

OVERVIEW Design Patterns and the Abstract Factory

Design patterns are recurring solutions to object-oriented design problems in a given context, smaller in scale than architectural patterns. They are classified by purpose (creational, structural, behavioural) and by scope (class or object). Creational patterns control how objects are created. The first one, Abstract Factory, gives a client one interface for creating a whole family of related products (dark-theme widgets, advanced computer parts) without naming their concrete classes, so switching families means swapping one factory object.

Purpose and scope

Creational = creating objects; structural = composing structures; behavioural = interaction and responsibilities. Class scope = design time; object scope = run time.

Five creational patterns

Abstract Factory, Factory Method, Builder, Prototype, Singleton.

Abstract Factory roles

AbstractFactory (one create method per product), ConcreteFactory per family, AbstractProduct per product type, ConcreteProducts, and a Client that sees only the abstract types.

Trade-off

Adding a new family is easy (a new factory and products, OCP); adding a new kind of product touches every factory. Many classes.

HOW THE LECTURE IS ORGANIZED

Slides 1–7 Design patterns: definition, purpose and scope, documentation, creational patterns

Slides 8–17 Abstract Factory: intent, the theme example, structure, adding families and products

Slides 18–31 Computer-store example: products, factories, client and main program

Slides 32–39 Step-by-step summary, pros and cons, the car-factory exercise, recap

EXAM FOCUS

Final 252 Q6 examined creational patterns through Builder and Singleton (Lectures 2–3). For Abstract Factory expect a design question: given a scenario with product families (themes, car brands, computer grades), name the pattern, draw the class diagram with factory and product interfaces, and explain how a new family is added.

HOW TO READ THIS PAGE

Every original slide is shown unchanged. **Yellow marks** on a slide point to a numbered note: hover or tap a mark to jump to its note, and hover a note to see its marks. Questions tagged **Past exam** appeared on a real paper; **Prediction** questions are likely angles, not confirmed.

Software Design and Architecture

[Creational Design Patterns] – Chapter 06, L01

NOTES

- 1 Chapter 6 starts **design patterns**, the detailed-design counterpart of Chapter 4's architectural patterns. This lecture: what design patterns are, how they are classified, and the first creational pattern, **Abstract Factory**.

SLIDE 2 / 39 Lecture Outlines

Lecture Outlines

- **Patterns in Detailed Design**
 - ✓ Architectural vs. Design Patterns.
- **Classification of Design Patterns**
 - ✓ Purpose
 - ✓ Scope
- **Documenting Design Patterns**
- **Creational Design Patterns**
 - ✓ **Abstract Factory**
 - ✓ Computer Store Example
- **What's next...**

NOTES

- 1 First, how design patterns differ from architectural patterns.
- 2 Then how they are classified (purpose and scope) and documented.
- 3 Then the creational patterns, starting with **Abstract Factory** and its computer-store example.

SLIDE 3 / 39 Patterns in Detailed Design

Patterns in Detailed Design

- **Design patterns** are recurring solutions to object-oriented design problems in a particular context. They are different than architectural patterns/styles!
- **Extra: Design patterns are reusable solutions to common problems**
They provide templates for how to solve problems efficiently and consistently, helping developers avoid reinventing the wheel.

In which way they are different?!!

NOTES

- 1 **Design patterns** are recurring solutions to object-oriented design problems in a particular context. They give templates that solve problems efficiently and consistently, so developers don't reinvent the wheel.
- 2 They are **different from architectural patterns** (Chapter 4, slide 11 of Lecture 1): architectural patterns fix the system-wide structure; design patterns work at the level of classes and objects inside a subsystem and don't change the overall structure.

Classifications of Design Patterns

- Design patterns can be classified based on:
 - ✓ **Purpose**
 - ✓ **Scope**
- The **purpose** of a design pattern identifies the **essence** of the pattern. The three types of purposes used for classification are:
 1. **Creational**: Patterns that deal with creation of objects.
 2. **Structural**: Patterns that deal with creation of structures to form the existing objects.
 3. **Behavioral**: Patterns that deal with how classes interact, the variation of behavior, and the assignment of responsibility between objects.

Prince Sultan University | Slides prepared from "Software Engineering Design: Theory and Practice" book and slides

5

NOTES

- 1 Patterns are classified by **purpose** (what kind of problem they solve) and **scope** (next slide).
- 2 **Creational**: deal with the creation of objects (this chapter).
- 3 **Structural**: deal with composing objects and classes into larger structures (Chapter 7: Adapter, Composite...).
- 4 **Behavioural**: deal with how classes interact, how behaviour varies, and how responsibilities are assigned between objects (Chapter 7: Observer...).

EXAM ANGLE

Prediction

Q1

Patterns that deal with how classes interact and how responsibilities are assigned between objects are:

- A. Behavioral
- B. Creational
- C. Structural
- D. Architectural

Answer: A. Behavioral

Classifications of Design Patterns

- The **scope criterion** captures whether a design pattern primarily applies to **classes** (during design time) or **objects** (during run-time).

<http://www.cs.unc.edu/~stotts/GOF/hires/chap1fso.htm>

Extra:

1. Class-level patterns:

Patterns that apply at the **class level**, focusing on relationships between classes during design time.

2. Object-level patterns:

Patterns that apply at the **object level**, focusing on relationships between objects at runtime.

In short, class-level patterns deal with class structures, while object-level patterns focus on interactions during execution.

Scope of pattern	Type of pattern		
	Creational	Structural	Behavioral
Class-level patterns	Factory method	Adapter	Interpreter Template Method
Object-level patterns	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Façade Proxy	Chain of Responsibility Command Iterator Mediator Memento Flyweight Observer State Strategy Visitor

6

NOTES

- 1 The **scope** criterion says whether a pattern applies mainly to classes or to objects.
- 2 **Class-level** patterns deal with relationships between classes, fixed at design time through inheritance. Examples in the table: Factory Method (creational), Adapter (structural), Interpreter and Template Method (behavioural).
- 3 **Object-level** patterns deal with relationships between objects, set up at run time through composition. Most patterns are here: Abstract Factory, Builder, Prototype, Singleton (creational); Adapter, Bridge, Composite, Decorator, Façade, Proxy (structural); Chain of Responsibility, Command, Iterator, Mediator, Memento, Flyweight, Observer, State, Strategy, Visitor (behavioural).

EXAM ANGLE

Prediction

Q1

Classify Abstract Factory, Composite and Observer by purpose and scope.

Answer: Abstract Factory: creational, object scope. Composite: structural, object scope. Observer: behavioural, object scope.

Documenting Design Patterns

Extra:

- This slide explains how **design patterns are documented**.
- Each pattern is described using several key categories such as **intent**, **motivation**, **applicability**, **structure**, **participants**, and more.
- These categories help provide a clear understanding of what the pattern does, when to use it, how it works, and its effects on the system.
- you don't need to focus on presenting this extensive documentation for each pattern.

Note:

In this course, we're **not** concerned with presenting this extensive documentation for each pattern

Category	Description
Name and Classification	The unique pattern name that reflects the essence of the patterns and its classification.
Intent	Describes the purpose of the pattern in such way that it is clear what types of design problems the pattern solves, what the pattern does, its rationale and intent.
Also Known As	A list of alternate well-known names for the pattern.
Motivation	En example scenario that serves as motivation for the application of the pattern.
Applicability	Describes the situations, or design problems, that lend themselves for the application of the design pattern. Provides examples of poor designs that can benefit from the pattern and ways for identifying these situations.
Structure	Provides a structural (e.g., UML class diagram) view of the design pattern.
Participants	List the classes and objects required in the design pattern and their responsibilities.
Collaborations	Provides information about how the participants work together to carry out their responsibilities.
Consequences	Describes the effects of the design pattern, good or bad, on the software solution.
Implementation	Provides information and techniques for successfully implementing the design pattern.
Sample Code	Provides sample code that demonstrates how to implement the design pattern in different programming languages.
Known Uses	Provides examples of real systems that employ the design pattern.
Related Patterns	Provides information about other design patterns that are related, or that can be used in combination with the design pattern.

NOTES

- Each pattern is documented with the same categories, so patterns can be compared and looked up.
- The categories:** name and classification, intent (what problem it solves), also known as, motivation (an example scenario), applicability (when to use it), structure (UML class diagram), participants, collaborations, consequences (good and bad effects), implementation, sample code, known uses, related patterns.
- Note on the slide: this course is **not** concerned with presenting the full documentation for each pattern; focus on intent, structure and participants.

Creational Design Patterns

- Creational design patterns **abstract and control** the way objects are created in software applications.
 - ✓ They do so by specifying a **common creational interface**.
- Examples of creational patterns include:
 - ✓ **The Abstract Factory**
 - ✓ The Factory Method
 - ✓ The Builder
 - ✓ The Prototype
 - ✓ **The Singleton**

Prince Sultan University | Slides prepared from "Software Engineering Design: Theory and Practice" book and slides

8

NOTES

- 1 **Creational** patterns abstract and control how objects are created, by giving clients a **common creational interface** instead of scattered `new ConcreteClass()` calls.
- 2 The five: **Abstract Factory, Factory Method, Builder, Prototype, Singleton**. This chapter covers Abstract Factory, Factory Method, Builder and Singleton.

EXAM ANGLE

Prediction

Q1

Which of these is NOT a creational pattern?

- A. Composite
- B. Builder
- C. Singleton
- D. Prototype

Answer: A. Composite

The Abstract Factory Pattern

- The **Abstract Factory** is an **object-creational design pattern** intended to manage and encapsulate the creation of a set of objects that conceptually belong together and that represent a specific **family of products**.
- The **intent** of the Abstract Factory is to:
 - ✓ Provide an interface for creating families of related objects without specifying their concrete classes.
- Like all creational patterns, Abstract Factory is composed of **creator classes and product classes**.
 - ✓ As it will be seen, some creational patterns fuse the creator and product into one class.

NOTES

- 1 The **Abstract Factory** is an object-creational pattern that manages and encapsulates the creation of a set of objects that belong together: a **family of products**.
- 2 Its **intent**: provide an interface for **creating families of related objects without specifying their concrete classes**.
- 3 Like all creational patterns it has **creator** classes (the factories) and **product** classes. (Some creational patterns, such as Singleton, fuse creator and product into one class.)

EXAM ANGLE

Prediction

Q1

What is the intent of the Abstract Factory pattern?

- A. Provide an interface for creating families of related objects without specifying their concrete classes
- B. Ensure a class has exactly one instance
- C. Construct a complex object step by step
- D. Create objects by copying a prototype

Answer: A. Provide an interface for creating families of related objects without specifying their concrete classes

The Abstract Factory Pattern

- The **Abstract Factory** is an **object-creational design pattern** intended to manage and encapsulate the creation of a set of objects that conceptually belong together and that represent a specific family of products.
- The **intent** of the Abstract Factory is to:
 - ✓ Provide an interface for creating families of related objects without specifying their concrete classes.
- Like all creational patterns, Abstract Factory is composed of **creator** classes and **product** classes.
 - ✓ As it will be seen, some creational patterns fuse the creator and product into one class.
- **Extra: Benefits:**
 - ✓ **Encapsulation:** The client **DOES** know the abstract product names (Button, Textbox). The client **does NOT** know the concrete product class names (DarkButton, LightTextbox, etc.).
 - ✓ **Flexibility:** Easy to add new families by creating a new factory.

10

NOTES

- 1 The same definition, plus the two benefits. **Encapsulation:** the client knows the abstract product names (Button, Textbox) but **not** the concrete classes (DarkButton, LightTextbox).
- 2 **Flexibility:** a new family is added by creating a new factory, without changing the client.

SLIDE 10 / 39 Extra: Question – Dark and Light themes

Extra

➤ Question:

Imagine you are designing a computer program that must support two different themes: **Dark Theme** and **Light Theme**.

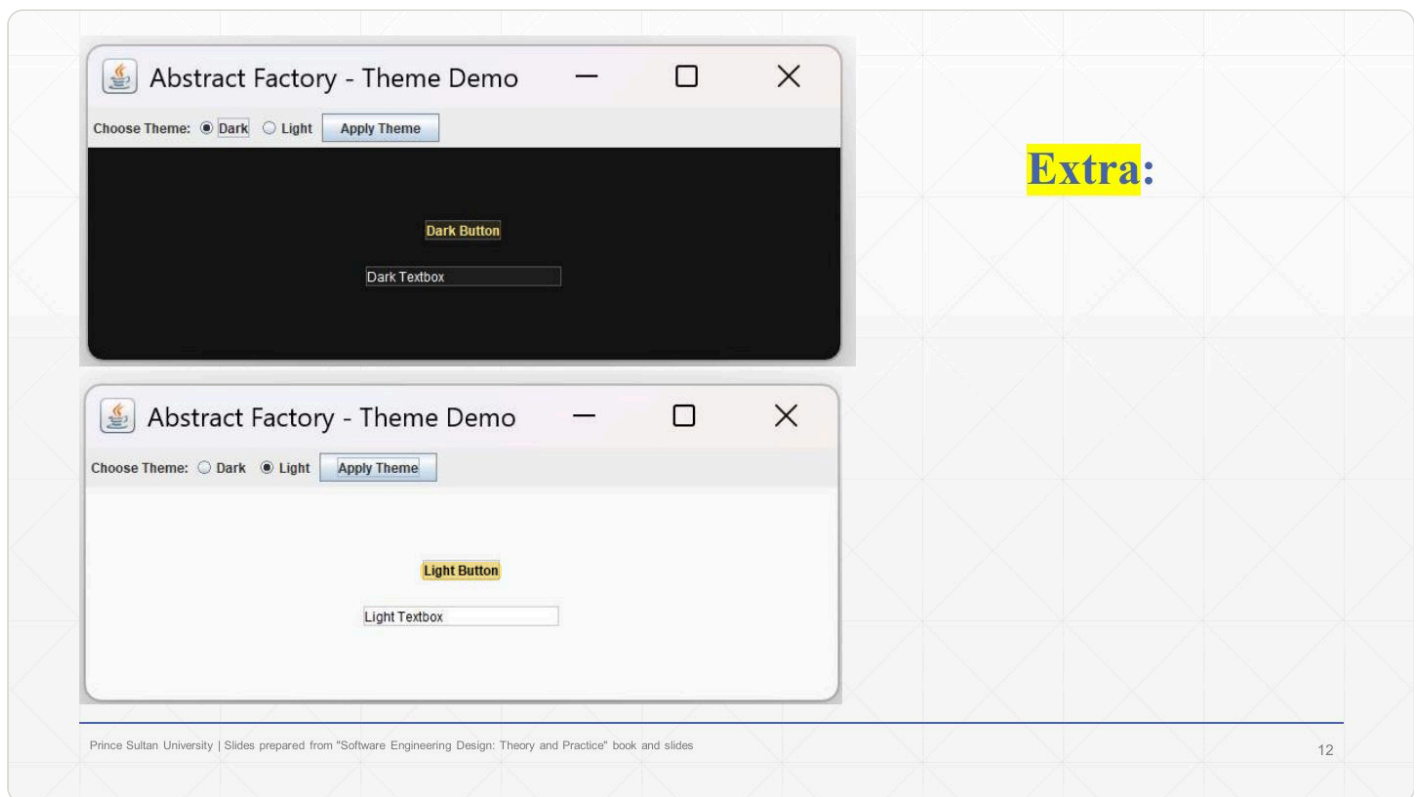
- In the Dark Theme, the **Button** is black color and the **TextField** is black color.
- In the Light Theme, the **Button** is white color and the **TextField** is white color.
- Instead of writing separate code for Dark and Light everywhere in the program, you decide to use the **Abstract Factory design pattern**.
- Define a common factory interface (**GUIFactory**) that specifies methods for creating a Button and a TextField.
- Explain how the **DarkThemeFactory** and **LightThemeFactory** would implement this interface.
- Describe how the client program (the main class) can use the factory without knowing whether it is working with Dark Theme or Light Theme.
- Solu:
 - ✓ a UI library with two themes: **Dark Theme** and **Light Theme**.
 - ✓ Products = **Button**, **Textbox**.
 - ✓ Factories = **DarkThemeFactory**, **LightThemeFactory**.
 - ✓ Client just asks the factory for a Button/Textbox → doesn't care which concrete class (**DarkButton**, **LightButton**) is returned.
 - ✓ ☞ If tomorrow we add a **Blue Theme**, just create **BlueThemeFactory**, **BlueButton**, **BlueTextbox** → no changes to client code.

➤ **So: Abstract Factory = "One stop shop for creating related objects (family), while hiding the details of their concrete classes."**

NOTES

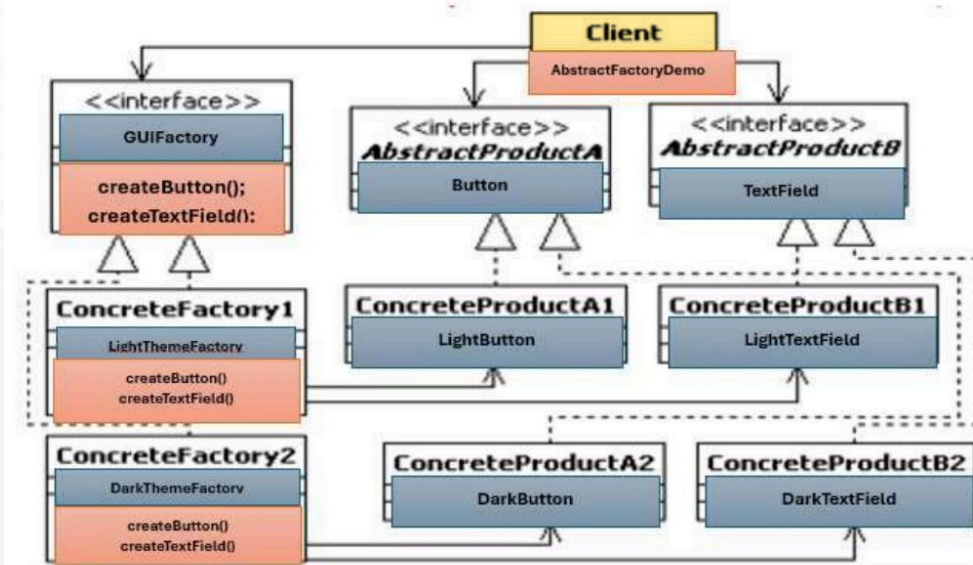
- 1 The exercise: a program with a **Dark** theme (black button and text field) and a **Light** theme (white ones). Instead of `if (dark)` checks everywhere, use Abstract Factory.
- 2 The factory interface **GUIFactory** declares `createButton()` and `createTextField()`.
- 3 **DarkThemeFactory** returns a `DarkButton` and `DarkTextField`; **LightThemeFactory** returns the light versions. The client asks its factory for a `Button` and a `TextField` and never knows which concrete class came back.
- 4 Adding a Blue theme means adding `BlueThemeFactory`, `BlueButton` and `BlueTextbox`, with no change to the client. In one line: Abstract Factory is a "one-stop shop" for creating a related family of objects while hiding their concrete classes.

SLIDE 11 / 39 Extra: The theme demo running



NOTES

- 1 The theme example running. Choosing **Dark** and pressing Apply Theme swaps in the `DarkThemeFactory`, and the window shows a dark button and text box; choosing **Light** swaps in the `LightThemeFactory`. The client code that builds the window is identical in both cases; only the factory object changed.



NOTES

- 1 The class diagram mapped onto the pattern's role names. «interface» **GUIFactory** (AbstractFactory) declares `createButton()` and `createTextField()`.
- 2 «interface» **Button** (AbstractProductA) and **TextField** (AbstractProductB) are the product types the client uses.
- 3 **LightThemeFactory** (ConcreteFactory1) creates **LightButton** and **LightTextField** (ConcreteProductA1/B1); **DarkThemeFactory** (ConcreteFactory2) creates **DarkButton** and **DarkTextField** (A2/B2). The hollow triangles are realizations; the arrows from factories to products show what each creates.
- 4 The **Client** (AbstractFactoryDemo) depends only on the three interfaces.

Step 1: Define Abstract Factory

Step 2: Implement Concrete Factories

Step 3: Define Abstract Products

```

java

interface GUIFactory {
    Button createButton();
    TextField createTextField();
}

```

```

java

class DarkThemeFactory implements GUIFactory {
    public Button createButton() {
        return new DarkButton();
    }

    public TextField createTextField() {
        return new DarkTextField();
    }
}

class LightThemeFactory implements GUIFactory {
    public Button createButton() {
        return new LightButton();
    }

    public TextField createTextField() {
        return new LightTextField();
    }
}

```

```

java

interface Button {
    void draw();
}

interface TextField {
    void display();
}

```

Extra:

14

NOTES

1 The code, steps 1–3:

CODE	WHAT IT DOES
<pre>interface GUIFactory { Button createButton(); TextField createTextField(); }</pre>	Step 1: the abstract factory, one create method per product type.
<pre>class DarkThemeFactory implements GUIFactory {</pre>	Step 2: a concrete factory for the dark family...
<pre> public Button createButton() { return new DarkButton(); }</pre>	...returns the dark button, typed as the abstract Button.
<pre> public TextField createTextField() { return new DarkTextField(); } }</pre>	...and the matching dark text field.
<pre>class LightThemeFactory implements GUIFactory { ... new LightButton() ... new LightTextField() ... }</pre>	The light family, same shape.
<pre>interface Button { void draw(); }</pre>	Step 3: the abstract products.
<pre>interface TextField { void display(); }</pre>	

2 The only place `new DarkButton()` appears is inside `DarkThemeFactory`. That is the point of the pattern.

◆ **Step 4: Create Concrete Products**

```

java

class DarkButton implements Button {
    public void draw() {
        System.out.println("Drawing Dark Button");
    }
}

class LightButton implements Button {
    public void draw() {
        System.out.println("Drawing Light Button");
    }
}

class DarkTextField implements TextField {
    public void display() {
        System.out.println("Showing Dark TextField");
    }
}

class LightTextField implements TextField {
    public void display() {
        System.out.println("Showing Light TextField");
    }
}

```

◆ **Step 5: Client (Main Program)**

```

java

public class AbstractFactoryDemo {
    public static void main(String[] args) {

        // ===== Using Light Theme =====
        System.out.println("---- Light Theme ----");
        GUIFactory lightFactory = new LightThemeFactory();

        Button lightBtn = lightFactory.createButton();
        TextField lightTxt = lightFactory.createTextField();

        lightBtn.draw();
        lightTxt.display();

        // ===== Using Dark Theme =====
        System.out.println("\n---- Dark Theme ----");
        GUIFactory darkFactory = new DarkThemeFactory();

        Button darkBtn = darkFactory.createButton();
        TextField darkTxt = darkFactory.createTextField();

        darkBtn.draw();
        darkTxt.display();
    }
}

```

15

NOTES

1 Steps 4–5:

CODE

WHAT IT DOES

```
class DarkButton implements Button {
    public void draw() { ... "Drawing Dark
    Button" ... } }
```

Step 4: concrete products, one per family per product type.

```
class LightTextField implements
TextField { public void display() {
... "Showing Light TextField" ... } }
```

(LightButton and DarkTextField are written the same way.)

```
GUIFactory lightFactory = new
LightThemeFactory();
```

Step 5, the client: pick a family once...

```
Button lightBtn =
lightFactory.createButton(); TextField
lightTxt =
lightFactory.createTextField();
```

...ask that factory for the parts, typed as abstract products...

```
lightBtn.draw(); lightTxt.display();
```

...and use them. Prints "Drawing Light Button", "Showing Light TextField".

```
GUIFactory darkFactory = new
DarkThemeFactory(); ... darkBtn.draw();
darkTxt.display();
```

The dark run is the same code with a different factory.

The Abstract Factory Pattern

Clients only know about creator and product interfaces! This allows us to vary behavior without changing client code!

Products need to obey the Product interface!

Factories need to obey the Factory interface!

Creator
Classes

Product
Classes

Important: Notice the Pattern!
Adding other products for existing families requires adding another AbstractProduct interface and concrete product classes!

Important: Notice the Pattern!
Adding a new family of products requires adding another Factory, AbstractProduct interface and concrete product classes!

16

NOTES

- 1 The general structure. Clients only know the **creator and product interfaces**, so behaviour can vary without changing client code.
- 2 Every factory must obey the **Factory interface** (AbstractFactory with `createProductA()`, `createProductB()`), and every product its **Product interface**.
- 3 **Creator classes** on the left (ConcreteFactory1, ConcreteFactory2); **product classes** on the right (ConcreteProductA1, B1, A2, B2). Each concrete factory creates exactly one family: factory 1 → A1 and B1; factory 2 → A2 and B2.

EXAM ANGLE

Prediction

Q1

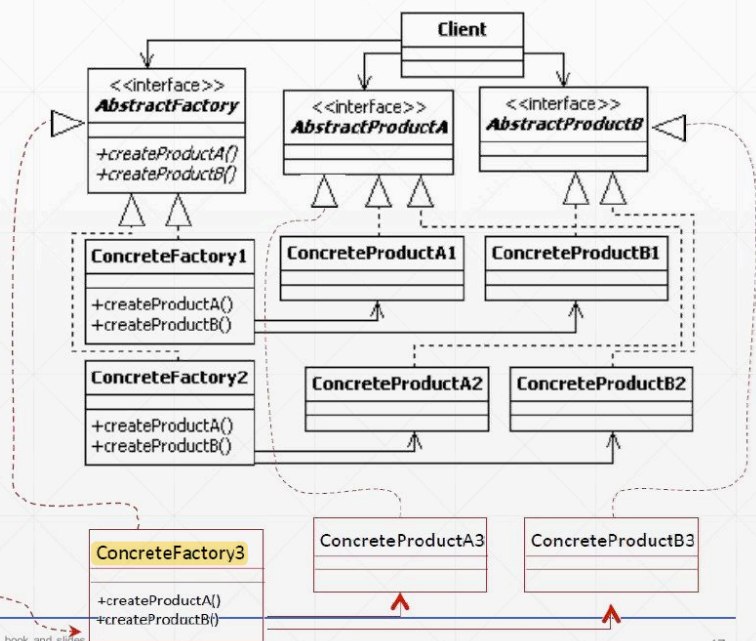
Name the participants of the Abstract Factory pattern.

Answer: AbstractFactory (interface with one create method per product), ConcreteFactory (one per family), AbstractProduct (one interface per product type), ConcreteProduct (one class per family per product), and Client (uses only AbstractFactory and AbstractProduct).

The Abstract Factory Pattern

Important:
Adding a new family of products!

A New Family of Products has been added!



Prince Sultan University | Slides prepared from "Software Engineering Design: Theory and Practice" book and slides

17

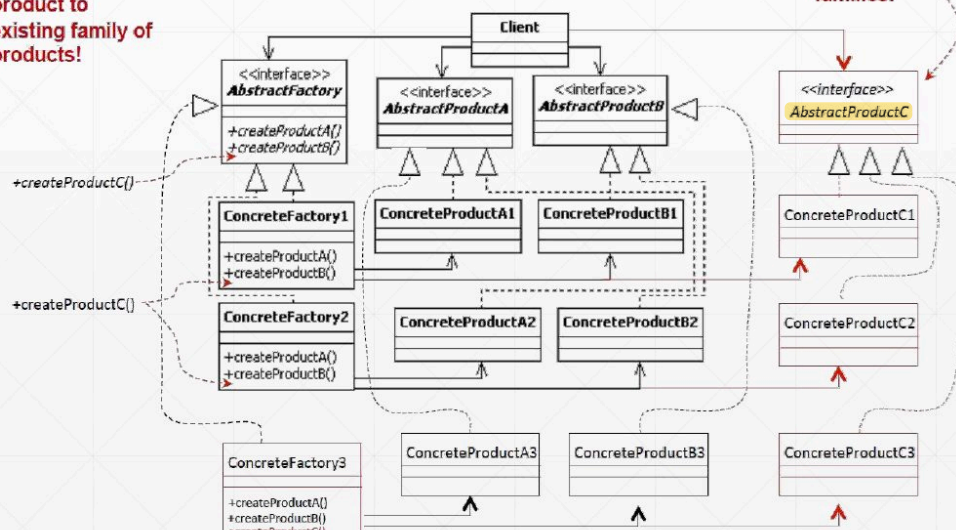
NOTES

- 1 **Adding a new family** (the red parts): add **ConcreteFactory3**, which implements the existing **AbstractFactory**, plus **ConcreteProductA3** and **B3**, which implement the existing product interfaces.
- 2 No existing class changes: the system is extended, not modified. This is the Open-Closed Principle in action, and the case Abstract Factory handles best.

The Abstract Factory Pattern

Important:
Adding a new product to existing family of products!

A New Product has been added to existing families!



Prince Sultan University | Slides prepared from "Software Engineering Design: Theory and Practice" book and slides

18

NOTES

- 1 **Adding a new product to the existing families** is harder: add «interface» **AbstractProductC** and a concrete C product for every family (C1, C2, C3)...
- 2 ...and add `createProductC()` to the **AbstractFactory interface** and therefore to **every concrete factory**. Existing classes must change, which is the pattern's main weakness.

EXAM ANGLE

Prediction

Q1

Why is adding a new product type harder than adding a new family in Abstract Factory?

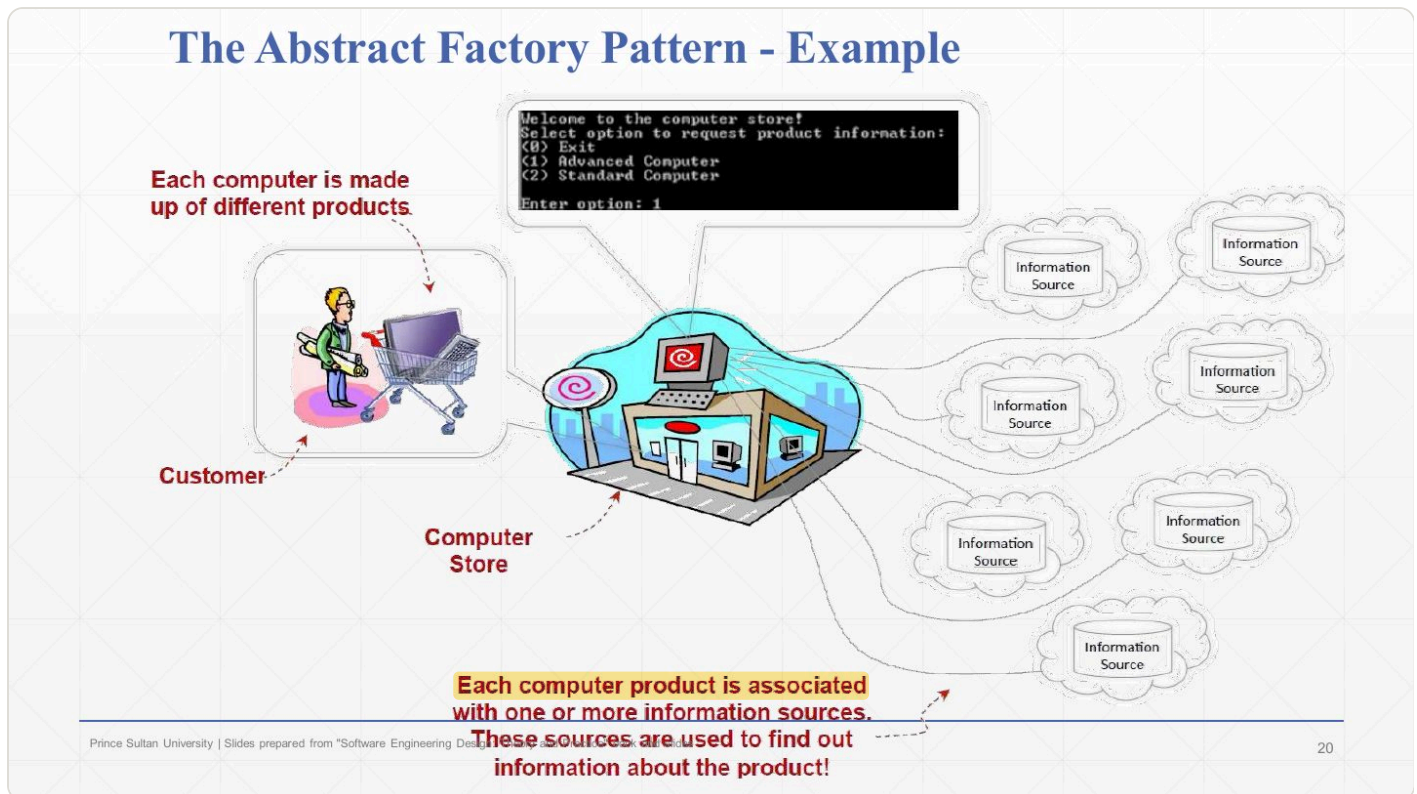
Answer: A new family only needs a new concrete factory and new concrete products, with no change to existing code. A new product type needs a new create method in the AbstractFactory interface, so every existing concrete factory must be modified (plus a new abstract product and one concrete product per family).

The Abstract Factory Pattern - Example

- Consider a software system for a computer store, where the store carries only two types of computers for sale:
 - ✓ Top of the line computer, we'll call these **advanced computers**
 - ✓ Inexpensive computers, we'll call these **standard computers**
 - ✓ Obviously, a computer store will need to carry more computers in the future!
- Advanced computers are made up of "advanced computer products," e.g. the latest multi-core CPU, wireless keyboard, advanced monitor (e.g., widescreen large 3D), advanced graphics & sound card, etc. For simplicity, we'll only use **CPU, keyboard, and Monitor** for our example.
- Standard computers are made up of "standard computer products," e.g., single core CPU, wired keyboard, small screen monitor, low-grade graphics and sound, etc. ü For simplicity, we'll only use CPU, keyboard, and Monitor for our example.
- The system is designed so that it searches **remote information sources**, e.g. online websites, remote databases, etc. for product information, such as:
 - ✓ Product reviews
 - ✓ Customer's comments from specific websites, e.g., Amazon.com
 - ✓ Manufacturers' comments
 - ✓ ...

NOTES

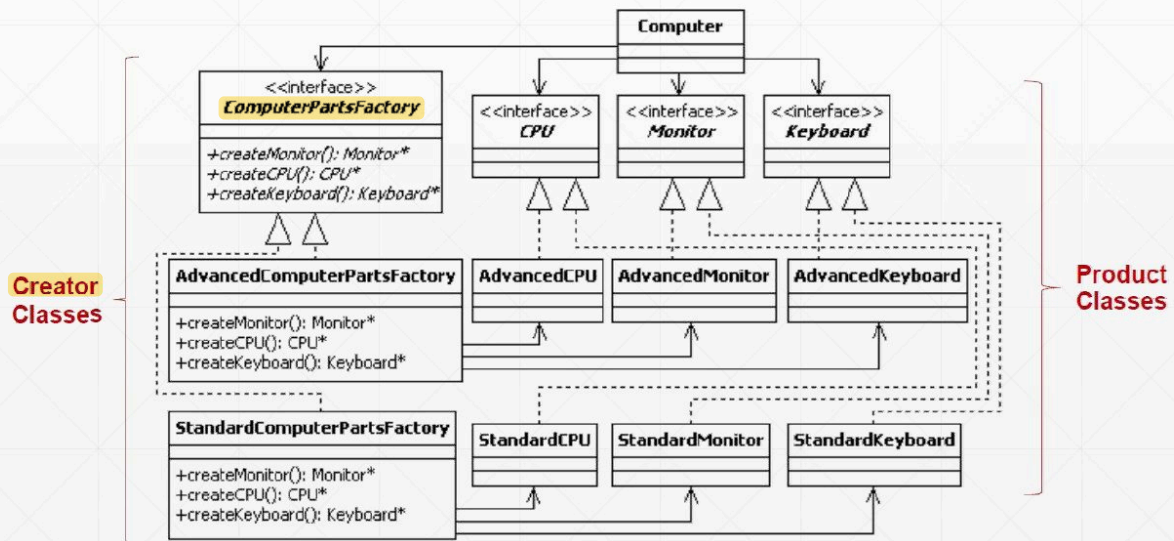
- 1 The textbook example: a computer store that sells two grades of computer, **advanced** (top of the line) and **standard** (inexpensive). More grades will come.
- 2 Each computer is made of products from its own grade: an advanced CPU, keyboard and monitor, or standard ones. The example uses only **CPU, keyboard and monitor**.
- 3 For each product the system looks up information (reviews, customer comments, manufacturer notes) from **remote sources**, and each grade's products come from different sources. So "advanced" and "standard" are product families.



NOTES

- 1 The store as a picture: a **customer** chooses an advanced or standard computer from the console menu (1 Advanced, 2 Standard).
- 2 The **computer store** system builds that computer from its products.
- 3 Each product is associated with one or more **information sources**, which are used to find out information about the product.

The Abstract Factory Pattern - Example



Prince Sultan University | Slides prepared from "Software Engineering Design: Theory and Practice" book and slides

21

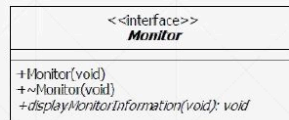
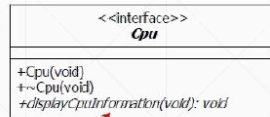
Let's break it down in the next slides...

NOTES

- 1 The example's class diagram. «interface» **ComputerPartsFactory** declares `createMonitor()`, `createCPU()` and `createKeyboard()`.
- 2 **AdvancedComputerPartsFactory** creates **AdvancedCPU**, **AdvancedMonitor** and **AdvancedKeyboard**; **StandardComputerPartsFactory** creates the standard ones.
- 3 «interface» **CPU**, **Monitor** and **Keyboard** are the abstract products; **Computer** is the client and uses only the interfaces. Creator classes on the left, product classes on the right, exactly the general structure of slide 15.

The Abstract Factory Pattern - Example

Notice the italics to denote the abstract method



// Interface for CPU

```
public interface Cpu {
    // Method for displaying CPU information.
    void displayCpuInformation();
}
```

// Interface for Monitor

```
public interface Monitor {
    // Method for displaying Monitor information.
    void displayMonitorInformation();
}
```

// Interface for Keyboard

```
public interface Keyboard {
    // Method for displaying Keyboard information.
    void displayKeyboardInformation();
}
```

NOTES

1 Step 1 of the example: the product interfaces.

CODE

```
public interface Cpu { void
displayCpuInformation(); }
```

```
public interface Monitor { void
displayMonitorInformation(); }
```

```
public interface Keyboard { void
displayKeyboardInformation(); }
```

WHAT IT DOES

Every CPU, of any grade, can display its information.

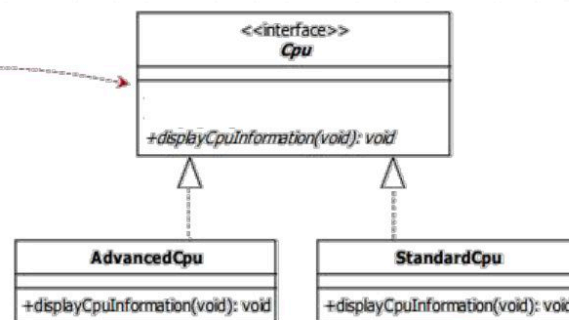
The monitor contract.

The keyboard contract.

2 In the UML on the left the method names are in *italics*: in UML, italics mark abstract operations (no body).

The CPU Product Design

```
// Cpu.java
public interface Cpu {
    void displayCpuInformation();
    // ... other methods for the Cpu class
}
```



```
// AdvancedCpu.java
public class AdvancedCpu implements Cpu {
    public AdvancedCpu() {
        // Constructor
    }

    @Override
    public void displayCpuInformation() {
        // Implementation code
    }
    // ... other methods for the AdvancedCpu class
}
```

```
// StandardCpu.java
public class StandardCpu implements Cpu {
    public StandardCpu() {
        // Constructor
    }

    @Override
    public void displayCpuInformation() {
        // Implementation code
    }
    // ... other methods for the StandardCpu class
}
```

Prince Sultan University | Sli

NOTES

- 1 The CPU product design: «interface» **Cpu** with the abstract `displayCpuInformation()`, realised by **AdvancedCpu** and **StandardCpu**.
- 2 The code:

CODE

WHAT IT DOES

```
public class AdvancedCpu implements
Cpu {
```

The advanced family's CPU.

```
    public AdvancedCpu() { /*
Constructor */ }
```

```
    @Override public void
displayCpuInformation() { /*
Implementation code */ }
```

Its own way of displaying information.

```
public class StandardCpu implements
Cpu { ... @Override public void
displayCpuInformation() { ... } }
```

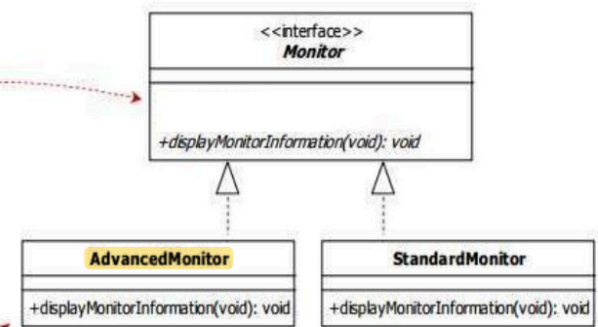
The standard family's CPU, same contract.

The monitor Product Design

```
// Monitor.java
public interface Monitor {
    void displayMonitorInformation();
    // ... other methods for the Monitor interface
}
```

```
// AdvancedMonitor.java
public class AdvancedMonitor implements Monitor {
    public AdvancedMonitor() {
        // Constructor
    }

    @Override
    public void displayMonitorInformation() {
        // Implementation code
    }
    // ... other advanced monitor methods
}
```



```
// StandardMonitor.java
public class StandardMonitor implements Monitor {
    public StandardMonitor() {
        // Constructor
    }

    @Override
    public void displayMonitorInformation() {
        // Implementation code
    }
    // ... other methods for the StandardMonitor class
}
```

Prince Sultan Univ

NOTES

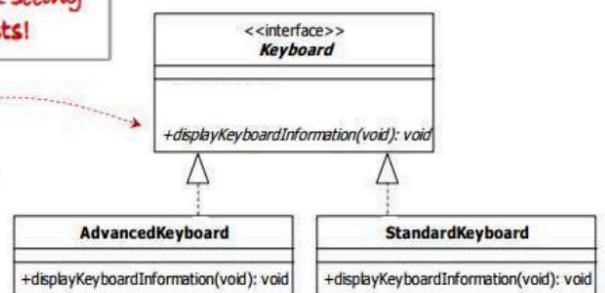
- 1 The monitor product design follows the same shape: «interface» **Monitor** with `displayMonitorInformation()`, realised by **AdvancedMonitor** and **StandardMonitor**, each with a constructor and an `@Override` of the display method.

The keyboard Product Design

```
// Keyboard.java
public interface Keyboard {
    void displayKeyboardInformation();
    // ... other methods for the Keyboard interface
}
```

```
// AdvancedKeyboard.java
public class AdvancedKeyboard implements Keyboard {
    public AdvancedKeyboard() {
        // Constructor
    }

    @Override
    public void displayKeyboardInformation() {
        // Implementation code for advanced keyboard
    }
    // ... other advanced keyboard methods
}
```



```
// StandardKeyboard.java
public class StandardKeyboard implements Keyboard {
    public StandardKeyboard() {
        // Constructor
    }

    @Override
    public void displayKeyboardInformation() {
        // Implementation code for standard keyboard
    }
    // ... other methods for the StandardKeyboard class
}
```

Prince Sultan University

Hopefully by this point you can start seeing the pattern for designing products!

NOTES

- 1 The keyboard product design: «interface» **Keyboard**, realised by **AdvancedKeyboard** and **StandardKeyboard**.
- 2 As the callout says, the pattern for designing products should now be visible: one interface per product type, one concrete class per family.

SLIDE 25 / 39 The CPU Product Implementation

The CPU Product Implementation

```

// AdvancedCpu.java
public class AdvancedCpu {

    // Constructor
    public AdvancedCpu() {
        // Intentionally left blank.
    }

    // Method for retrieving the CPU's information
    public void displayCpuInformation() {
        // Since this is an example, we will assume that the advanced CPU's information
        // will be retrieved from information source A, e.g., database A, file A, etc.
        // This may require a particular database connection, file access, etc.
        System.out.println("\nInformation retrieved from source A.
                            \nDisplaying the advanced CPU's information.\n");
    }
}

```

Information Source A

Database for Advanced Products

File with Information of Advanced Products

The code in this function knows how to retrieve information from data source **A**, which can use specific format, location, etc.

```

// StandardCpu.java
public class StandardCpu {

    // Constructor
    public StandardCpu() {
        // Intentionally left blank.
    }

    // Method for retrieving the CPU's information
    public void displayCpuInformation() {
        // Since this is an example, we will assume that the standard CPU's information
        // will be retrieved from information source B, e.g., database B, file B, etc.
        // This may require a particular database connection, file access, etc.
        System.out.println("\nInformation retrieved from source B.
                            \nDisplaying the standard CPU's information.\n");
    }
}

```

Information Source B

Database for Standard Products

File with Information of Standard Products

The code in this function knows how to retrieve information from data source **B**, which can use specific format, location, etc.

Prince Sultan University | Slides prepared from "Software Engineering Design: Theory and Practice" book and slides

26

NOTES

- 1 The CPU implementations:

CODE	WHAT IT DOES
<pre>public class AdvancedCpu { public AdvancedCpu() { } ...</pre>	Constructor intentionally left blank.
<pre> public void displayCpuInformation() { ... System.out.println("\nInformation retrieved from source A.\nDisplaying the advanced CPU's information.\n"); }</pre>	The advanced CPU gets its data from information source A (database A, file A...).
<pre>public class StandardCpu { ... displayCpuInformation() { ... "retrieved from source B" ... } }</pre>	The standard CPU gets its data from source B .

- 2 Each concrete product knows how to reach its own source (connection, format, location). The client never sees this: it just calls `displayCpuInformation()`.

Other Product Implementation

```
// Method for retrieving the keyboard's information
public void displayKeyboardInformation() {
    // Since this is an example, we will assume that the advanced keyboard's information
    // will be retrieved from information source A, e.g., database A, file A, etc.
    System.out.println("\nInformation retrieved from source A.\nDisplaying the advanced keyboard's information.\n");
}

// Method for retrieving the keyboard's information
public void displayKeyboardInformation() {
    // Since this is an example, we will assume that the standard keyboard's information
    // will be retrieved from information source B, e.g., database B, file B, etc.
    System.out.println("\nInformation retrieved from source B.\nDisplaying the standard keyboard's information.\n");
}

// Method for retrieving the monitor's information
public void displayMonitorInformation() {
    // Since this is an example, we will assume that the advanced monitor's information
    // will be retrieved from information source A, e.g., database A, file A, etc.
    System.out.println("\nInformation retrieved from source A.\nDisplaying the advanced monitor's information.\n");
}

// Method for retrieving the monitor's information
public void displayMonitorInformation() {
    // Since this is an example, we will assume that the standard monitor's information
    // will be retrieved from information source B, e.g., database B, file B, etc.
    System.out.println("\nInformation retrieved from source B.\nDisplaying the standard monitor's information.\n");
}
```

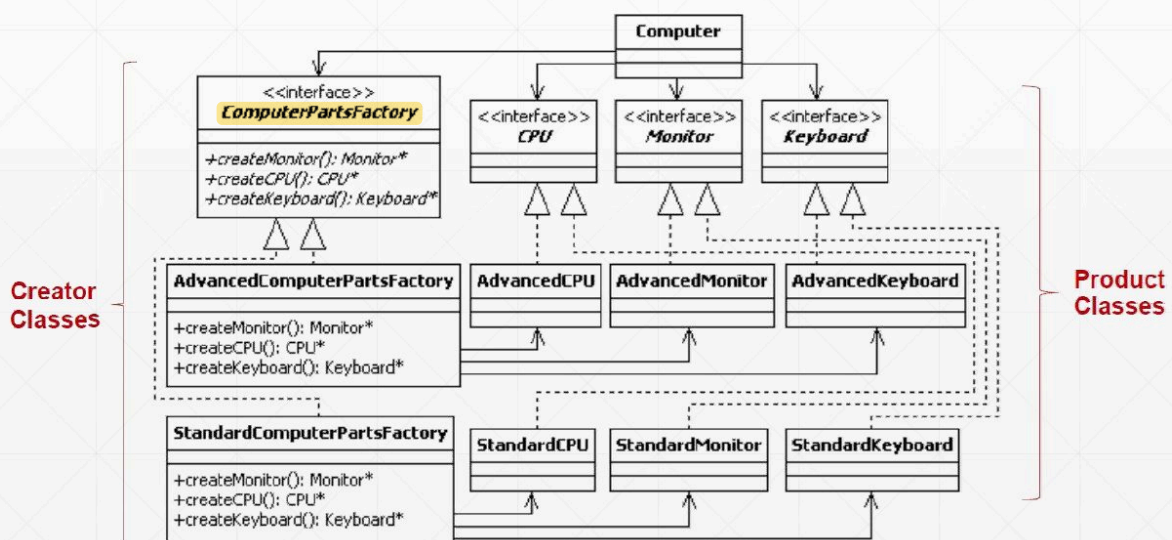
All other products
are implemented
using the same
pattern!

NOTES

- 1 The keyboards and monitors are implemented the same way: advanced products print "Information retrieved from source A", standard products "source B", each followed by its own display text.

SLIDE 27 / 39 The Abstract Factory Pattern - Example

The Abstract Factory Pattern - Example



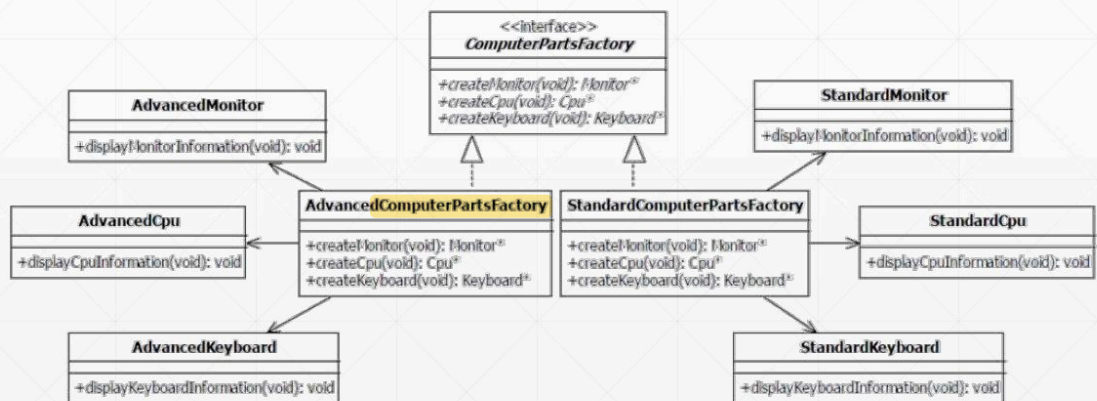
Let's break it down in the next slides...

NOTES

- 1 The example class diagram again (same as slide 20) before the factories are built: the creator classes are next.

SLIDE 28 / 39 Factory Interfaces and

Factory Interfaces and Concrete Factories



Important:
This design connects the products designed in the previous slides with the factories used to abstract their creation!

NOTES

- 1 The factories in detail: «interface» **ComputerPartsFactory** with `createMonitor()`, `createCpu()`, `createKeyboard()`.
- 2 **AdvancedComputerPartsFactory** is connected to **AdvancedMonitor**, **AdvancedCpu** and **AdvancedKeyboard**; **StandardComputerPartsFactory** to the standard three. This design connects the products from the previous slides to the factories that abstract their creation.

The Client Computer Design

```
class ComputerStore {
    private Monitor monitor;
    private Cpu cpu;
    private Keyboard keyboard;

    public ComputerStore(ComputerPartsFactory computerPartsFactory) {
        this.monitor = computerPartsFactory.createMonitor();
        this.cpu = computerPartsFactory.createCpu();
        this.keyboard = computerPartsFactory.createKeyboard();
    }

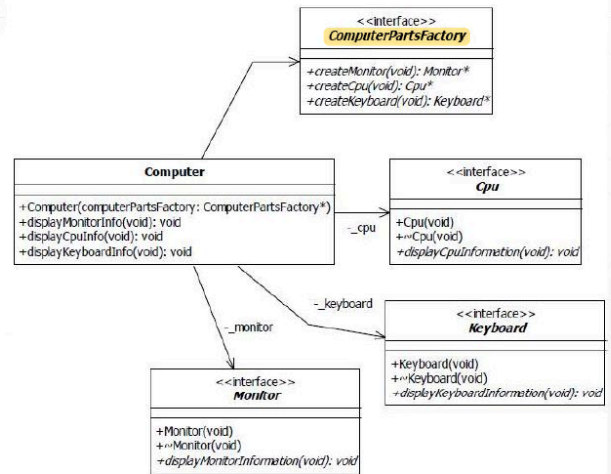
    public void displayMonitorInfo() {
        monitor.displayMonitorInformation();
    }

    public void displayCpuInfo() {
        cpu.displayCpuInformation();
    }

    public void displayKeyboardInfo() {
        keyboard.displayKeyboardInformation();
    }

    // Other ComputerStore methods...

    // Make sure to properly clean up resources if needed,
    // though Java has garbage collection, you might need to clean up
    // resources manually if you are using resources like sockets, files etc.
}
```



NOTES

1 The client:

CODE

```
class ComputerStore { private Monitor
monitor; private Cpu cpu; private
Keyboard keyboard;
```

WHAT IT DOES

Its fields are abstract product types only.

```
    public
    ComputerStore(ComputerPartsFactory
    computerPartsFactory) {
```

The factory is passed in: the store never chooses the family itself.

```
        this.monitor =
        computerPartsFactory.createMonitor();
```

Ask the factory for each part...

```
        this.cpu =
        computerPartsFactory.createCpu();
        this.keyboard =
        computerPartsFactory.createKeyboard();
    }
```

...all from the same family, so they always match.

```
    public void displayMonitorInfo() {
        monitor.displayMonitorInformation(); }
```

Using a part: a call on the interface.

```
        ... displayCpuInfo(),
        displayKeyboardInfo() ...
```

2 In the UML the client (Computer) depends on ComputerPartsFactory and on the Cpu, Monitor and Keyboard interfaces, and on nothing concrete.

The Abstract Factory Example – Putting it all together

```
import java.util.Scanner;

public class Main {

    public static void main(String[] args) {
        AdvancedComputerPartsFactory advancedFactory = new AdvancedComputerPartsFactory();
        StandardComputerPartsFactory standardFactory = new StandardComputerPartsFactory();

        ComputerStore cs = null;
        Scanner scanner = new Scanner(System.in);
        int option = 1;

        System.out.println("Welcome to the computer store!");

        while (option != 0) {
            System.out.println("Select option to request product information:");
            System.out.println("(0) Exit");
            System.out.println("(1) Advanced Computer");
            System.out.println("(2) Standard Computer");
            System.out.print("Enter option: ");

            option = scanner.nextInt();

            if (option == 1) {
                cs = new ComputerStore(advancedFactory);
            } else if (option == 2) {
                cs = new ComputerStore(standardFactory);
            }

            if (cs != null) {
                cs.displayMonitorInfo();
                cs.displayKeyboardInfo();
                cs.displayCpuInfo();
            }

            scanner.close();
        }
    }
}
```

```
Welcome to the computer store!
Select option to request product information:
<0> Exit
<1> Advanced Computer
<2> Standard Computer
Enter option: 1
Information retrieved from source A.
Displaying the advanced monitor's information.

Information retrieved from source A.
Displaying the advanced keyboard's information.

Information retrieved from source A.
Displaying the advanced cpu's information.
Select option to request product information:
<0> Exit
<1> Advanced Computer
<2> Standard Computer
Enter option:
```

31

NOTES

1 The main program:

CODE

```
AdvancedComputerPartsFactory
advancedFactory = new
AdvancedComputerPartsFactory();
```

WHAT IT DOES

Create one factory per family (the only place concrete factories are named).

```
StandardComputerPartsFactory
standardFactory = new
StandardComputerPartsFactory();
```

```
while (option != 0) { ... option =
scanner.nextInt();
```

Menu: 0 Exit, 1 Advanced, 2 Standard.

```
if (option == 1) cs = new
ComputerStore(advancedFactory);
```

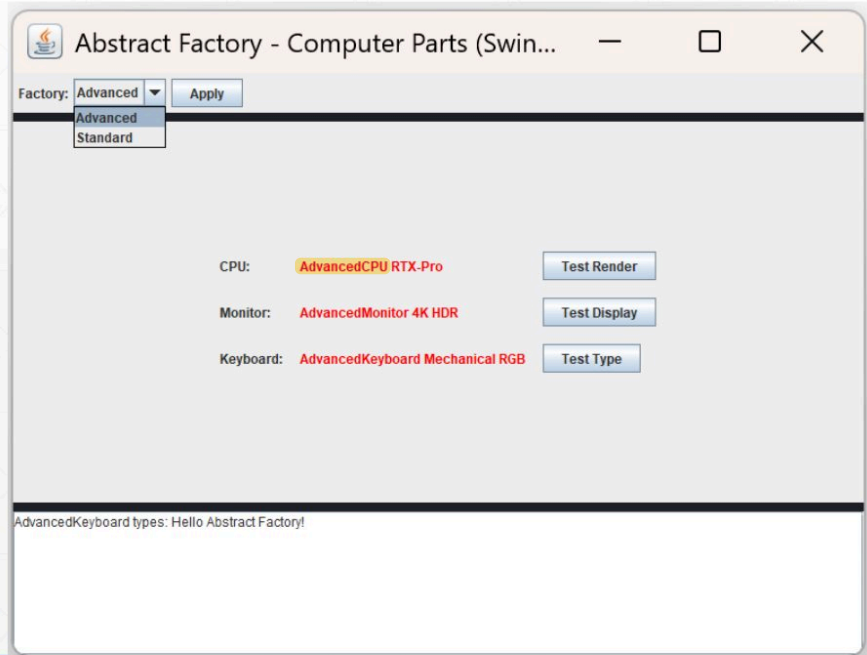
Choosing the family = choosing the factory.

```
else if (option == 2) cs = new
ComputerStore(standardFactory);
```

```
if (cs != null) {
cs.displayMonitorInfo();
cs.displayKeyboardInfo();
cs.displayCpuInfo(); }
```

The same calls for either family.

2 The console on the right shows option 1: every line says "retrieved from source A", so all parts came from the advanced family.

Extra:**NOTES**

- 1 A Swing version of the example: picking **Advanced** or **Standard** in the Factory menu and pressing Apply swaps the factory, and the CPU, monitor and keyboard shown (AdvancedCPU RTX-Pro, AdvancedMonitor 4K HDR, AdvancedKeyboard Mechanical RGB) all switch family together.

Abstract Factory – Step by Step Summary

- As seen, the Abstract Factory pattern can be used over and over to support new family of products or to add new products to existing ones. When designing with the Abstract Factory, execute the following steps:
 1. Design the **product interfaces** (e.g., Cpu, Monitor, and Keyboard)
 2. Identify the different **families or groups** required for the problem (e.g., standard vs. advanced computers)
 3. For each **product group** identified in step 2, design **concrete products** that realize the respective product interfaces identified in step 1.
 4. Create the **factory interface** (e.g., ComputerPartsFactory). The factory interface contains *n* interface methods, one for each product interface identified in step 1.
 5. For each **family or group** identified in step 2, create **concrete factories** that realize the factory interface created in step 4.
 6. Associate each **concrete factory** from step 5 with their **respective products** from step 3.
 7. **Create the Client** (e.g., Computer) which is associated with both **product** and **factory** interfaces created in steps 1 and 4, respectively.

NOTES

- Steps 1–3:** design the product interfaces (Cpu, Monitor, Keyboard); identify the families (standard, advanced); for each family, design concrete products that realise the interfaces.
- Steps 4–6:** create the factory interface with one method per product interface (ComputerPartsFactory); create one concrete factory per family; associate each concrete factory with its family's products.
- Step 7:** create the client (Computer), associated with both the product interfaces and the factory interface.
- This recipe is the model answer to “apply Abstract Factory to ...” questions: name the product interfaces, the families, the factory interface and the concrete factories, then draw them.

EXAM ANGLE

Prediction

Q1

Apply the Abstract Factory pattern to a furniture shop that sells Modern and Victorian sets, each with a Chair and a Sofa. Name all classes and interfaces.

Answer: Product interfaces: Chair, Sofa. Families: Modern, Victorian. Concrete products: ModernChair, ModernSofa, VictorianChair, VictorianSofa. Factory interface: FurnitureFactory with createChair(), createSofa(). Concrete factories: ModernFurnitureFactory, VictorianFurnitureFactory, each creating its own family. Client: Shop, which holds a FurnitureFactory and uses only Chair and Sofa.

SLIDE 33 / 39 Cons & Pros of Abstract Factory

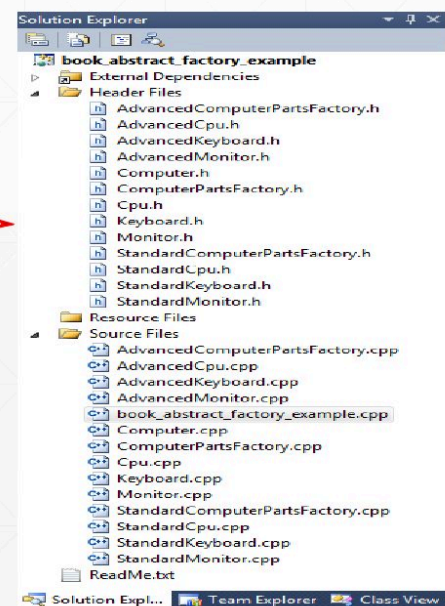
Cons & Pros of Abstract Factory

➤ Cons

- ✓ Large number of classes are required

➤ Pros

- ✓ Isolates concrete product classes so that reusing them becomes easier
- ✓ Promotes consistency within specific product families.
- ✓ Adding new families of products require no modification to existing code.
 - Additions are made through extension, therefore, obeying the OCP.
- ✓ Increases modifiability



NOTES

- 1 **Con:** a large number of classes (an interface per product, a class per product per family, a factory per family).
- 2 **Pros:** concrete product classes are isolated, so they are easier to reuse; products within a family are guaranteed **consistent**; adding a new family needs no change to existing code, because additions are made by extension (**OCP**); modifiability increases. (The hidden con from slide 17: adding a new product type changes every factory.)

EXAM ANGLE

Prediction

Q1

Give one disadvantage and two advantages of the Abstract Factory pattern.

Answer: Disadvantage: many classes (and adding a new product type requires changing every factory). Advantages: concrete classes are isolated from the client; products of a family are always consistent; new families are added by extension without modifying existing code (OCP), increasing modifiability.

SLIDE 34 / 39 Abstract Factory Step by Step Summary (repeated)

Abstract Factory – Step by Step Summary

- As seen, the Abstract Factory pattern can be used over and over to support new family of products or to add new products to existing ones. When designing with the Abstract Factory, execute the following steps:
 1. Design the **product interfaces** (e.g., Cpu, Monitor, and Keyboard)
 2. Identify the different **families or groups** required for the problem (e.g., standard vs. advanced computers)
 3. For each **product group** identified in step 2, design **concrete products** that realize the respective product interfaces identified in step 1.
 4. Create the **factory interface** (e.g., ComputerPartsFactory). The factory interface contains n interface methods, one for each product interface identified in step 1.
 5. For each **family or group** identified in step 2, create **concrete factories** that realize the factory interface created in step 4.
 6. Associate each **concrete factory** from step 5 with their **respective products** from step 3.
 7. **Create the Client** (e.g., Computer) which is associated with both **product** and **factory** interfaces created in steps 1 and 4, respectively.

NOTES

- 1 The seven-step summary repeated (see slide 32).

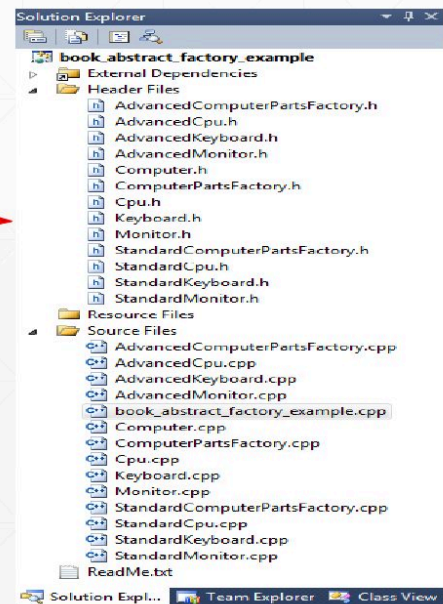
Cons & Pros of Abstract Factory

➤ Cons

- ✓ Large number of classes are required

➤ Pros

- ✓ Isolates concrete product classes so that reusing them becomes easier
- ✓ Promotes consistency within specific product families.
- ✓ Adding new families of products require no modification to existing code.
 - Additions are made through extension, therefore, obeying the OCP.
- ✓ Increases modifiability



NOTES

- 1 The pros and cons repeated (see slide 33).

SLIDE 36 / 39 Extra Example: RiyadhHQ car factories

Extra Example:

- **Question:**
Imagine you are managing **RiyadhHQ**, the central client that needs to request cars from different branches.
- Instead of directly creating specific classes like **JeddahMercedes** or **DammamBMW**, RiyadhHQ only works with the **CarFactory** interface.
- The abstract factory (**CarFactory**) defines the family of products RiyadhHQ can request — in this case, **Mercedes** and **BMW**.
- To handle these requests, we implement concrete factories: **JeddahCarFactory** and **DammamCarFactory**, each responsible for producing their own variations of the cars (e.g., JeddahCarFactory produces JeddahMercedes and JeddahBMW, while DammamCarFactory produces DammamMercedes and DammamBMW).
- Now answer the following:
 - Why does RiyadhHQ (the client) not need to know which specific factory or city produces the car?
 - How does the use of the **CarFactory** interface make it easier to switch between factories at runtime without changing RiyadhHQ's code?
 - Explain how this example demonstrates the **Abstract Factory** design pattern.

NOTES

- 1 An exercise: **RiyadhHQ** is the client that requests cars from different branches, without creating JeddahMercedes or DammamBMW itself.
- 2 **CarFactory** is the abstract factory: it defines the products RiyadhHQ can request, a Mercedes and a BMW.
- 3 **JeddahCarFactory** makes JeddahMercedes and JeddahBMW; **DammamCarFactory** makes DammamMercedes and DammamBMW. Each city is a family.
- 4 The questions: RiyadhHQ doesn't need to know the city because it only uses CarFactory, Mercedes and BMW; switching factories at run time is just passing a different CarFactory object; and the example shows Abstract Factory because each concrete factory creates a consistent family of related products behind one interface.

EXAM ANGLE

Prediction

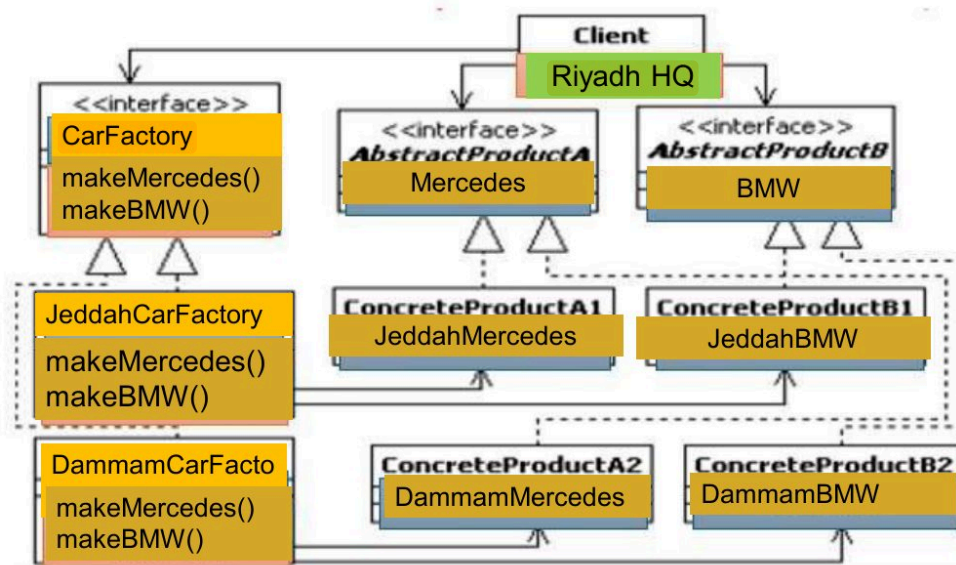
Q1

Why does RiyadhHQ not need to know which branch produced its cars, and how does it switch branches?

Answer: It depends only on the CarFactory interface and the abstract Mercedes and BMW types, so the concrete classes stay hidden in the branch factories. Switching branches means giving RiyadhHQ a different CarFactory object (JeddahCarFactory or DammamCarFactory) at run time; its own code does not change.

SLIDE 37 / 39 Extra: RiyadhHQ class diagram

Extra



NOTES

- 1 The model diagram: «interface» **CarFactory** with `makeMercedes()` and `makeBMW()`; **JeddahCarFactory** and **DammamCarFactory** implement it.
- 2 Client **Riyadh HQ** uses **CarFactory** and the abstract products **Mercedes** and **BMW**; **JeddahMercedes/DammamMercedes** and **JeddahBMW/DammamBMW** are the concrete products, each created by its own city's factory.

SLIDE 38 / 39 Extra: Do we create concrete classes?

Extra:

We do create concrete classes in the Abstract Factory?

Yes, at the end we need to write the concrete classes (e.g., `DarkButton`, `LightButton`, `DarkTextField`, `LightTextField`). Otherwise, nothing real would exist in the program.

We also create concrete factories (e.g., `DarkThemeFactory`, `LightThemeFactory`) that know how to instantiate those concrete products.

2. But the *client* doesn't care

The whole point of the **Abstract Factory** pattern is **decoupling**.

- The **client code** (the part of the system that *uses* the objects) doesn't know or care about the *concrete classes*.
- The client only works with **abstract types** (interfaces or base classes).

```
// Client only knows about these:
Button btn;
TextField tf;

// Client doesn't know if it's Dark or Light theme
GUIFactory factory = FactoryProvider.getFactory("dark");

btn = factory.createButton();
tf = factory.createTextField();
```

Notice: the client never uses `new DarkButton()` directly. It just calls the abstract method.

3. Why this matters

- If you later add a **new family** (say `NeonThemeFactory` with `NeonButton`, `NeonTextField`), you don't touch client code at all.
- Without Abstract Factory, you'd have **if-else spaghetti** inside the client, directly creating `new DarkButton()` or `new LightButton()` everywhere.

39

NOTES

- 1 Yes, concrete classes are still written (`DarkButton`, `LightTextField`...), and concrete factories that instantiate them. Otherwise nothing real would exist.
- 2 But the **client** doesn't care: the point of the pattern is **decoupling**. Client code works only with abstract types (`Button btn`; `GUIFactory factory = FactoryProvider.getFactory("dark"); btn = factory.createButton();`) and never writes `new DarkButton()`.
- 3 Why it matters: a new `NeonThemeFactory` with `NeonButton` and `NeonTextField` needs no client change. Without the pattern you get if-else "spaghetti" creating `new DarkButton()` or `new LightButton()` all over the client.

Summary

- In this session, we presented fundamentals concepts of design patterns and creational design patterns, including:
 - ✓ Abstract Factory
- In the next sessions, we will continue the presentation on creational design patterns, including:
 - ✓ Factory method
 - ✓ Builder
 - ✓ Singleton

NOTES

- 1 This lecture: design-pattern basics and the **Abstract Factory**.
- 2 Next: the other creational patterns, **Factory Method**, **Builder** and **Singleton**.

Original slides from the SE322 lecture deck, with study notes. Check exact wording against the [original deck](#).
Notes and questions are study aids. Made by Shoug Alomran.