Factory Pattern

Factory Pattern

- The factory patten is another way to instantiate an object instead of invoking the contractor directly in the client's application.
- The instantiation of the object is encapsulated to avoid coupling problem.
- Every time you use the constructor to create an object, you are creating a concrete object. For example, obj = Student();
- The following example leads to a maintenance problem. And this code might appear several places in your application.
- When you have code that uses many concrete classes, you will need to change it when new concrete classes are added.

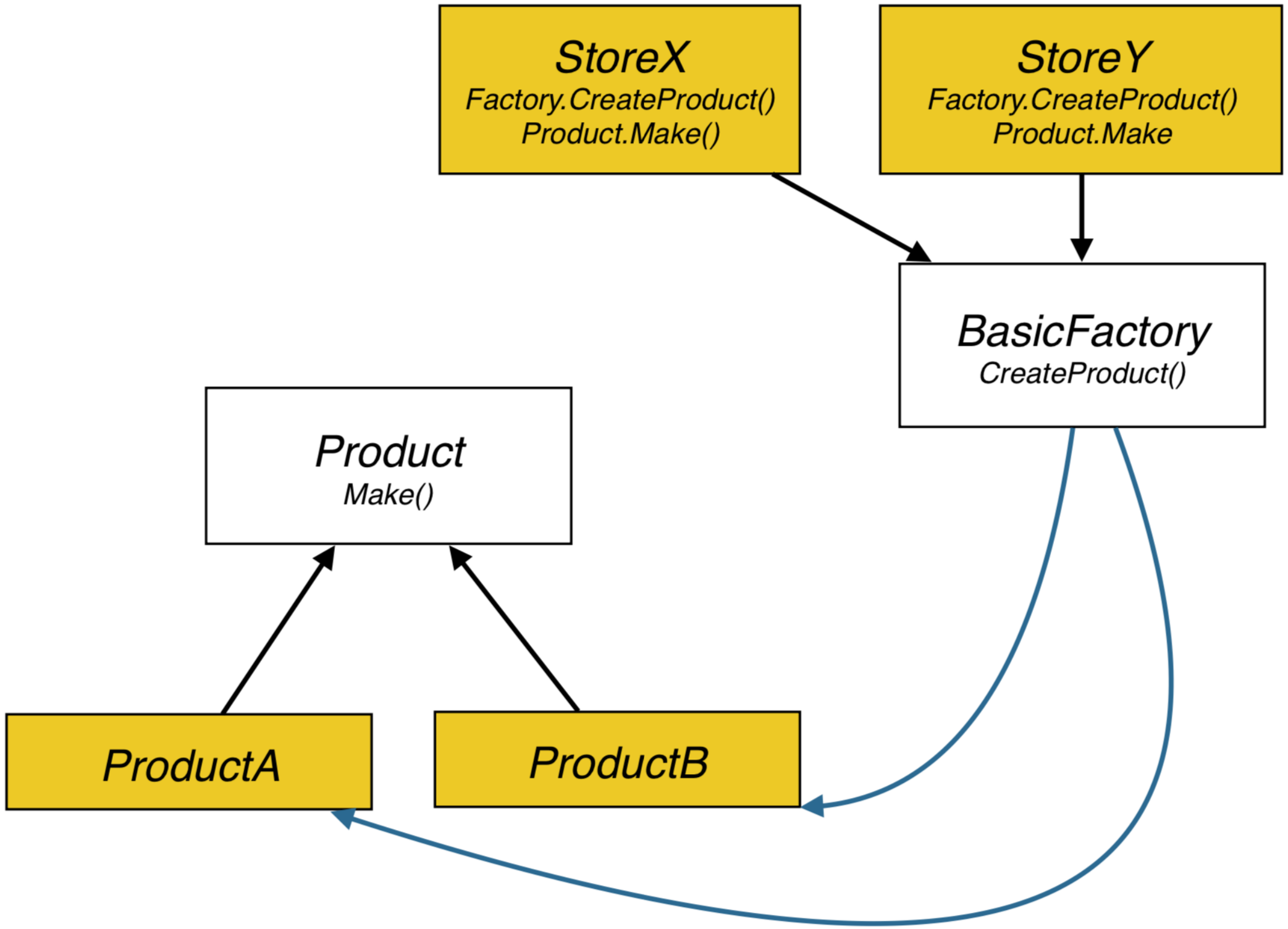
```
from abc import ABC, abstractmethod
import enum

class Product(ABC):
        def make(self, type):
                return " is making a product " + type

        @abstractmethod
        def getName(self):
                pass

class ProductA(Product):
        def getName(self):
                return "Product A"

class ProductB(Product):
        def getName(self):
                return "Product B"

class StoreX:
        def order(self, customer, type):
                product = None
                if type == "A":
                        product = ProductA()
                elif type == "B":
                        product = ProductB()

                print("Store X -", product.make(type))
                print(customer, "ordered", product.getName(), "from Store X")
                return product

class StoreY:
        def order(self, customer, type):
                product = None
                if type == "A":
                        product = ProductA()
                elif type == "B":
                        product = ProductB()

                print("Store Y -", product.make(type))
                print(customer, "ordered", product.getName(), "from Store Y")
                return product


def main():
        print("Enter a product type: ")
        type = input()

        store1 = StoreX()
        store1.order("Peter", type)
        store2 = StoreY()
        store2.order("Lily", type)

main()
```

This program leads to a maintenance problem

Encapsulating object creation

- We are going to take the object creation code out of the order() method into another object that concerns with creating products.
- Call this new object as a Factory.
- The BasicComputerFactory class that handles the details of object creation.
- Advantages
  - The Factory can have many different clients. By encapsulating the computer creating in one class, we now have only one place to make modifications.
  - Composition allows us to change behavior dynamically at runtime because we can swap in and out implementation.

Encapsulating object creation



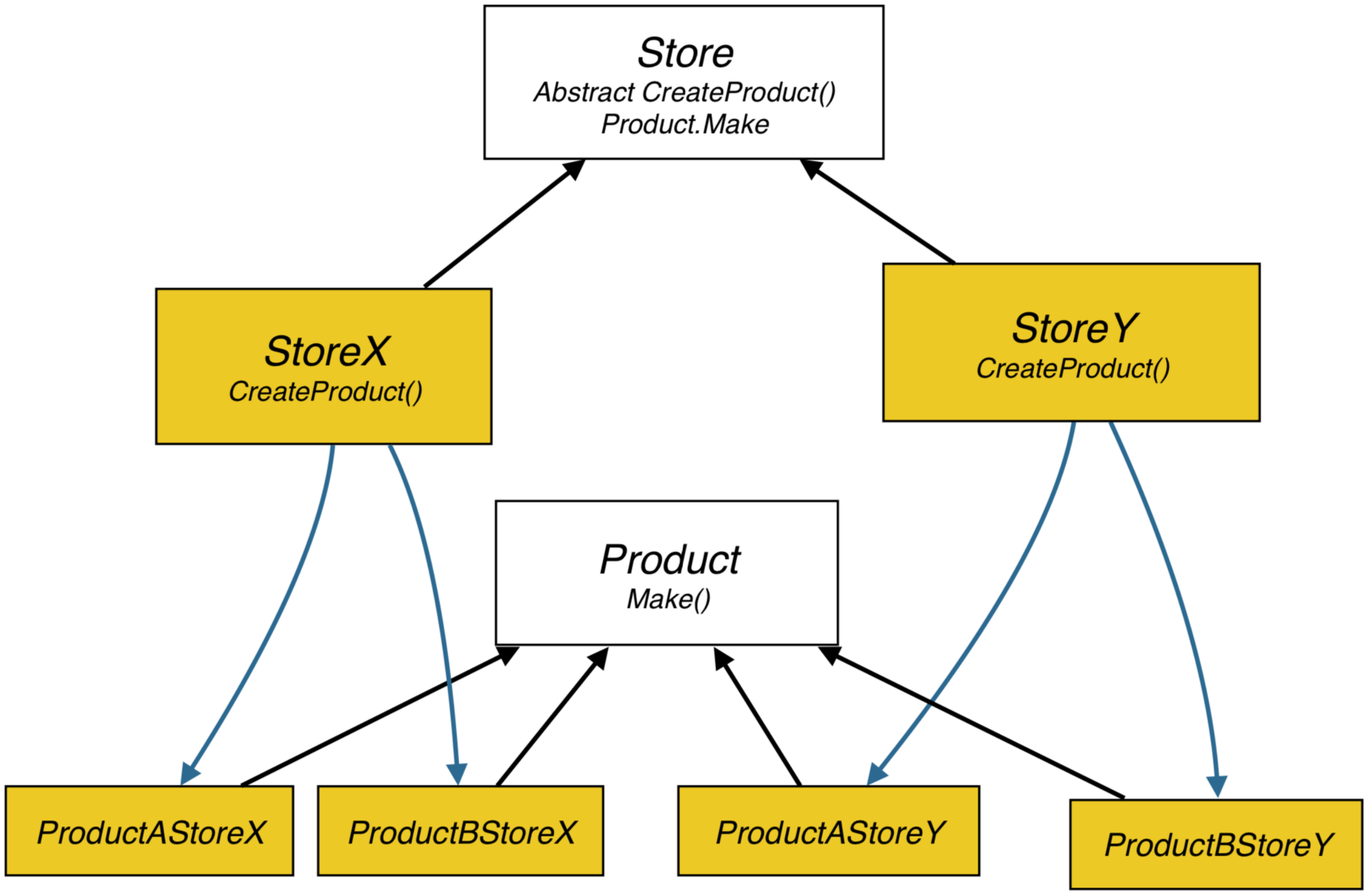
```
from abc import ABC, abstractmethod
import enum

class Product(ABC):
        def make(self, type):
                return " is making a product " + type

        @abstractmethod
        def getName(self):
                pass

class ProductA(Product):
        def getName(self):
                return "Product A"

class ProductB(Product):
        def getName(self):
                return "Product B"

class BasicFactory:
        def createProduct(self, type):
                product = None;

                if type == "A":
                        product = ProductA()
                elif type == "B":
                        product = ProductB()
                return product

class StoreX:
        def order(self, customer, type):
                factory = BasicFactory()
                product = factory.createProduct(type)
                print("Store X -", product.make(type))
                print(customer, "ordered", product.getName(), "from Store X")
                return product

class StoreY:
        def order(self, customer, type):
                factory = BasicFactory()
                product = factory.createProduct(type)
                print("Store Y -", product.make(type))
                print(customer, "ordered", product.getName(), "from Store Y")
                return product


def main():
        print("Enter a product type: ")
        type = input()

        store1 = StoreX()
        store1.order("Peter", type)
        store2 = StoreY()
        store2.order("Lily", type)
```

The Example of Encapsulating object creation

Factory method pattern

- Defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclass.
- The abstract creator gives you an interface with a method of creating objects, known as "factory method"
- Any other method implemented in the abstract Creator are written to operate on products produced by the factory method.
- Subclasses actually implement the factory method and create product.
- Advantages:
  - To control all the product making activities to the Store class, and yet give the freedom to the individual stores to have their own specialized products.
  - Make Store class as abstract and put the createProduct() method back but mark it as abstract too.
  - Then we create subclasses of Store class. And each subclass can make the decision to make the product.

Factory method pattern

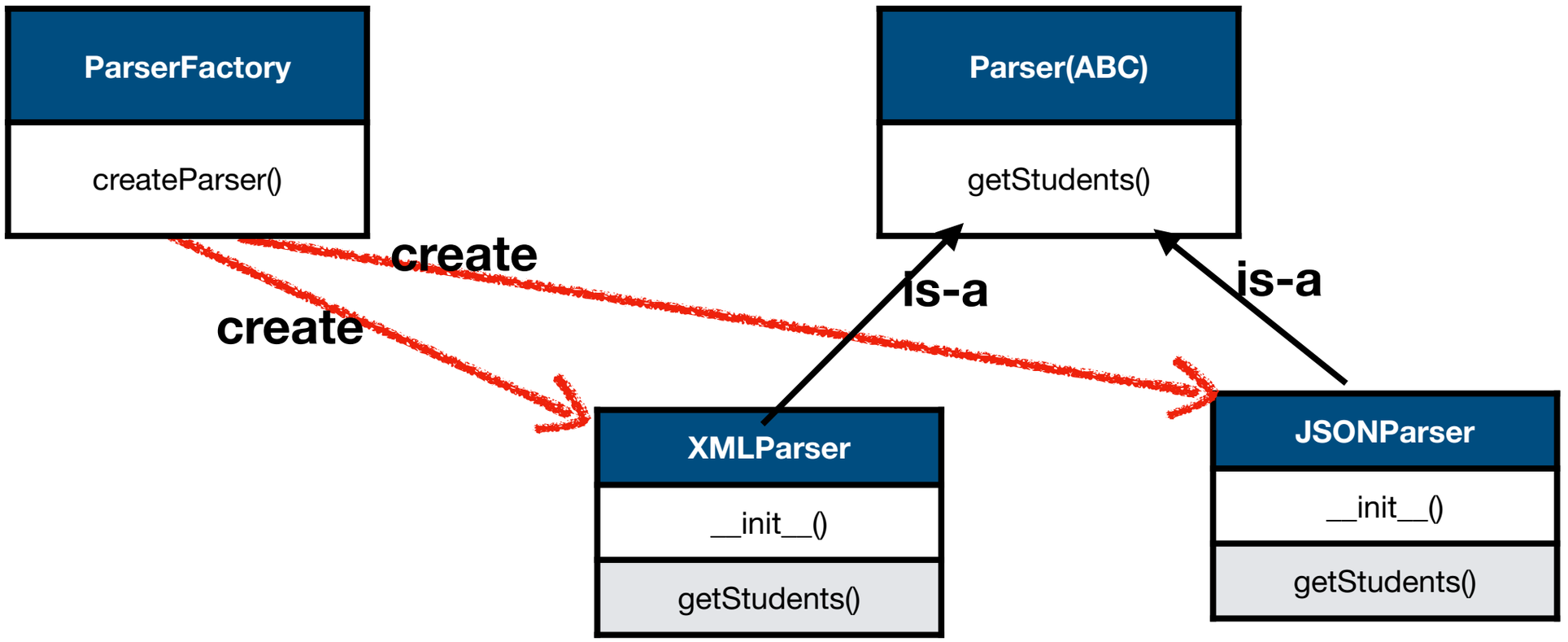


```
class Product(ABC):
	def make(self, type):
		return " is making a product " + type

	@abstractmethod
	def getName(self):
		pass

class ProductAStoreX(Product):
	def getName(self):
		return "Product A from Store X"

class ProductAStoreY(Product):
	def getName(self):
		return "Product A from Store Y"

class ProductBStoreX(Product):
	def getName(self):
		return "Product B from Store X"

class ProductBStoreY(Product):
	def getName(self):
		return "Product B from Store Y"

class Store(ABC):
	def order(self, customer, type):
		product = self.createProduct(type)
		print(product.make(type))
		print(customer, "ordered", product.getName(), "from Store X")
		return product

	@abstractmethod
	def createProduct(self, type):
		passs

class StoreX(Store):
	def createProduct(self, type):
		product = None;

		if type == "A":
			product = ProductAStoreX()
		elif type == "B":
			product = ProductBStoreX()
		return product

class StoreY(Store):
	def createProduct(self, type):
		product = None;

		if type == "A":
			product = ProductAStoreY()
		elif type == "B":
			product = ProductBStoreY()
		return product

def main():
	print("Enter a product type: ")
	type = input()

	store1 = StoreX()
	store1.order("Peter", type)
	store2 = StoreY()
	store2.order("Lily", type)
```

The Example of Factory Method

Case Study: File Parser

- In this example, we will focus on two popular human-readable formats: XML and JSON. These two file formats make data exchange, inspection, and modification much easier.
- We have some student data stored in an XML and a JSON file, and we want to parse them and retrieve some information. At the same time, we would like to make file format transparent to the client. We will use the Factory Method to solve this problem.



Case Study: File Parser

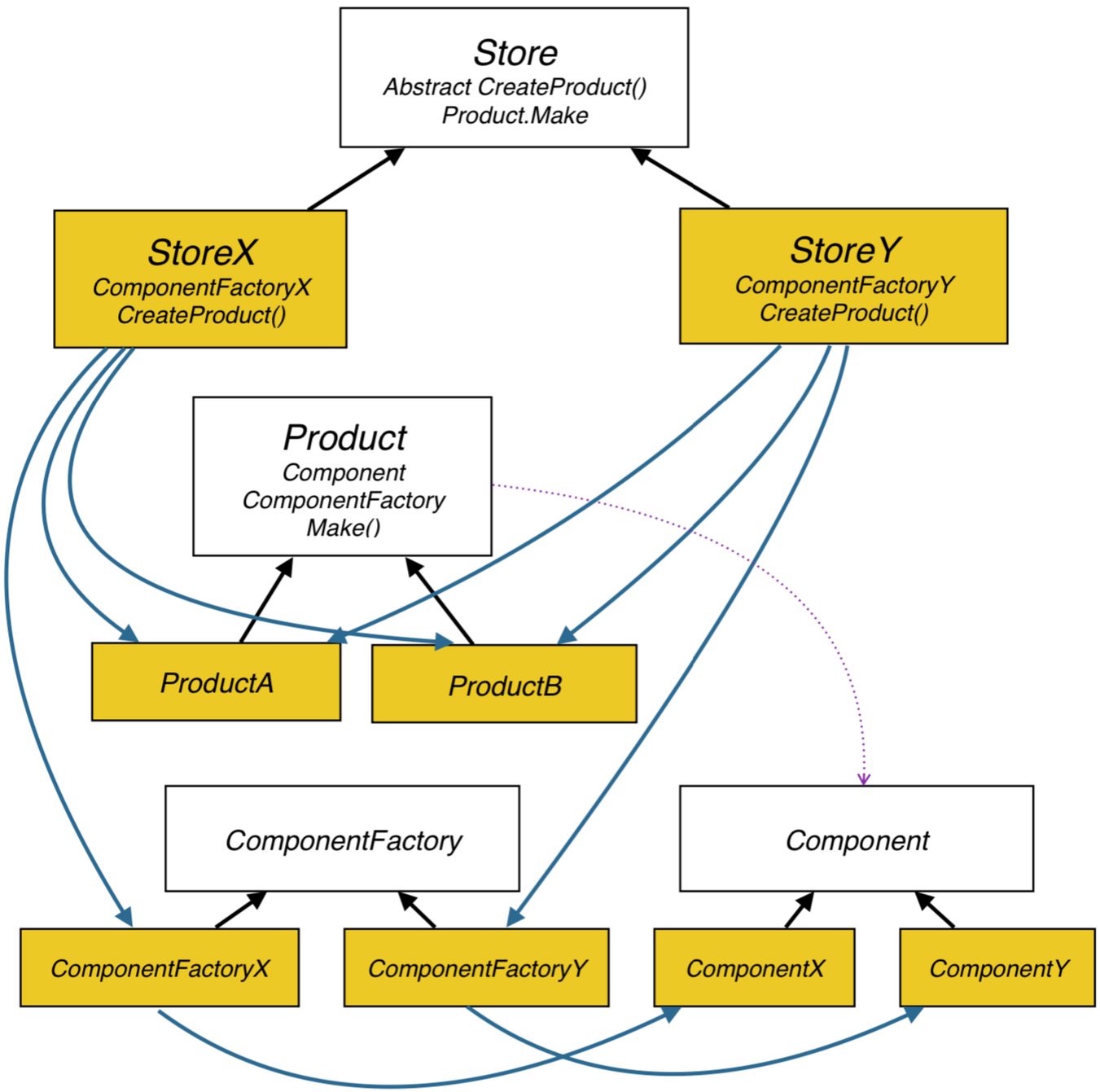
```
class ParserFactory:
    def createParser(self, filepath):
        if filepath.endswith('json'):
            parser = JSONParser
        elif filepath.endswith('xml'):
            parser = XMLParser
        else:
            raise ValueError('Cannot create a parser for {}'.format(filepath))
        return parser(filepath)


def main():
    factory = ParserFactory()
    try:
        filepath = input("Enter the file name: ")
        parser = factory.createParser(filepath)
        for student in parser.students:
            student.display()
    except ValueError as ve:
        print(ve)

if __name__ == '__main__':
    main()
```

```
Enter the country code: US
03-17-2021
$899.99

Enter the country code: FR
17/03/2021
899€99
```

```
class JSONParser(Parser):
    def __init__(self, filepath):
        self.data = dict()
        with open(filepath, mode='r', encoding='utf-8') as f:
            self.data = json.load(f)

    @property
    def getStudents(self):
        list = []
        students = self.data
        for student in students:
            obj = Student(student['firstName'],
                student['lastName'],
                student['age'],
                student['gpa'],
                student['gender']
            )
            list.append(obj)

        return list

class XMLParser(Parser):
    def __init__(self, filepath):
        self.tree = etree.parse(filepath)

    @property
    def getStudents(self):
        list = []
        students = self.tree.findall(".//{}".format('student'))
        for student in students:
            obj = Student(student.find('firstName').text,
                student.find('lastName').text,
                int(student.find('age').text),
                float(student.find('gpa').text),
                student.find('gender').text
            )
            list.append(obj)

        return list
```

Abstract Factory Pattern

- Abstract Factory pattern is almost similar to Factory Pattern is considered as another layer of abstraction over factory pattern. Abstract Factory patterns work as a super-factory which creates other factories.
- Abstract factory pattern implementation provides us a framework that allows us to create objects that follow a general pattern. So at runtime, abstract factory is coupled with any desired concrete factory which can create objects of desired type.



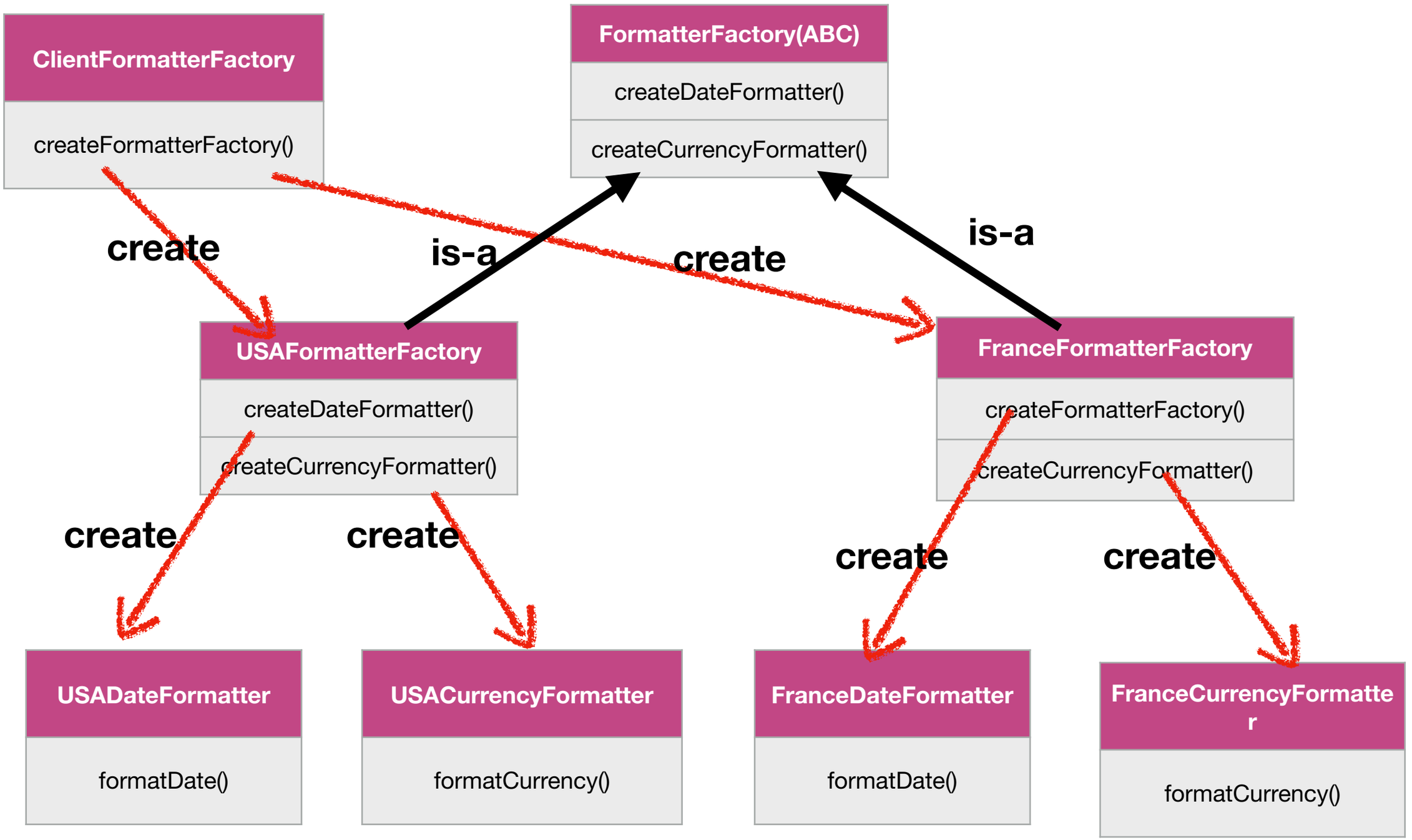
The Example of Abstract Factory

```
from abc import ABC, abstractmethod
import enum

class Component(ABC):
        @abstractmethod
        def getComponent(self):
                pass

class ComponentStoreX(Component):
        def getComponent(self):
                return "Component for Store X"

class ComponentStoreY(Component):
        def getComponent(self):
           return "Component for Store Y"

class ComponentFactory():
        def createComponent(self):
                pass

class ComponentFactoryStoreX(ComponentFactory):
        def createComponent(self):
                return ComponentStoreX()

class ComponentFactoryStoreY(ComponentFactory):
        def createComponent(self):
                return ComponentStoreY()

class Product(ABC):
        def __init__(self):
                self.factory = None
                self.component = None

        def make(self, type):
                return "Making a " + self.getName() + " with " + self.component.getComponent()

        @abstractmethod
        def getName(self):
                pass

class ProductA(Product):
        def __init__(self, factory):
                self.factory = factory
                self.component = factory.createComponent()

        def getName(self):
                return "Product A"

class ProductB(Product):
        def __init__(self, factory):
                self.factory = factory
                self.component = factory.createComponent()

        def getName(self):
                return "Product B"
```

```
class Store(ABC):
        def __init__(self):
                self.factory = None

        def order(self, customer, type):
                product = self.createProduct(type)
                print(product.make(type))
                print(customer, "ordered", product.getName())
                return product

        @abstractmethod
        def createProduct(self, type):
                passs

class StoreX(Store):
        def createProduct(self, type):
                factory = ComponentFactoryStoreX()
                product = None;

                if type == "A":
                        product = ProductA(factory)
                elif type == "B":
                        product = ProductB(factory)
                return product

class StoreY(Store):
        def createProduct(self, type):
                factory = ComponentFactoryStoreY()
                product = None;

                if type == "A":
                        product = ProductA(factory)
                elif type == "B":
                        product = ProductB(factory)
                return product

def main():
        print("Enter a product type: ")
        type = input()

        store1 = StoreX()
        store1.order("Peter", type)
        store2 = StoreY()
        store2.order("Lily", type)

main()
```

Case Study: Formatters

- In this example, we'll create a set of formatters that depend on a specific locale and help us format dates and currencies



Case Study: Formatters

```
class FormatterFactory(ABC):
    @abstractmethod
    def createDateFormatter(self):
        pass
    @abstractmethod
    def createCurrencyFormatter(self):
        pass

class USAFormatterFactory(FormatterFactory):
    def createDateFormatter(self):
        return USADateFormatter()
    def createCurrencyFormatter(self):
        return USACurrencyFormatter()

class FranceFormatterFactory(FormatterFactory):
    def createDateFormatter(self):
        return FranceDateFormatter()
    def createCurrencyFormatter(self):
        return FranceCurrencyFormatter()

class ClientFormatterFactory:
    def createFormatterFactory(self, countryCode):
        factoryMap = {
            "US": USAFormatterFactory,
            "FR": FranceFormatterFactory}
        formatterFactory = factoryMap.get(countryCode)()
        return formatterFactory

def main():
    countryCode = input("Enter the country code: ")
    formatterFactory = ClientFormatterFactory().createFormatterFactory(countryCode)
    formatter = formatterFactory.createDateFormatter()
    print(formatter.formatDate(2021, 3, 17))
    formatter = formatterFactory.createCurrencyFormatter()
    print(formatter.formatCurrency(899, 99))

main()
```

```
Enter the country code: US
03-17-2021
$899.99

Enter the country code: FR
17/03/2021
899€99
```

Summary

- All factories encapsulates object creation.
- Factory method relies on inheritance: object creation is delegated to subclasses which implement the factory method to create objects.
- To allow a class to defer instantiation to its subclass.
- Abstract Factory relies on object composition: object creation is implemented in methods exposed in the factory interface.
- To create families of related objects without having to depend on their concrete classes
- The Dependency Inversion Principle guides us to avoid dependencies on concrete types and to achieve abstractions.