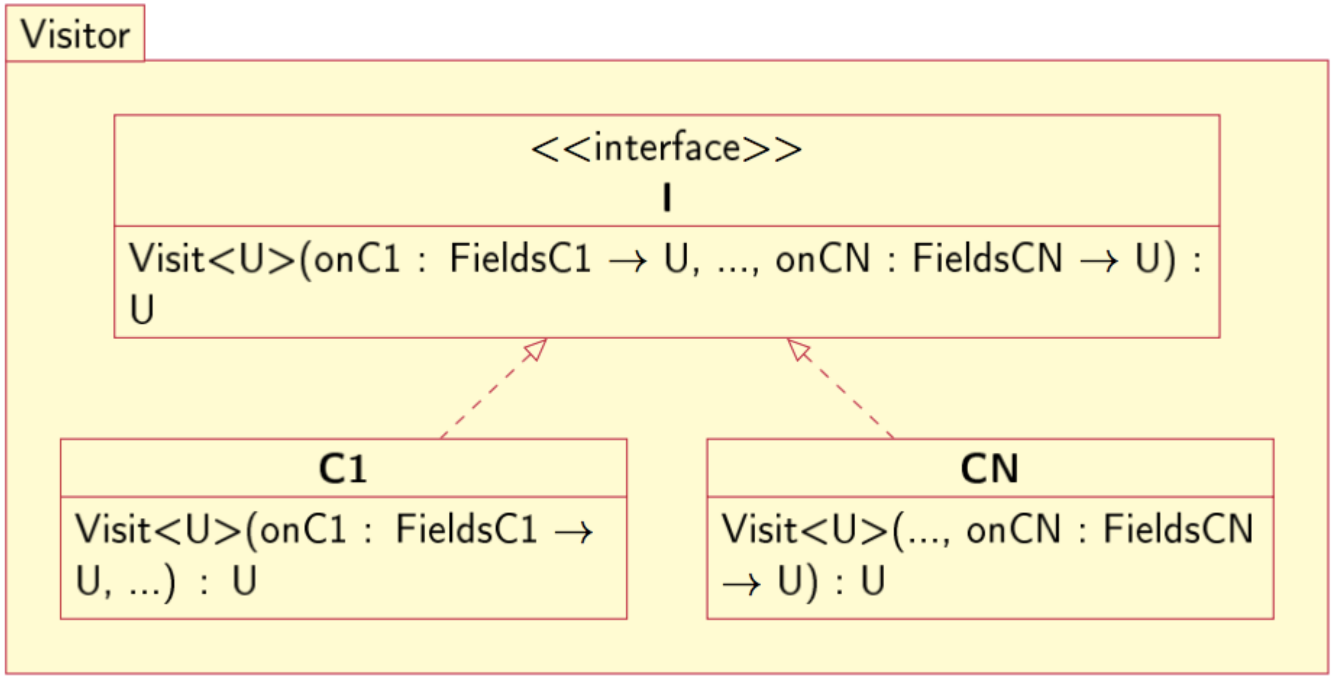
⏳ Lesson 7: Revision (what needs to be revised? Look at the example exams and ask questions; so far I only see the need to explain heap, stack, declarations)

⏳ Lesson 8: Mock exam

- Thanks for your attention!

λ version of the Visitor

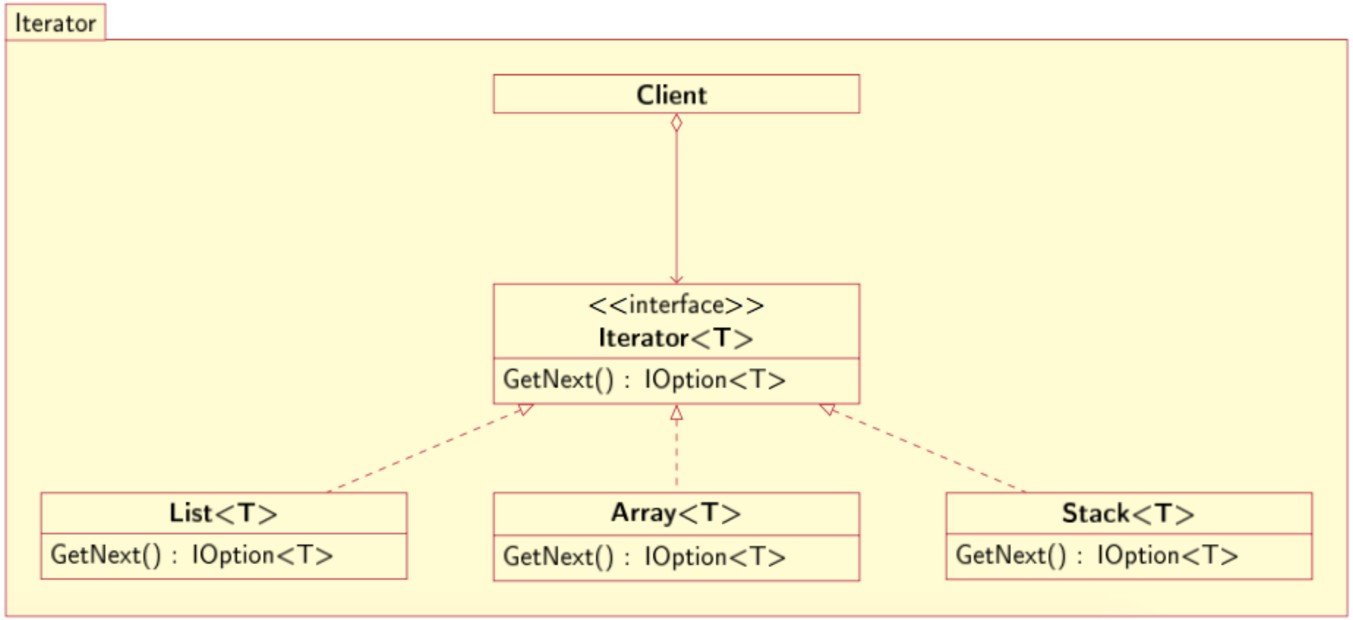
Intent: represent an operator to be performed on the elements of an object structure. Visitor allows a definition of a new operation without changing the classes of the elements on which it operates



REVISING λ NOTATION
ITERATOR: OFFICIAL SLIDES

λ version of the Iterator

Intent: a way to access elements of an aggregate object without exposing its underlying representation

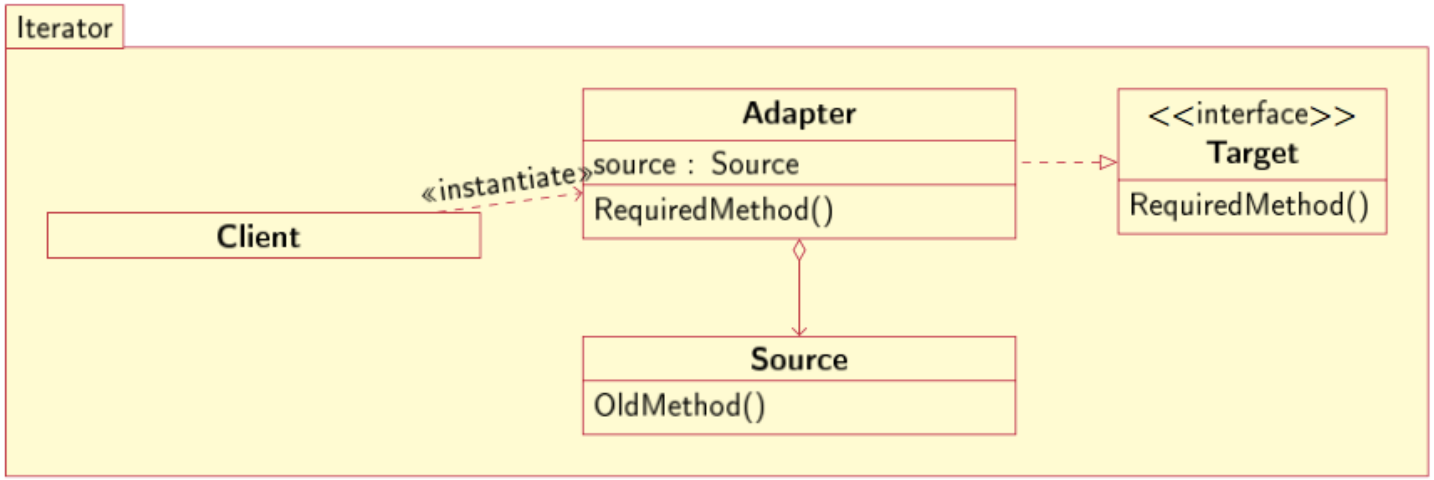


REVISING λ NOTATION
ADAPTER: OFFICIAL SLIDES

λ version of the Adapter

Intent: to convert the interface of a class into another interface that is expected by a client

(typo in the diagram below: title should read Adapter)



ABSTRACT FACTORY AND FACTORY METHOD PATTERNS: OFFICIAL SLIDES