Design Patterns & Software Architecture

Factory Method & Abstract Factory

dr. Joost Schalken-Pinkster

Windesheim University of Applied Science

The Netherlands

The contents of these course slides is (in great part) based on:
Chris Loftus, *Course on Design Patterns & Software Architecture for NEU*. Aberystwyth University, 2013.
Jeroen Weber & Christian Köppe, *Course on Patterns and Frameworks*. Hogeschool Utrecht, 2013.
Leo Pruijt, *Course on Software Architecture*. Hogeschool Utrecht, 2010-2013.

Session overview

- Factory Method
- Abstract Factory

Factory Method

Let’s find a design pattern

Will now present, on the board, and using Eclipse, a solution that utilizes a creational design pattern.

Case: Meat producing plant
Initial requirements

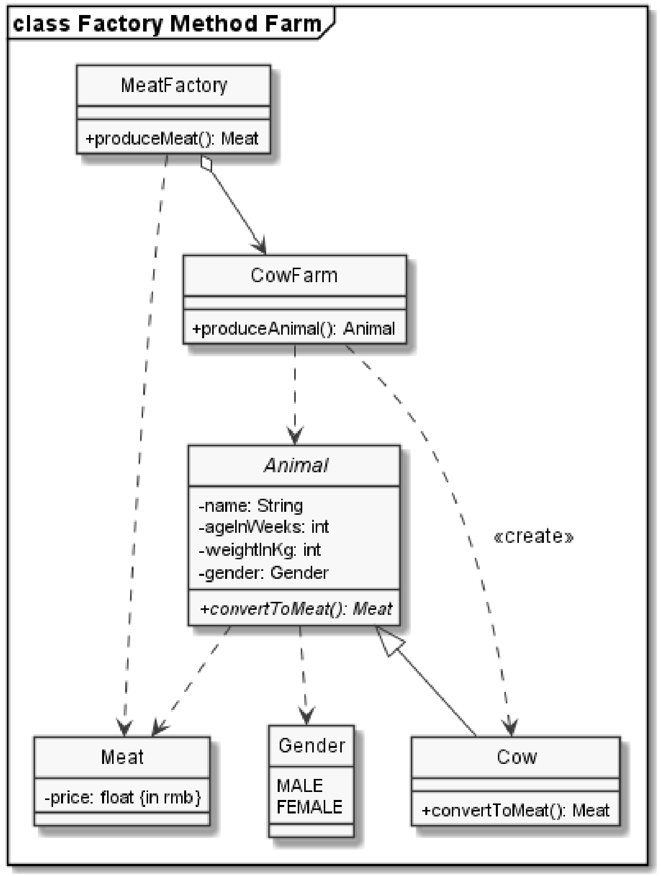
A software house decides to develop an application suite for a meat producing company.

The application has the following requirements:

- The application knows the class Animal and the subclass Cow. The class Animal has properties: name, ageInWeeks, weightInKg and gender. The class also has an (abstract) method convertToMeat.
- The application also know CowFarm's that produce Cow's.
- In addition the application knows a Meat Factory, that asks the CowFarm to produce an Cow and then converts the Cow into Meat (which has weightInKg and pricePerKg as properties).
- Cow's method convertToMeat leads to meat with a value of 60 RMB per kg.

Can you make an information model for this application?

Case: Meat producing plant Design 1



Case: Meat producing plant
New requirements

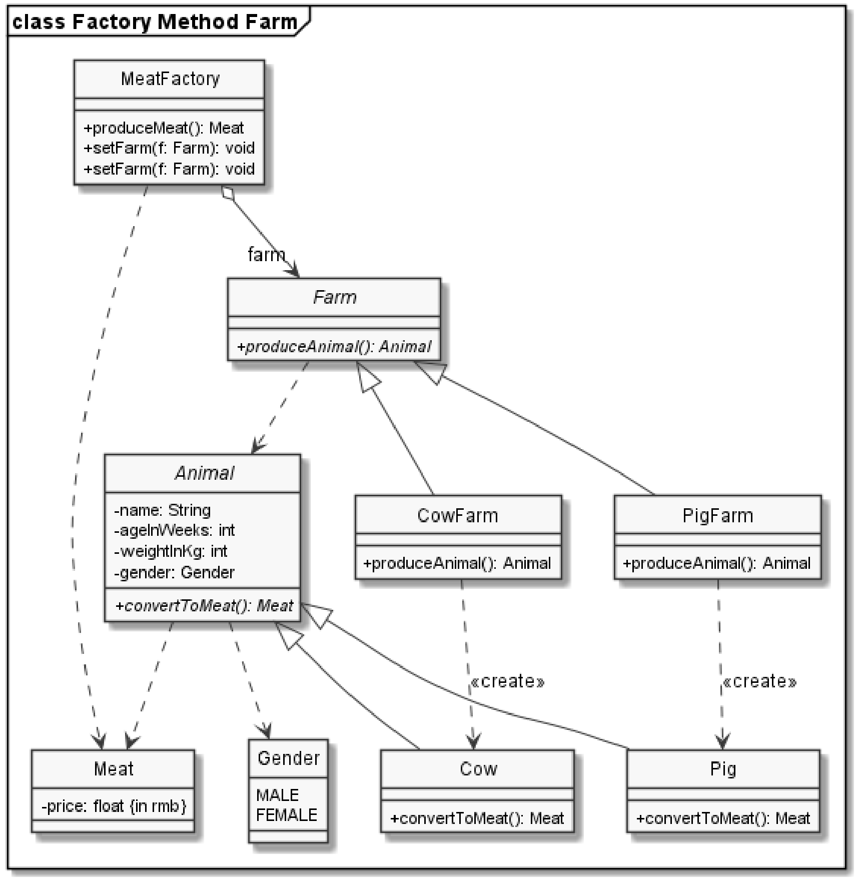
A software house decides to develop an application suite for a meat producing company.

The application receives the following new requirements:

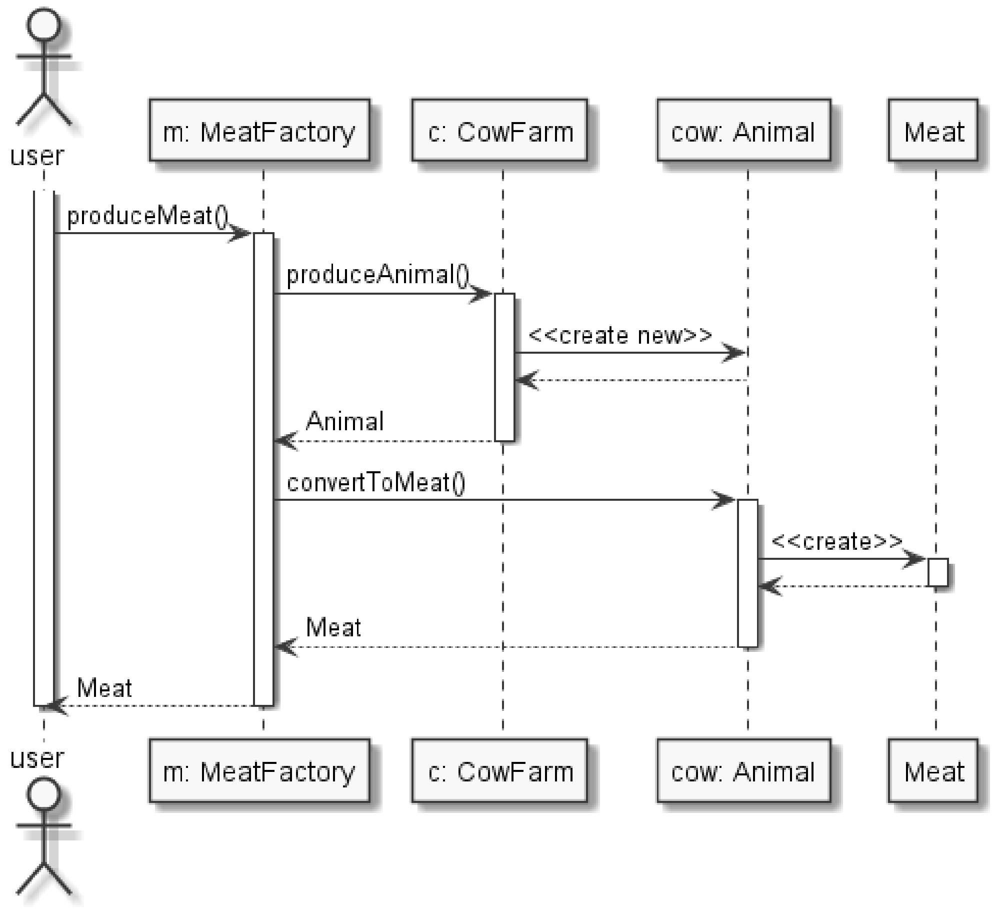
- There is also a need to support Pig’s and PigFarms.
- Cow’s method convertToMeat leads to meat with a value of 50 RMB per kg.

Can you make an adjusted information model for this application?

Case: Meat producing plant
Design 2



Case: Meat producing plant
Design 2



Let’s find a design pattern

A software house decides to develop an application suite for a meat producing company.

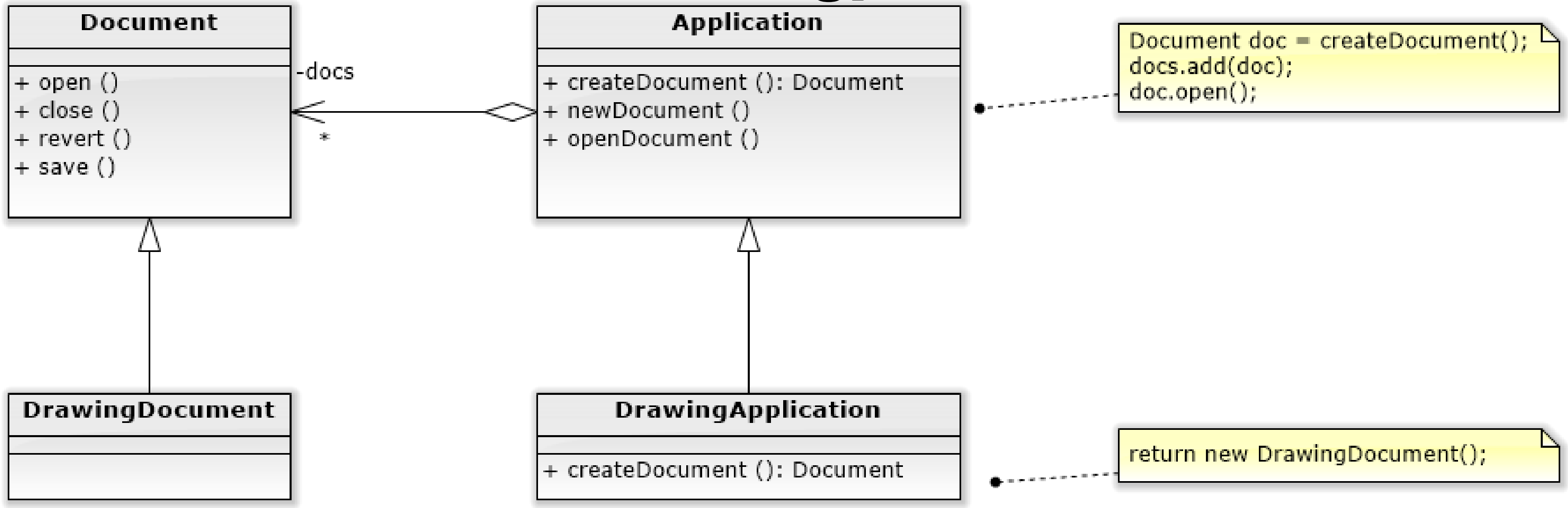
The application receives the following new requirements:

- There is also a need to support Chicken’s and ChickenFarms.
- Chicken’s method convertToMeat leads to meat with a value of 45 RMB per kg.

Can you make an adjusted information model for this application?

Factory Method

- **Intent:** Define an interface for creating an object, but let subclasses decide which class to instantiate…
- **Motivation:** [DP Book text omitted for brevity, but I'll describe it with the following]



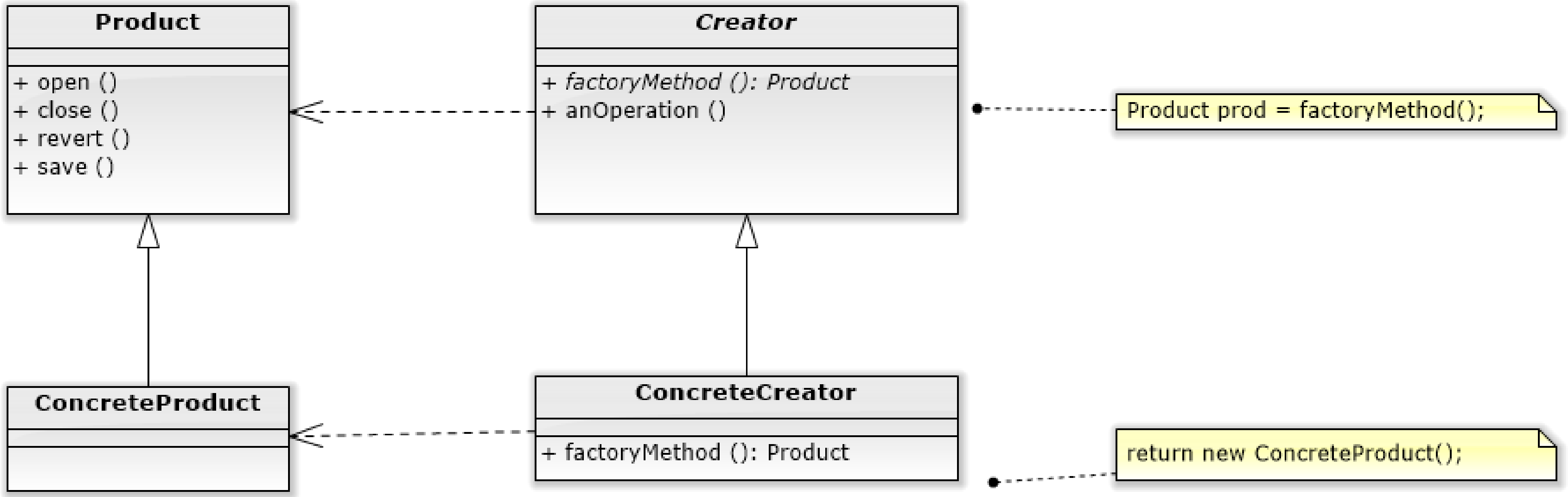
Factory Method

- **Applicability**: Use the Factory Method when:
  - A class can’t anticipate the class of objects it must create.

  - A class/interface requires its subclasses/implementation classes to specify the objects it creates.

Factory Method

- **Structure:**


Product
+ open ()
+ close ()
+ revert ()
+ save ()
Creator
+ factoryMethod (): Product
+ anOperation ()
Product prod = factoryMethod();
ConcreteProduct
ConcreteCreator
+ factoryMethod (): Product
return new ConcreteProduct();

Factory Method

- **Participants:**
  - Product (Document)
    - Defines the interface/abstract class of objects the factory method creates.
  - ConcreteProduct (DrawingDocument)
    - Implements/subclasses the Product interface/abstract class.
  - Creator (Application)
    - Declares the factory method, which returns an object of type Product. May also define a default implementation of factory method.
    - Might call the factory method from one of its operations.
  - ConcreteCreator (DrawingApplication)
    - Overrides/implements the factory method to return an instance of the ConcreteProduct.

Factory Method

- **Consequences**:
  - Creation requester class is independent of the class of concrete product objects actually created.
  - The set of product classes that can be instantiated can change dynamically.

- **Consequences in detail:**
  1. *Provides hooks for subclasses.*
     Creating objects inside a class with a factory method is always more flexible than creating an object directly. Factory Method gives subclasses a hook for providing an extended version of an object.
  2. *Connects parallel class hierarchies.*
     In the examples we've considered so far, the factory method is only called by Creators. But this doesn't have to be the case; clients can find factory methods useful, especially in the case of parallel class hierarchies.
     Parallel class hierarchies result when a class delegates some of its responsibilities to a separate class.
     Consider graphical figures that can be manipulated interactively; that is, they can be stretched, moved, or rotated using the mouse. Implementing such interactions isn't always easy. It often requires storing and updating information that records the state of the manipulation at a given time. This state is needed only during manipulation; therefore it needn't be kept in the figure object. Moreover, different figures behave differently when the user manipulates them. For example, stretching a line figure might have the effect of moving an endpoint, whereas stretching a text figure may change its line spacing.

Factory Method

- **Implementation**:
  - Two major varieties:
    - Creator is abstract class/interface without a default implementation for factory method.
    - A default implementation is provided.
  - Parameterised factory methods…

Design Principles dealt with by this pattern

Principles dealt with by this pattern:

- Encapsulate what varies…
- Program to a public interface, not an implementation…
- The open-closed principle…
- *The dependency inversion principle…*

Design Principle: Dependency Inversion

Depend upon abstractions.
Do not depend upon concrete classes.

a.k.a. Dependency Inversion principle

E. Freeman, E. Freeman, B. Bates, K. Sierra, Head First Design Patterns. O'Reilly, 2004. p. 139

Abstract Factory

Let’s find a design pattern

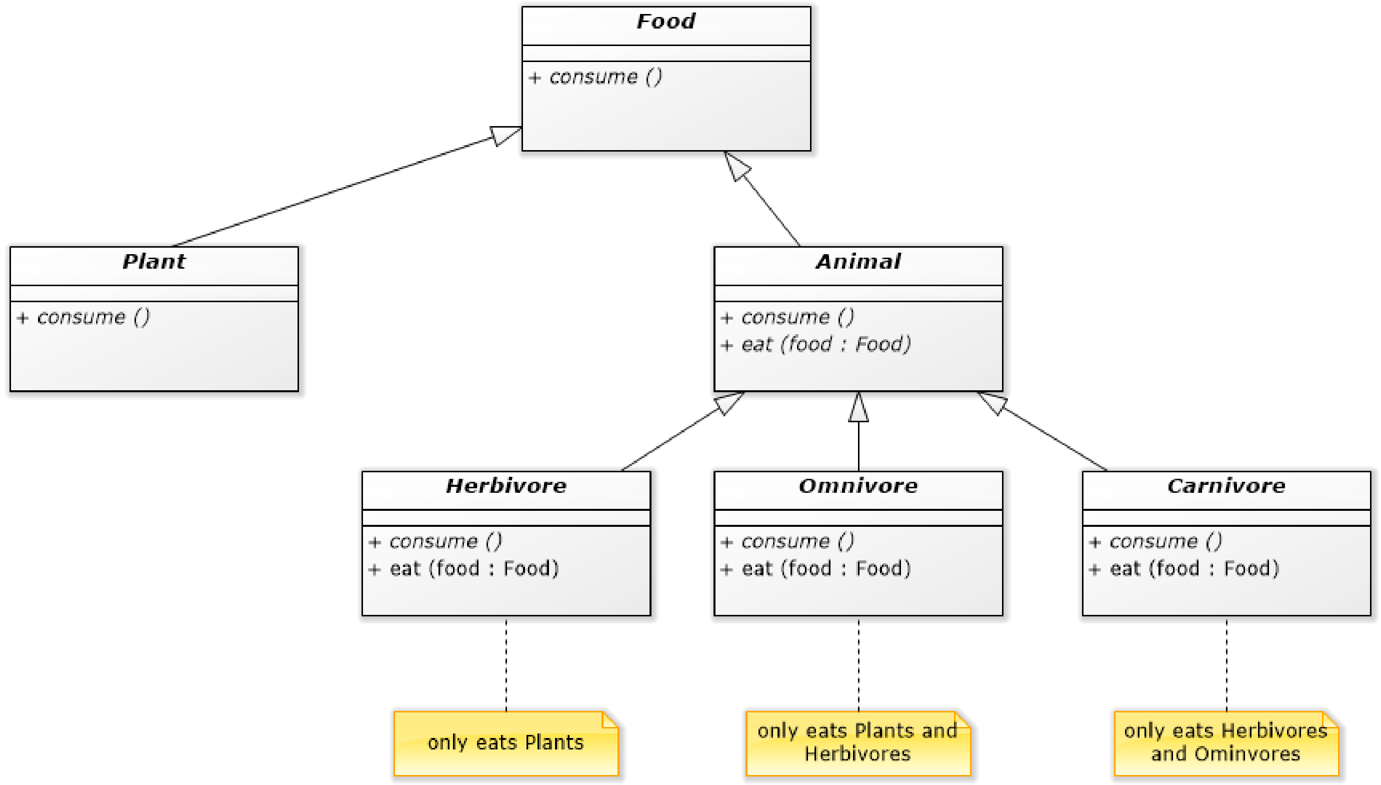
Will now present, on the board, and using Eclipse, a solution that utilizes the **abstract factory** design pattern

Case: Ecosystem simulator Requirements

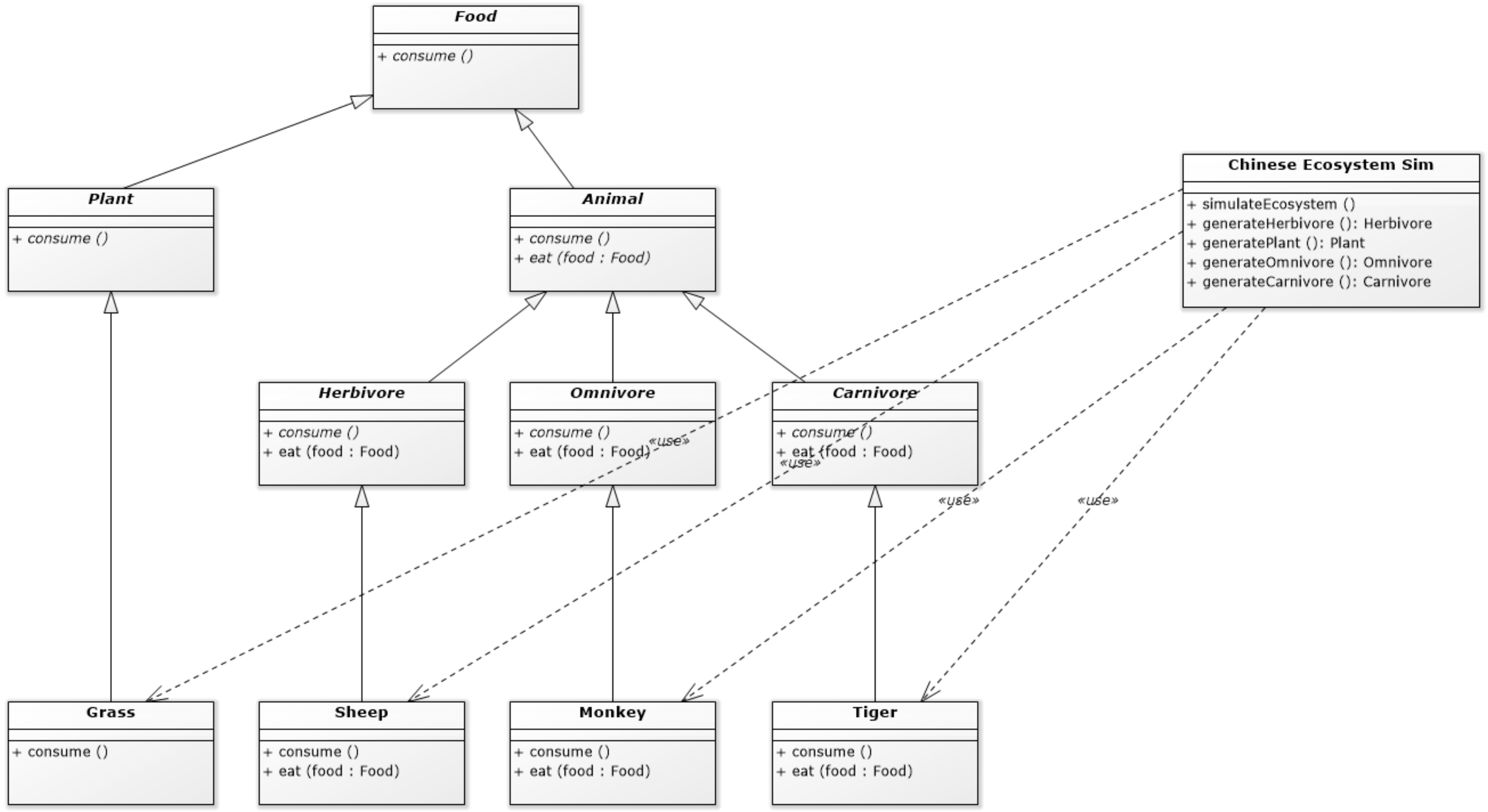
We decide to develop a simple software game that represents animals and their behaviours. The requirements are:

- We can choose whether to observe the ecosystem of Africa, East Asia, North America etc…and these will increase with new versions of the game.
- Each region has its own set of animals categorised as either herbivores, carnivores or omnivores, e.g. oxon, tiger, lion etc..
- In this simple game behaviours supported are:
  - Herbivores can only eat vegetation
  - Omnivores can eat vegetation and herbivores
  - Carnivores eat omnivores and herbivores

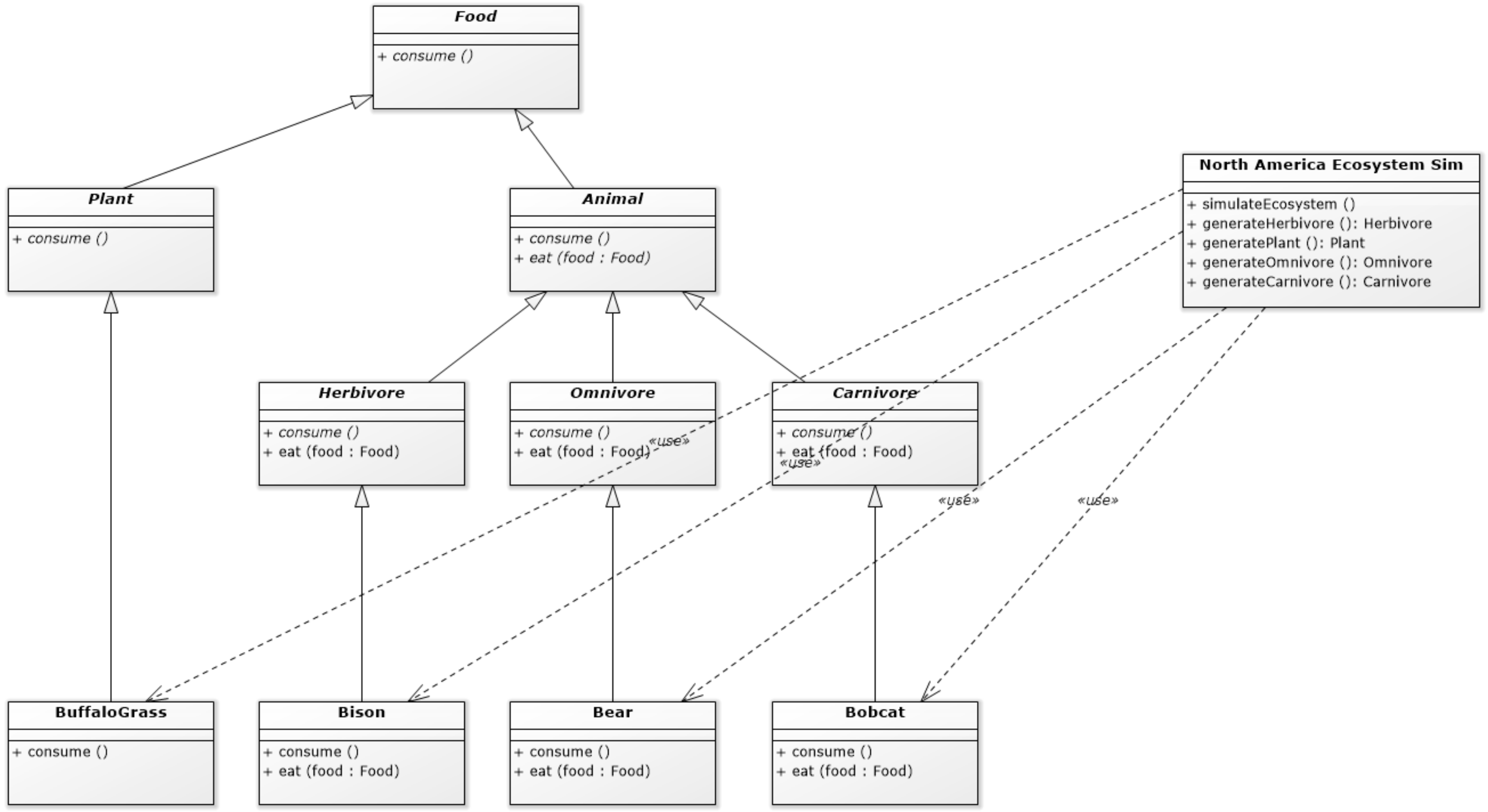
Case: Ecosystem simulator
Design: Domain model



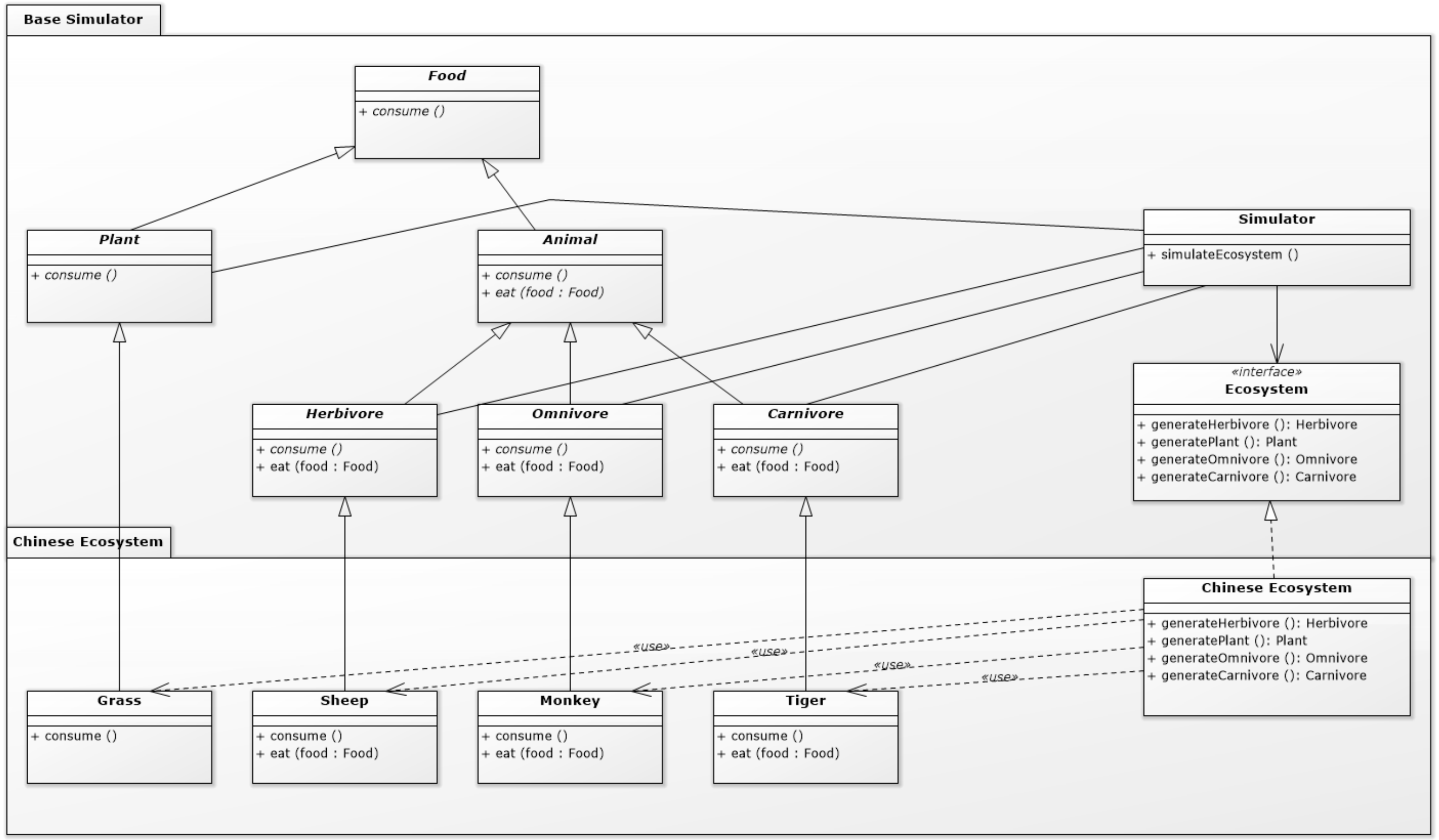
Case: Ecosystem simulator
Design: attempt 1 – duplication in ecosystem



Case: Ecosystem simulator
Design: attempt 1 – duplication in ecosystem



Case: Ecosystem simulator
Design: attempt 2

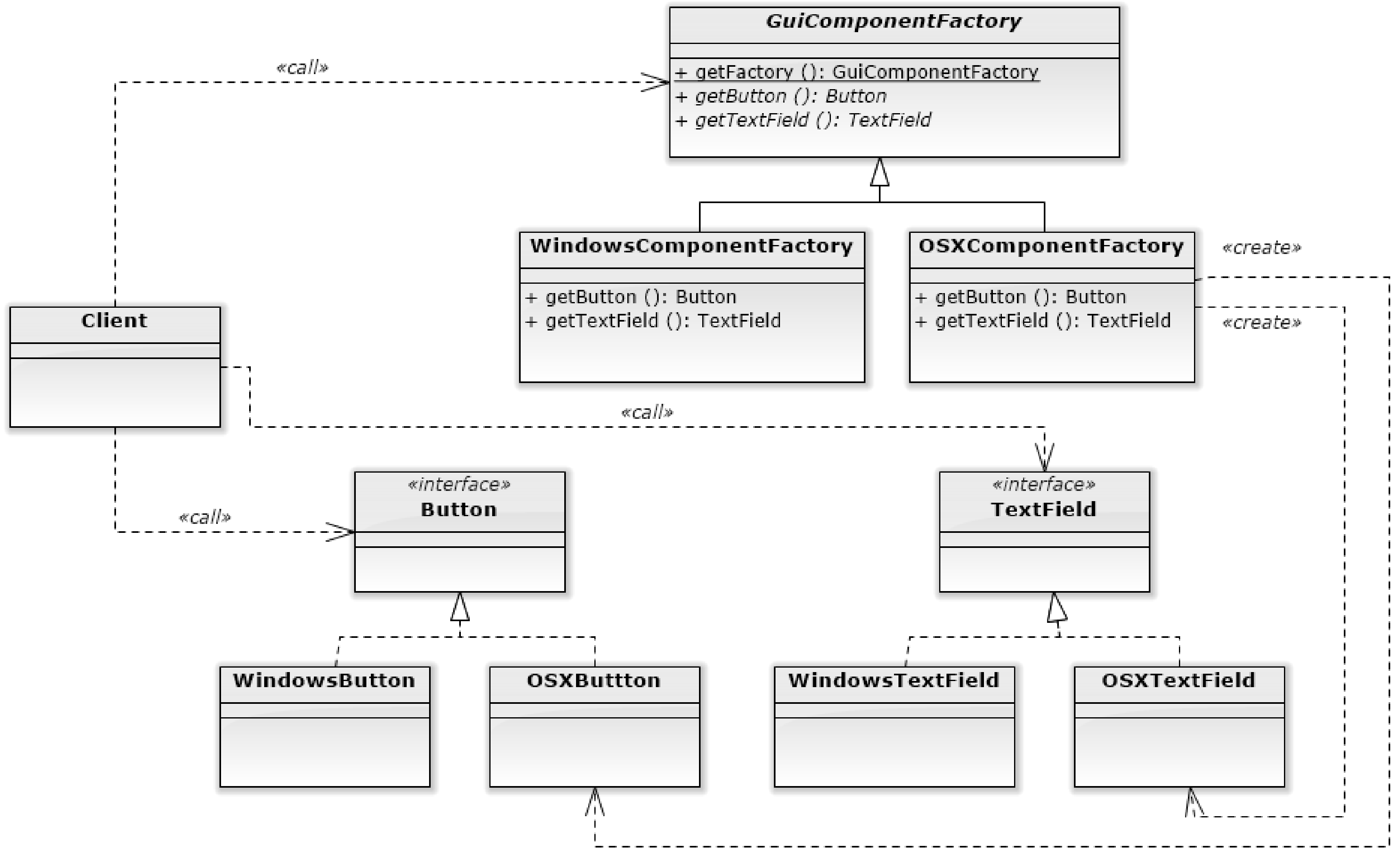


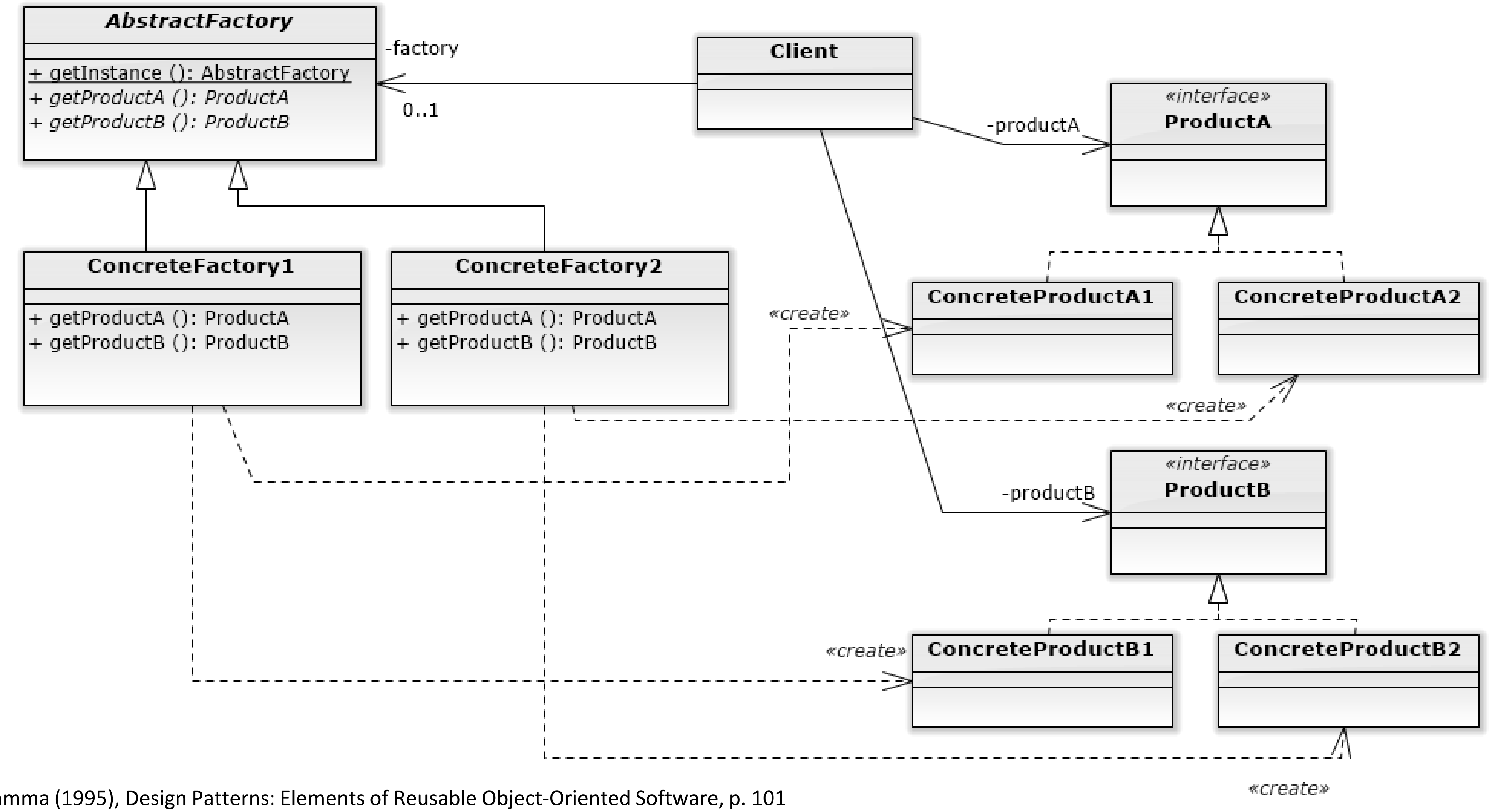
The Abstract Factory

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

- **Motivation:** Might want a GUI toolkit that supports multiple look-and-feels…Portability dictates that we should not hard-code concrete GUI component classes in client software, therefore…

  Define interfaces for GUI component classes. Also define factory classes that know how to create instances of GUI component classes. These factory classes present an interface to clients that delivers the abstract GUI component interfaces.


GuiComponentFactory
+ getFactory (): GuiComponentFactory
+ getButton (): Button
+ getTextField (): TextField
«call»
WindowsComponentFactory
+ getButton (): Button
+ getTextField (): TextField
OSXComponentFactory
+ getButton (): Button
+ getTextField (): TextField
«create»
«create»
Client
«call»
«call»
«interface»
Button
«interface»
TextField
WindowsButton
OSXButtton
WindowsTextField
OSXTextField



Gamma (1995), Design Patterns: Elements of Reusable Object-Oriented Software, p. 101

- **Participants**
  - AbstractFactory: Abstract operations to create abstract product objects and ConcreteFactory object
  - ConcreteFactory: Actually creates instances of concrete products, but returns as abstract type
  - AbstractProduct: Interface for a type of product
  - ConcreteProduct: Concrete implementation of an abstract product
  - Client: Uses only the interfaces declared by AbstractFactory and AbstractProduct

- **Collaborations**:
  - Normally single instance of ConcreteFactory created. This knows how to create ConcreteProduct objects of a particular type. For different products use a different ConcreteFactory
- **Consequences:**
  - Isolates concrete classes
  - Makes exchanging product families easy
  - Promotes consistency amongst products
  - Supporting new kinds of products is difficult

Gamma (1995), Design Patterns: Elements of Reusable Object-Oriented Software, p. 101-102

- **Consequences in detail:**
  1. *It isolates concrete classes.*
     The Abstract Factory pattern helps you control the classes of objects that an application creates. Because a factory encapsulates the responsibility and the process of creating product objects, it isolates clients from implementation classes. Clients manipulate instances through their abstract interfaces. Product class names are isolated in the implementation of the concrete factory; they do not appear in client code.
  2. *It makes exchanging product families easy.*
     The class of a concrete factory appears only once in an application—that is, where it's instantiated. This makes it easy to change the concrete factory an application uses. It can use different product configurations simply by changing the concrete factory. Because an abstract factory creates a complete family of products, the whole product family changes at once.
  3. *It promotes consistency among products.*
     When product objects in a family are designed to work together, it's important that an application use objects from only one family at a time. AbstractFactory makes this easy to enforce.
  4. *Supporting new kinds of products is difficult*
     Extending abstract factories to produce new kinds of Products isn't easy. That's because the AbstractFactory interface fixes the set of products that can be created. Supporting new kinds of products requires extending the factory interface, which involves changing the AbstractFactory class and all of its subclasses

Gemma (1995), Design Patterns: Elements of Reusable Object-Oriented Software, p. 102

- **Applicability:** Use when:
  - Application should be independent of how its products are created
  - Application should be configured with one of multiple families of products
  - A family of related product objects is designed to be used together and you wish to enforce this
  - You want to provide a class library of products and you want to reveal just their interfaces, not their implementations

- **Implementation:** (issues)
  - Factories as Singletons
  - Use of static for AbstractFactory’s getFactory method
  - Mechanisms used in getFactory for knowing which ConcreteFactory to return:
    - Static variable with appropriate factory instance
    - Parameter provided by client and hard-coded logic
    - External config information allows mapping to class name from which we create an instance

Reading

For this lesson please read:

- Chapter 4 (Baking with OO Goodness) of Head First Design Patterns