YORK UNIVERSITÉ UNIVERSITY

Design Patterns
EECS 3311

Ilir Dema
demailir@eecs.yorku.ca

Overview of Last Lectures

- **Control Flow Graphs**
  - If Statements
  - Loop Statements
- **Code Coverage**
  - JaCoCo
- **UML**
  - Use Case Diagram
  - Activity Diagram
  - Sequence Diagram
  - **Class Diagram**
    - different relations

Acknowledge

- Some of the covered materials are based on SOEN 344 at Concordia, SE463 at UW, SENG321 at UVic, CSE331 at University of Washington and previous EECS3311 offerings:
  - Nikolaos Tsantalis, Jo Atlee, Marty Stepp, Mike Godfrey, Jonathan S. Ostroff, M. Ernst, S. Reges, D. Notkin, R. Mercer, Davor Svetinovic, Jack Jiang, Jackie Wang

Outlines

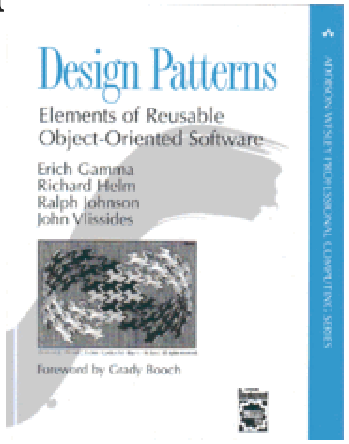
- **Introduction to Design Patterns**
  - Pattern's Elements
  - Types of Design Patterns

- **Java Design Patterns**
  - The Singleton Pattern
  - The Factory Pattern
  - The Builder Pattern
  - The Prototype Pattern
  - The Adapter Pattern
  - The Bridge Pattern
  - The Composite Pattern
  - Visitor Design Pattern
  - Iterator Pattern
  - State Design Pattern
  - Event-driven pattern
  - Observer design patterns

Design patterns

- **Design pattern**: A standard solution to a common software problem in a context.
  - describes a **recurring software structure or idiom**
  - is **abstract** from any particular programming language
  - identifies classes and their roles in the solution to a problem

- In 1990 a group called the *Gang of Four* or "**GoF**" (Gamma, Helm, Johnson, Vlissides) compile a catalog of design patterns
  - 1995 book *Design Patterns: Elements of Reusable Object-Oriented Software* is a classic of the field


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Pattern’s Elements

In general, a pattern has four essential elements.

- The pattern name
- The problem
- The solution
- The consequences

Pattern's Elements – The Pattern Name

The **pattern name** is a handle we can use to describe a design problem, its solutions, and consequences in a word or two.

- Naming a pattern immediately increases the design vocabulary. It lets us design at a higher level of abstraction.
- Having a vocabulary for patterns lets us talk about them.
- It makes it easier to think about designs and to communicate them and their trade-offs to others.

Pattern’s Elements – The Problem

The **problem** describes when to apply the pattern.

- It explains the problem and its context.
- It might describe specific design problems such as how to represent algorithms as objects.
- It might describe class or object structures that are symptomatic of an inflexible design.
- Sometimes the problem will include a list of conditions that must be met before it makes sense to apply the pattern.

Pattern’s Elements – The Solution

The **solution** describes the elements that make up the design, their relationships, responsibilities, and collaborations.

- The solution doesn't describe a particular concrete design or implementation, because a pattern is like a template that can be applied in many different situations.

- Instead, the pattern provides an abstract description of a design problem and how a general arrangement of elements (classes and objects in our case) solves it.

Pattern’s Elements – The Consequences

The **consequences** are the results and trade-offs of applying the pattern.

- The consequences for software often concern space and time trade-offs.
- They may address language and implementation issues as well.
- Since reuse is often a factor in object-oriented design, the consequences of a pattern include **its impact on a system's flexibility, extensibility, or portability**.
- Listing these consequences explicitly helps you understand and evaluate them.

Benefits of using patterns

- Patterns give a design **common vocabulary** for software design:
  - Allows engineers to abstract a problem and talk about that abstraction in isolation from its implementation.
  - A culture; domain-specific patterns increase design speed.

- **Capture expertise** and allow it to be communicated:
  - Promotes design reuse and avoid mistakes.
  - Makes it easier for other developers to understand a system.

- **Improve documentation** (less is needed):
  - Improve understandability (patterns are described well, once).

Gang of Four (GoF) patterns

Erich Gamma, Richard Helm, Ralph Johnson and John Vlisides in their **Design Patterns** book define 23 design patterns divided into three types:

- ***Creational patterns*** are ones that **create objects** for you, rather than having you instantiate objects directly. This gives your program more flexibility in deciding which objects need to be created for a given case.
- ***Structural patterns*** help you c**ompose groups of objects into larger structures**, such as complex user interfaces or accounting data.
- ***Behavioral patterns*** help you **define the communication between objects** in your system and how the flow is controlled in a complex program.

Gang of Four (GoF) patterns (cont.)

- **Creational Patterns** *(abstracting the object-instantiation process)*
  - Singleton Factory
  - Builder Prototype …
- **Structural Patterns** *(how objects/classes can be combined)*
  - Adapter Bridge Composite
  - Decorator Facade Flyweight
  - Proxy …
- **Behavioral Patterns** *(communication between objects)*
  - Command Interpreter Iterator
  - Observer State
  - Visitor Even-driven …

Pattern: Singleton

A class that has only a single instance

Singleton pattern

- **singleton**: An object that is **the only object of its type**.
  (one of the most known / popular design patterns)

  - Ensuring that a class has at most one instance.
  - Providing a global access point to that instance.
    - e.g. Provide an accessor method that allows users to see the instance.

- *Benefits:*
  - Takes responsibility of managing that instance away from the programmer (illegal to construct more instances).
  - Saves memory.
  - Avoids bugs arising from multiple instances.

Restricting objects

- One way to avoid creating objects: use static methods
  - Examples: Math, System
  - Is this a good alternative choice? Why or why not?

- *Disadvantage*: Lacks flexibility.
  - **Static methods can't be passed as an argument**, nor returned.

- *Disadvantage*: Cannot be extended.
  - Static methods **can't be subclassed and overridden** like an object's methods could be.

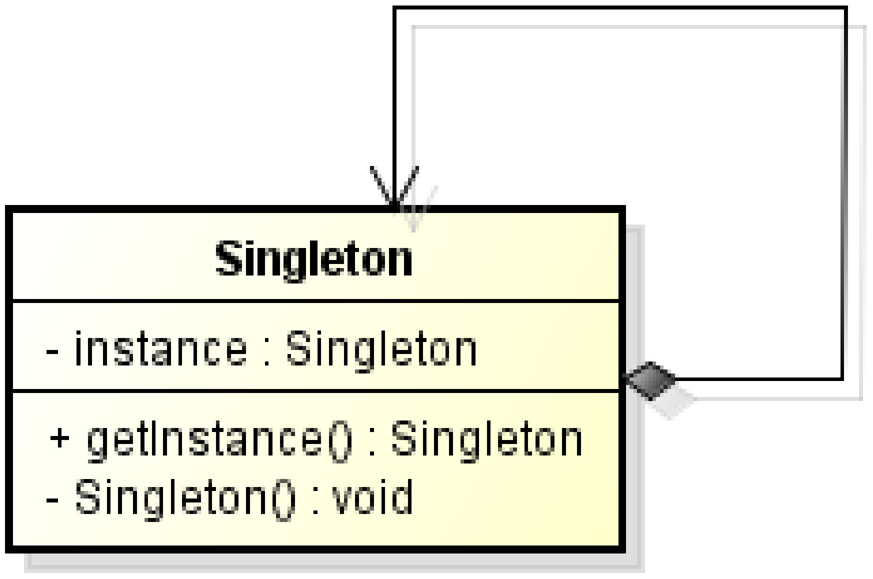
Implementing Singleton

- Make ***constructor(s) private*** so that they can not be called from outside by clients.

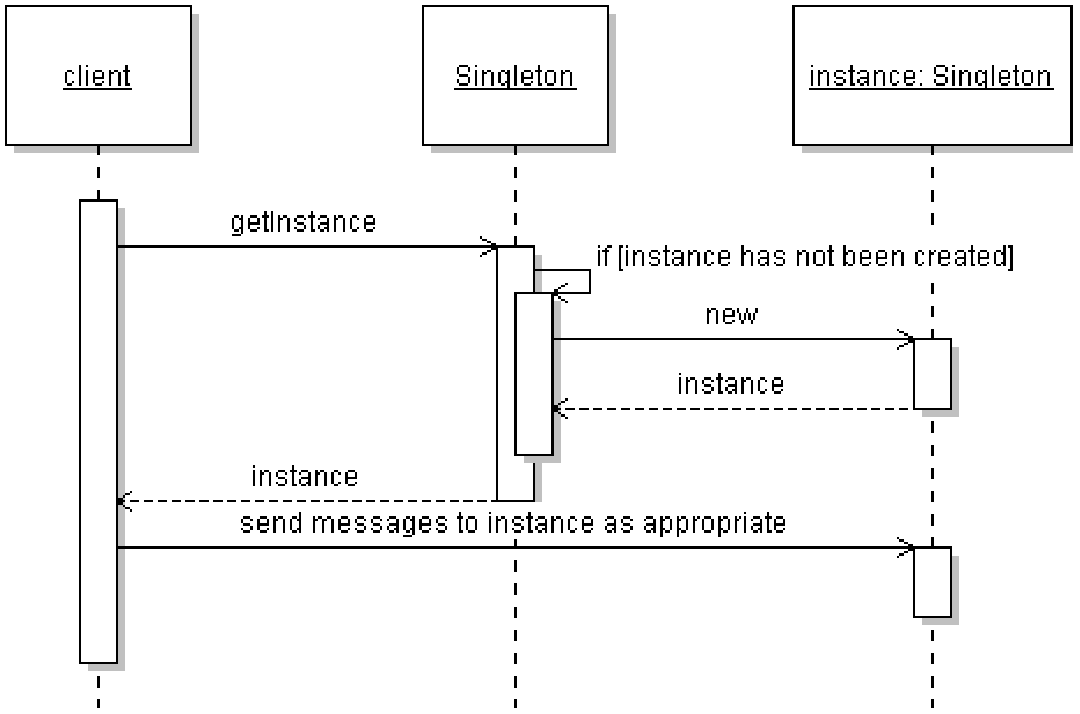
- Declare a single ***private static*** instance of the class.

- Write a ***public getInstance()*** or similar method that allows access to the single instance.
  - May need to protect / synchronize this method to ensure that it will work in a multi-threaded program.

Singleton class diagram



Singleton sequence diagram


client
Singleton
instance: Singleton
getInstance
if [instance has not been created]
new
instance
instance
send messages to instance as appropriate

Singleton example

- Class `RandomGenerator` generates random numbers.

```
public class RandomGenerator {
    private static final RandomGenerator gen =
             new RandomGenerator();

    public static RandomGenerator getInstance() {
        return gen;
    }

    private RandomGenerator() {}

    ...
}
```

Lazy initialization

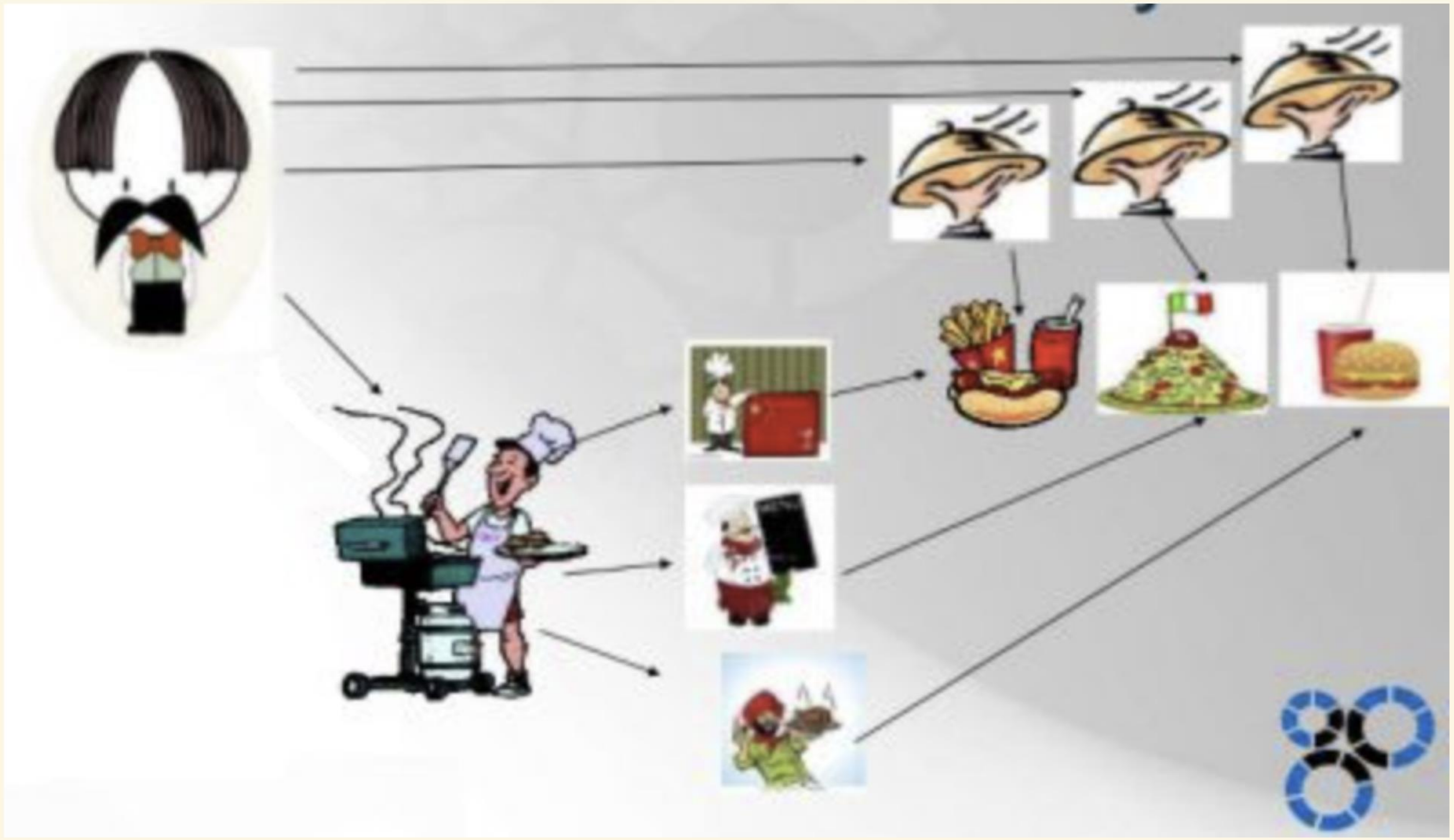
- Can wait until client asks for the instance to create it:

```
public class RandomGenerator {
    private static RandomGenerator gen = null;

    public static RandomGenerator getInstance() {
        if (gen == null) {
            gen = new RandomGenerator();
        }
        return gen;
    }

    private RandomGenerator() {}

    ...
}
```

Pattern: Factory

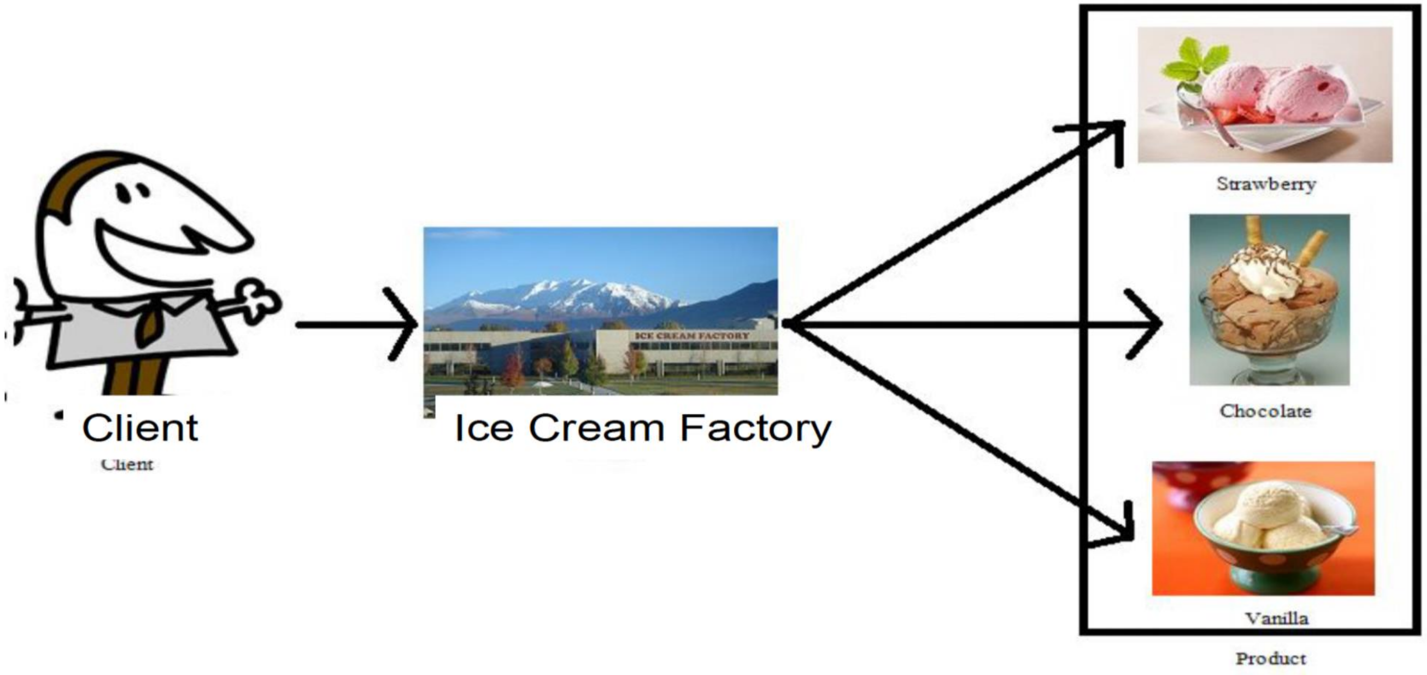
WHY DO WE NEED FACTORY PATTERN?

… AND HOW TO RECOGNIZE WHEN!

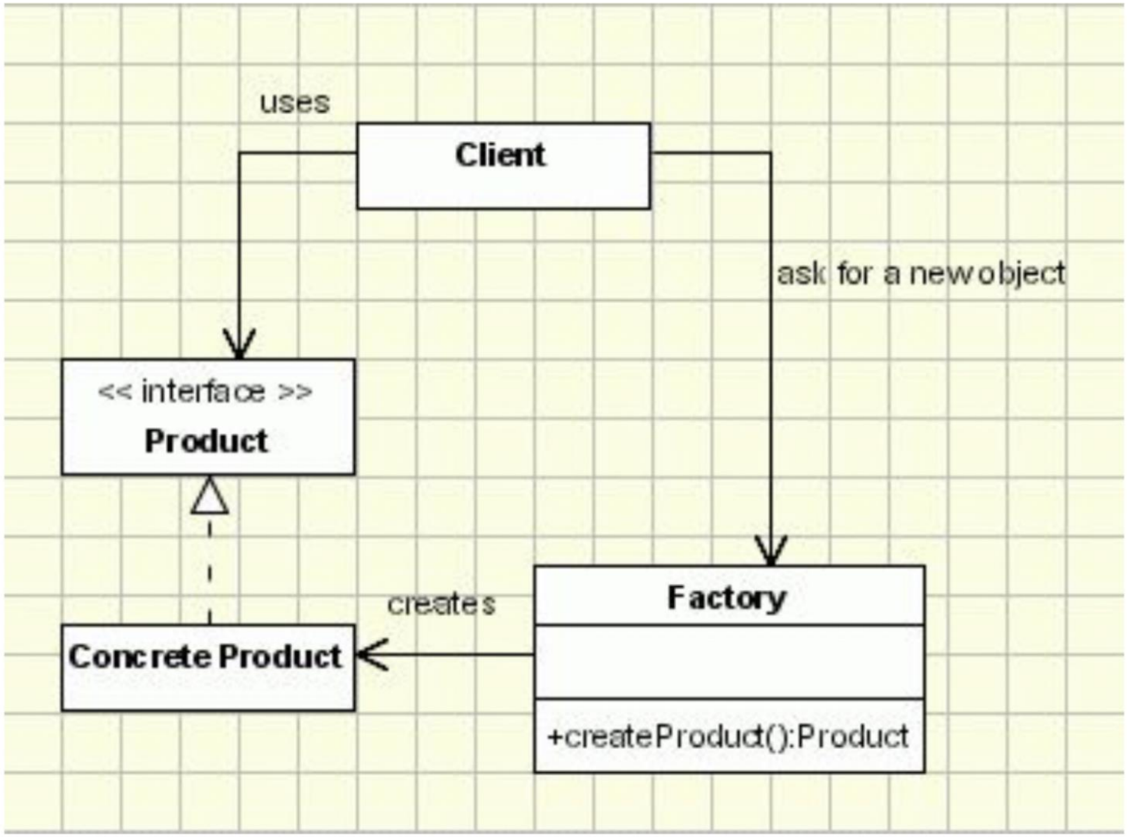


How does it Work?

The Factory pattern returns an instance of one of several possible classes depending on the data provided to it.



UML of the **Factory** pattern

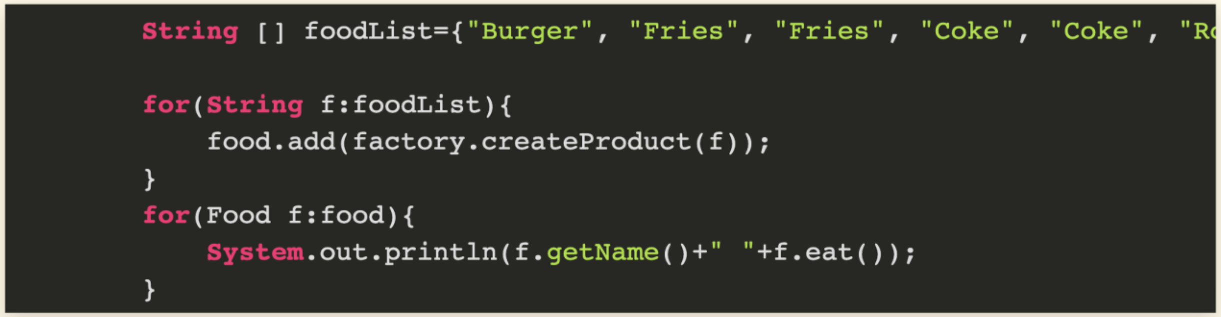


WHEN TO USE THE FACTORY PATTERN

- This is one of the most used design patterns in practice.
- Creates objects without exposing the instantiation logic to the client.
  - `Food food = FoodFactory.createProduct(“Burger”);`
  - `food` is a `Burger` object
- Refers to the newly created object through a common interface.

```
food.getName();
food.getCalories();
food.eat();
```

THE CLIENT CLASS:



```
        String [] foodList={"Burger", "Fries", "Fries", "Coke", "Coke", "Ro

        for(String f:foodList){
            food.add(factory.createProduct(f));
        }
        for(Food f:food){
            System.out.println(f.getName()+" "+f.eat());
        }
```

HOW TO IMPLEMENT THE FACTORY PATTERN

- Create a base class or interface for the product
  - e.g., Food
- Implement concrete product classes by extending the base class
  - e.g., Burger, Pizza, Salad, etc.
- Create the Factory class with a createProduct() method
  - ProductBase createProduct(String productID)
  - return objects of different types according to productID.

Using a Factory Pattern

You should consider using a Factory pattern when:

- **Create objects dynamically**
- **A class can't anticipate which kind of class of objects it must create**.
- A class uses its **subclasses** to specify which objects it creates.
- You want to localize the knowledge of which class gets created.

There are several similar variations on the factory pattern to recognize:

- The base class is abstract and the pattern must return a complete working class.

- The base class contains default methods and is only subclassed for cases where the default methods are insufficient.

- Parameters are passed to the factory telling it which of several class types to return. In this case the classes may share the same method names but may do something quite different.

The Abstract Factory Pattern

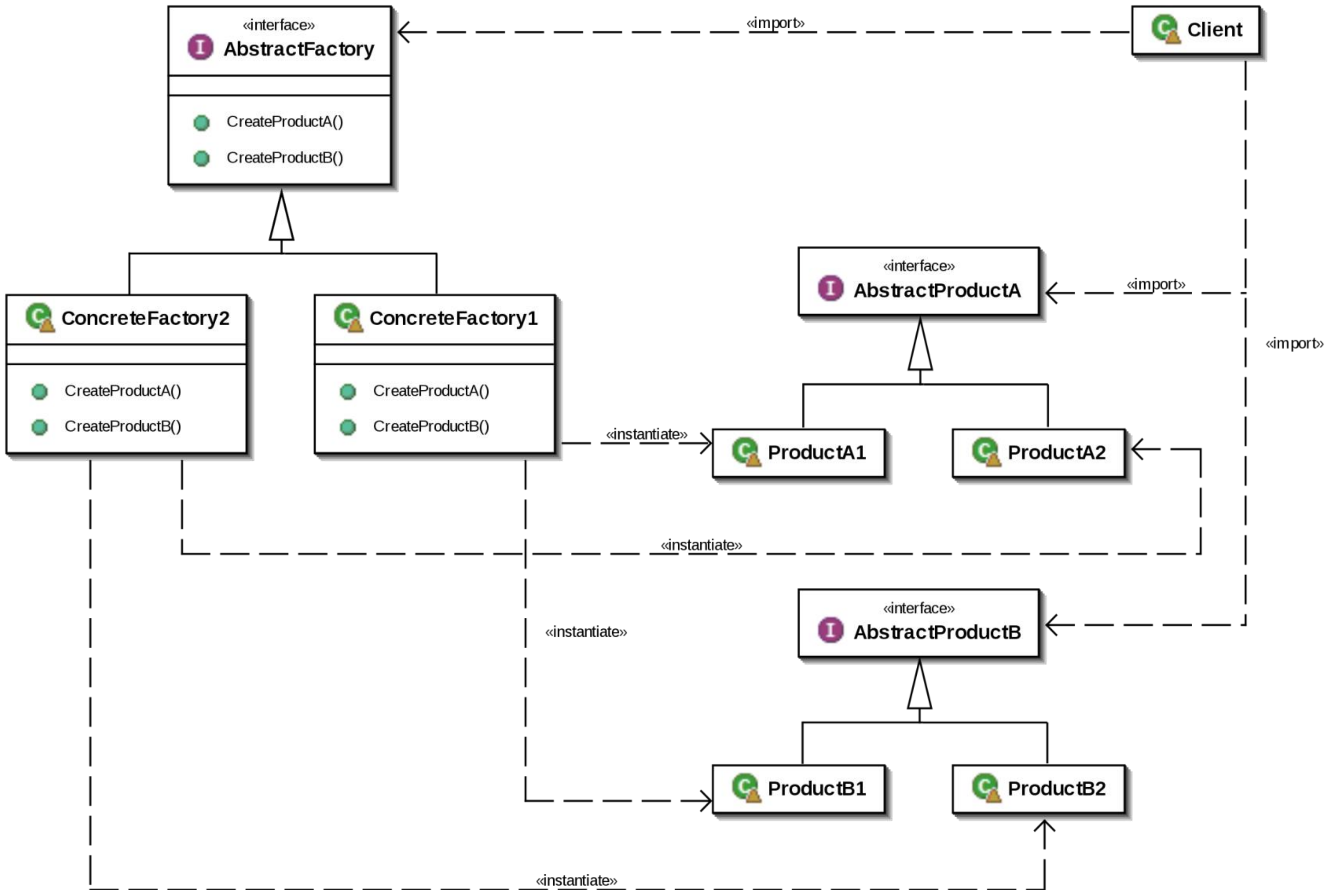
How does it Work?

The Abstract Factory pattern is one level of abstraction **higher than the factory pattern.** This pattern returns one of several related classes, each of which can return several different objects on request. In other words, **the Abstract Factory is a factory object that returns one of several factories.**

One classic application of the abstract factory is the case where your system needs to support multiple "look-and-feel" user interfaces, such as Windows, Motif or Macintosh:

- You tell the factory that you want your program to look like Windows and it returns a GUI factory which returns Windows-like objects.

- When you request specific objects such as buttons, check boxes and windows, the GUI factory returns Windows instances of these visual interface components.


«interface»
AbstractFactory
CreateProductA()
CreateProductB()
«import»
Client
ConcreteFactory2
CreateProductA()
CreateProductB()
ConcreteFactory1
CreateProductA()
CreateProductB()
«interface»
AbstractProductA
«import»
«import»
«instantiate»
ProductA1
ProductA2
«instantiate»
«instantiate»
«interface»
AbstractProductB
ProductB1
ProductB2
«instantiate»

A Garden Maker Factory

Suppose you are writing a program to plan the layout of gardens. These could be annual gardens, vegetable gardens or perennial gardens. However, no matter which kind of garden you are planning, you want to ask the same questions:

- What are good border plants?
- What are good center plants?
- What plants do well in partial shade?

We want a base *Garden* class that can answer these questions:

```
public abstract class Garden {
    public abstract Plant getCenter();
    public abstract Plant getBorder();
    public abstract Plant getShade();
}
```

The Plant Class

The *Plant* class simply contains and returns the plant name:

```
public class Plant {
      String name;
      public Plant(String pname) {
        name = pname; //save name
      }
      public String getName() {
        return name;
      }
}
```

A Garden Class

A Garden class simply returns one kind of each plant. So, for example, for the **vegetable garden** we simply write:

```
public class VegieGarden extends Garden {
    public Plant getShade() {
        return new Plant("Broccoli");
    }
    public Plant getCenter() {
        return new Plant("Corn");
    }
    public Plant getBorder() {
        return new Plant("Peas");
    }
}
```

A Garden Maker Class – The Abstract Factory

We create a series of *Garden* classes - ***VegieGarden***, ***PerennialGarden, and AnnualGarden,*** each of which returns one of several Plant objects. Next, we construct our **abstract factory** to return an object instantiated from one of these *Garden* classes and based on the string it is given as an argument:

```
class GardenMaker {
    //Abstract Factory which returns one of three gardens
   private Garden gd;
   public Garden getGarden(String gtype) {
      gd = new VegieGarden(); //default
      if(gtype.equals("Perennial"))
            gd = new PerennialGarden();
      if(gtype.equals("Annual"))
            gd = new AnnualGarden();
      return gd;
   }
}
```

Consequences of Abstract Factory

- One of the main purposes of the **Abstract Factory** is that **it isolates the concrete classes that are generated**.
- The actual class names of these classes are hidden in the factory and need not be known at the client level at all.
- Because of the isolation of classes, you can change or interchange these product class families freely.
- Since you generate only one kind of concrete class, this system keeps you for inadvertently using classes from different families of products.
- While all of the classes that the Abstract Factory generates have the same base class, there is nothing to prevent some derived classes from having additional methods that differ from the methods of other classes.