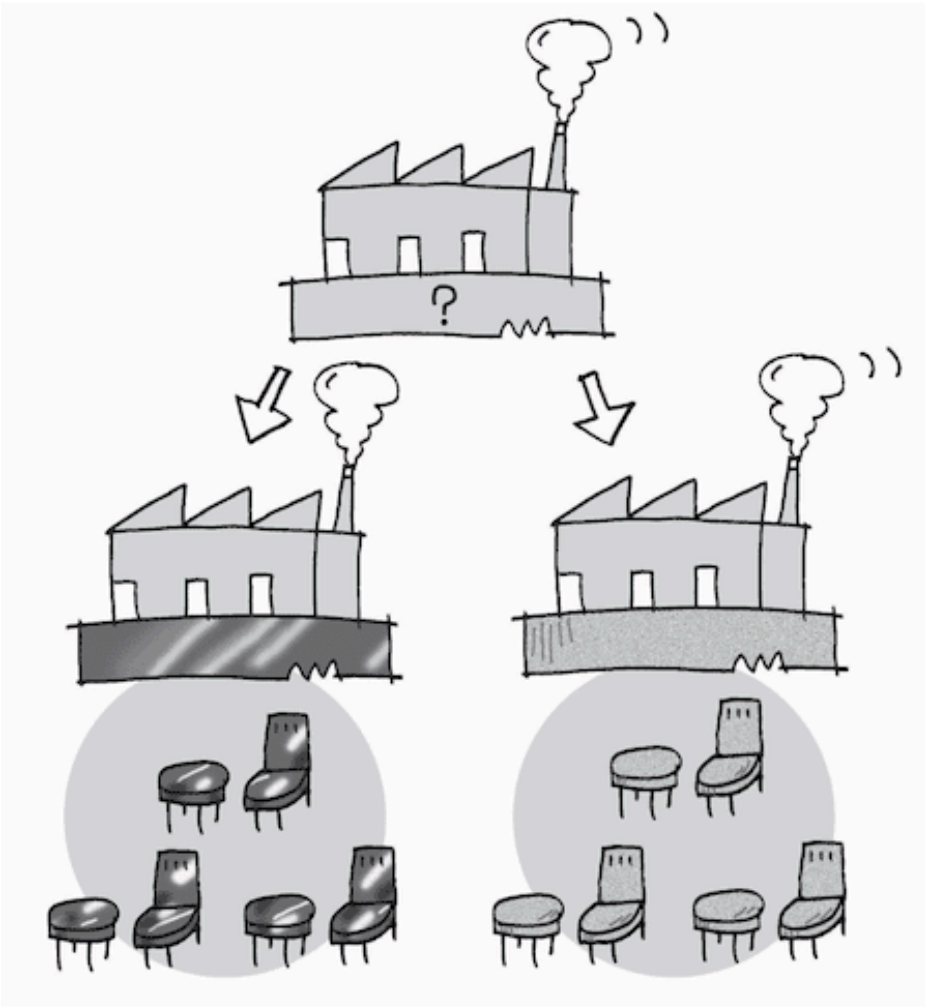
Abstract Factory Pattern

Combine Related Parts to Make a Product



Abstract Factory Pattern

LG and Samsung both follow the same **manufacturing interface** (make TV + controller),
but each produces its own **product family**:

- LG → QLED TV + LG controller
- Samsung → Quantum TV + Samsung controller

The Problem

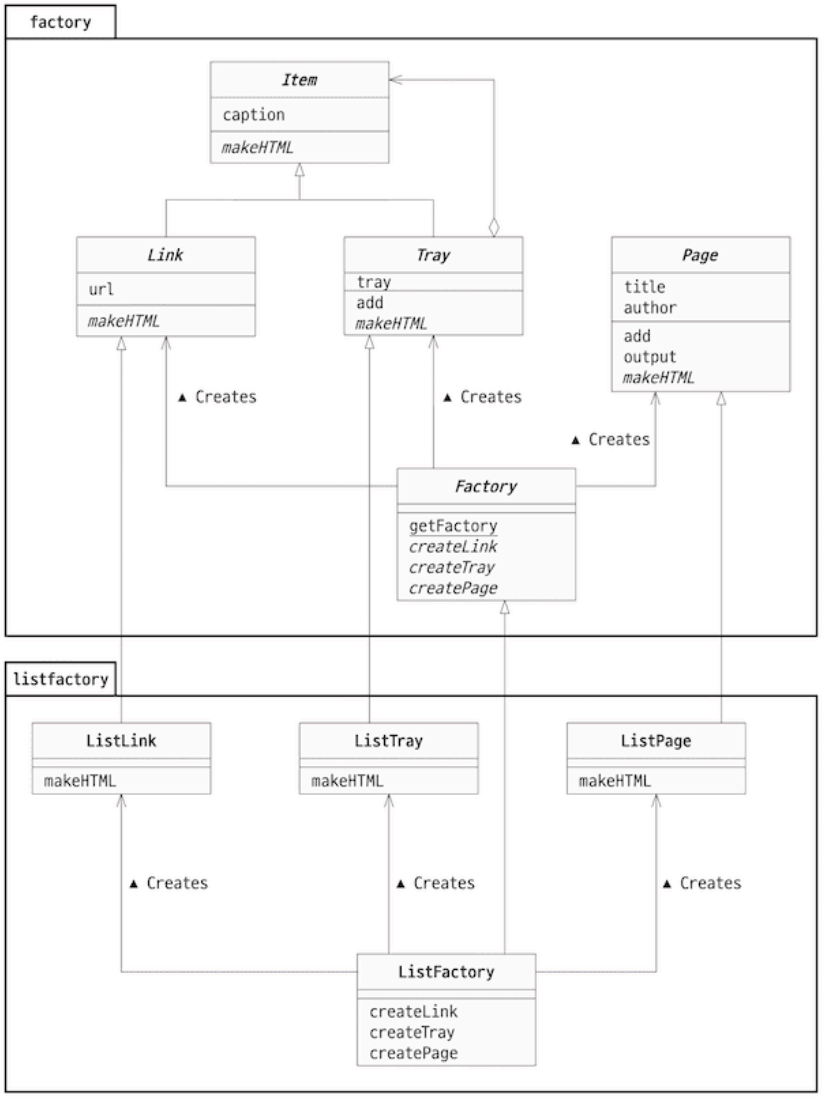
- We need to create **HTML pages** with different styling approaches.
- We want to create related products (**Link**, **Tray**, **Page**) that work together.
- Different **factories** create products in different ways, but the GUI structure should remain the same.

The challenge: how to create **families of related objects** without specifying their **concrete classes** and ensure they work together?

The *Abstract Factory* as the Solution

- We have an abstraction *Abstract Factory* that creates families of related *products*.
- We do not need to `know` about the specific **factory implementation**, we only need to `use` the *abstract interface* to create consistent products.

The Solution (Design)



Step 1: Understand the Players

In this design, we have players:

- *Abstract Factory*
    - **Concrete Factory** (ListFactory, DivFactory)
- *Abstract Product* (Link, Tray, Page)
    - **Concrete Product** (ListLink, DivLink, etc.)

We have the code that uses the factory:

- **Client**

Step 2: Understand abstractions

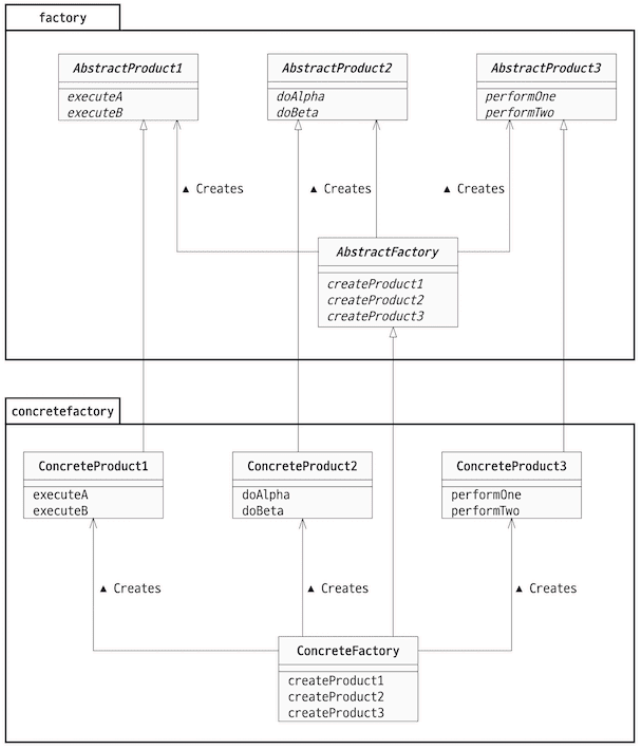
- We have an *Abstract Factory* that `creates` families of related *products*.
    - *Abstract Factory*
    - *Abstract Product* (Link, Tray, Page)
- In short, we use the *factory* to create related products that work together.
    - All products from the same factory are `compatible`.

Step 3: Understand concretion

- We have specific/concrete **ListFactory** and **DivFactory** that create specific/concrete HTML **products** in different styles.
    - **ConcreteFactory** (ListFactory, DivFactory)
    - **ConcreteProduct** (ListLink, DivLink, ListTray, DivTray, etc.)

The Design

We have an abstract factory/product, and a matching concrete factory/product.



Code

- Main Method
- Factory Classes
- Product Classes

Main Method

```
def main():
    # Demo with ListFactory
    print("\n1. Creating page with ListFactory...")
    list_factory = ListFactory()
    list_page = create_content(list_factory)
    list_page.printit()

    # Demo with DivFactory
    print("\n2. Creating page with DivFactory...")
    div_factory = DivFactory()
    div_page = create_content(div_factory)
    div_page.printit()
```

Step 1: Get the appropriate factory

```
list_factory = ListFactory()
```

Step 2: Create related products using the factory

In the create_content() method:

```
    blog1 = factory.create_link("Python Tutorial", "https://python.org")
    blog_tray = factory.create_tray("Learning Resources")
    page = factory.create_page("My Bookmarks", "Student")
```

- All products created by the same factory are `compatible` and work together.
- The **client** doesn't know which concrete factory is being used.

Step 3: Compose products into a complete structure

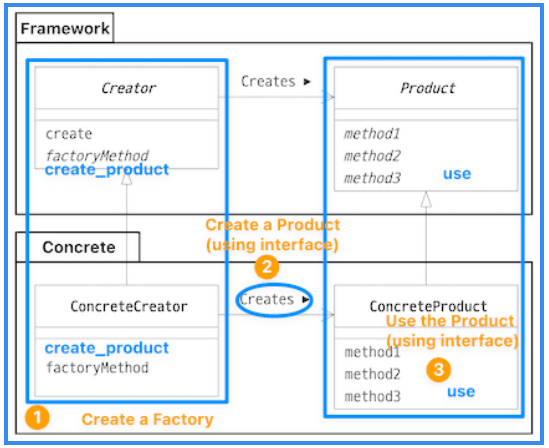
```
    blog_tray.add(blog1)
    blog_tray.add(blog2)

    page.add(blog_tray)
```

- Products from the same factory family can be `composed` together.
- The final result is a consistent HTML page with unified styling.

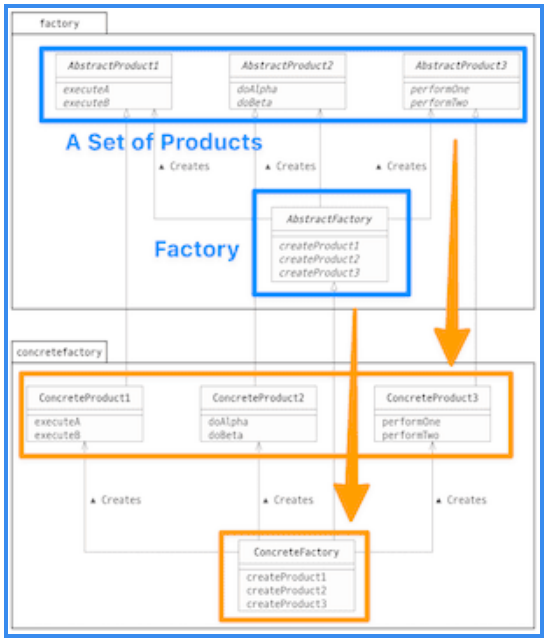
Discussion

What's the difference between the FactoryMethod pattern and the AbstractFactory pattern?

The FactoryMethod pattern is about creating a product with different configurations.



The AbstractFactory pattern is about creating a set of related products.



DIP

The client depends on abstractions (AbstractFactory and AbstractProduct) rather than concrete implementations - **Dependency Inversion Principle**

Key Benefits

1. **Consistency**: All products from the same factory work together
2. **Flexibility**: Easy to switch between different product families
3. **Isolation**: Client code is isolated from concrete classes
4. **Extensibility**: Easy to add new product families

Key Drawback

Abstract Factory pattern is **Difficult to extend**:

Adding a new type of product requires changing the AbstractFactory interface and all concrete factories.

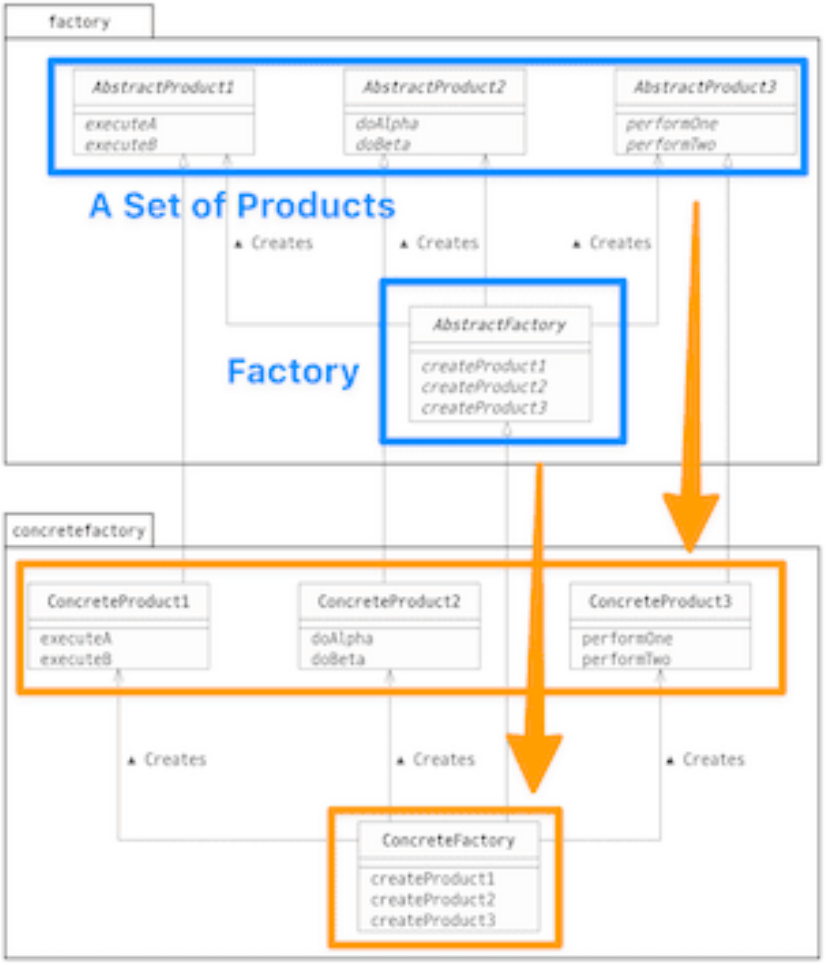
When to Use Abstract Factory

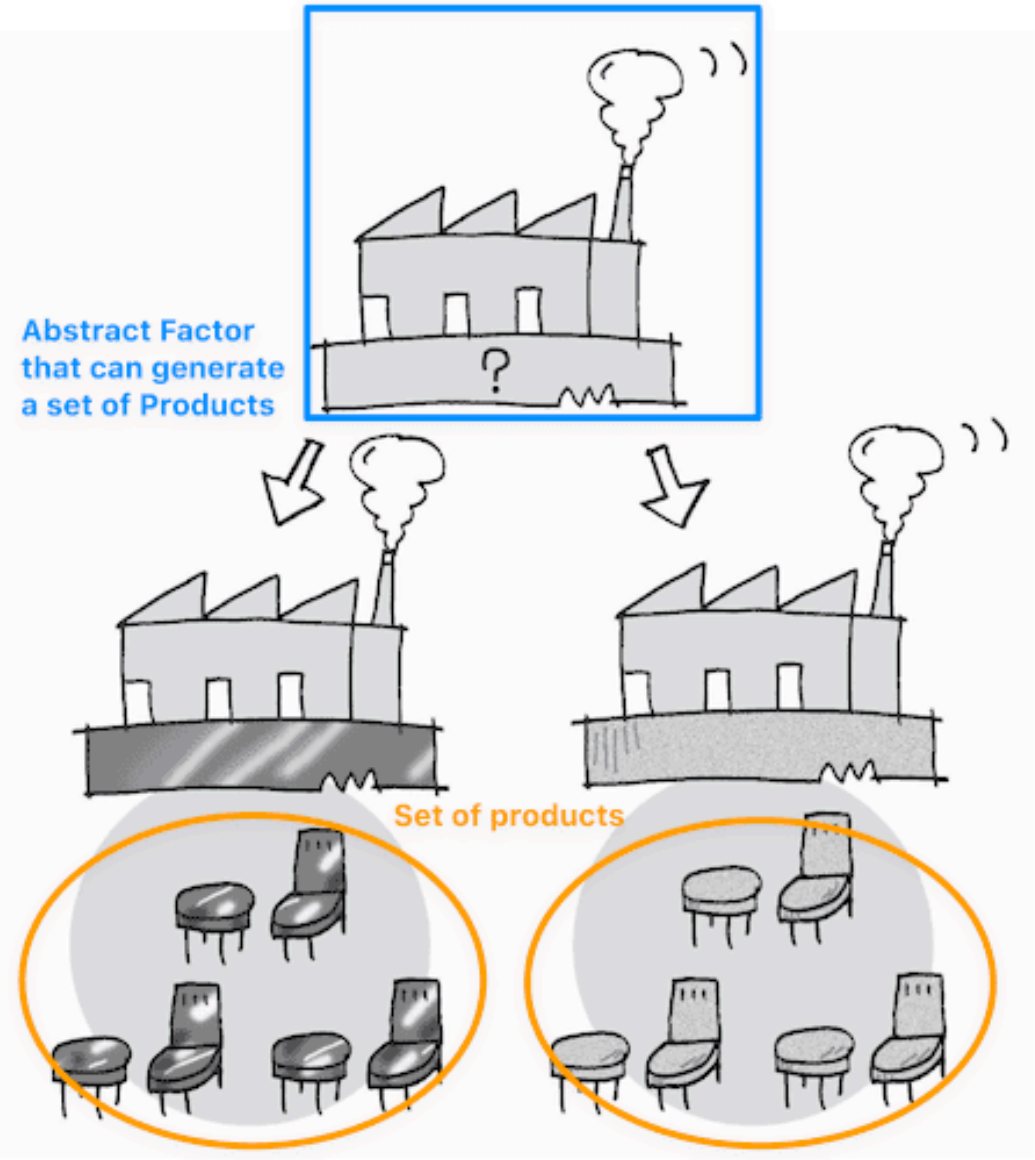
- When you need to create families of related products
- When you want to ensure products work together
- When you need to support multiple product lines
- When you want to hide concrete classes from clients

Related Patterns

- **Factory Method**: Abstract Factory uses Factory Method to create products
- **Singleton**: Factory instances are often Singletons
- **Prototype**: Products can be created by cloning prototypes

UML


factory
AbstractProduct1
executeA
executeB
AbstractProduct2
doAlpha
doBeta
AbstractProduct3
performOne
performTwo
A Set of Products
Creates
Creates
Creates
AbstractFactory
createProduct1
createProduct2
createProduct3
Factory
concretefactory
ConcreteProduct1
executeA
executeB
ConcreteProduct2
doAlpha
doBeta
ConcreteProduct3
performOne
performTwo
Creates
Creates
Creates
ConcreteFactory
createProduct1
createProduct2
createProduct3


Abstract Factor
that can generate
a set of Products
?
Set of products