Software Design and Architecture

[Creational Design Patterns] – Chapter 06, L01

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…**

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?!!

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.

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

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

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.

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.

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

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.

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.

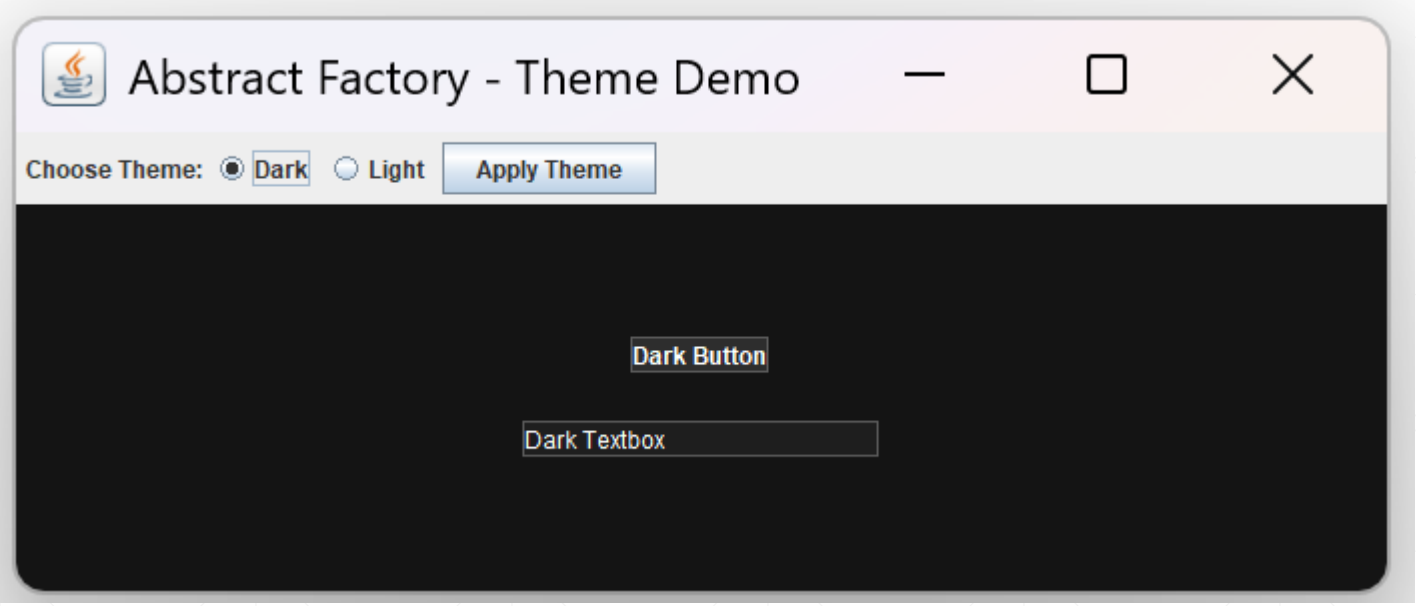
Extra

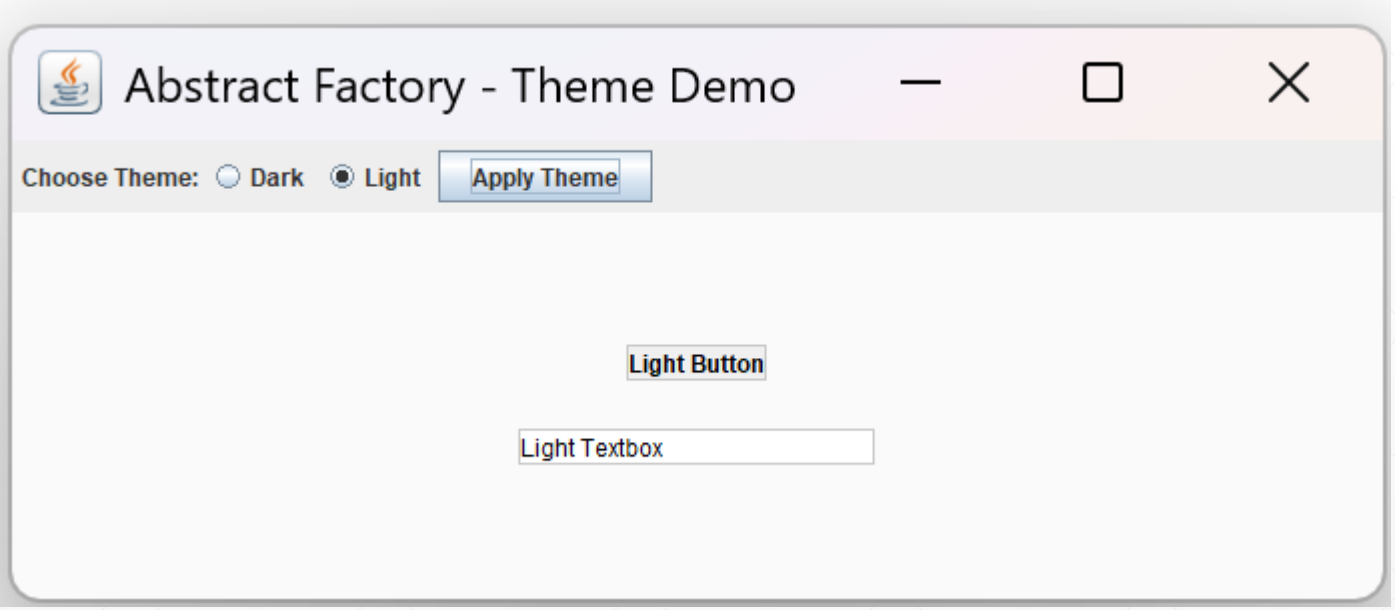
- **Question:**

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.

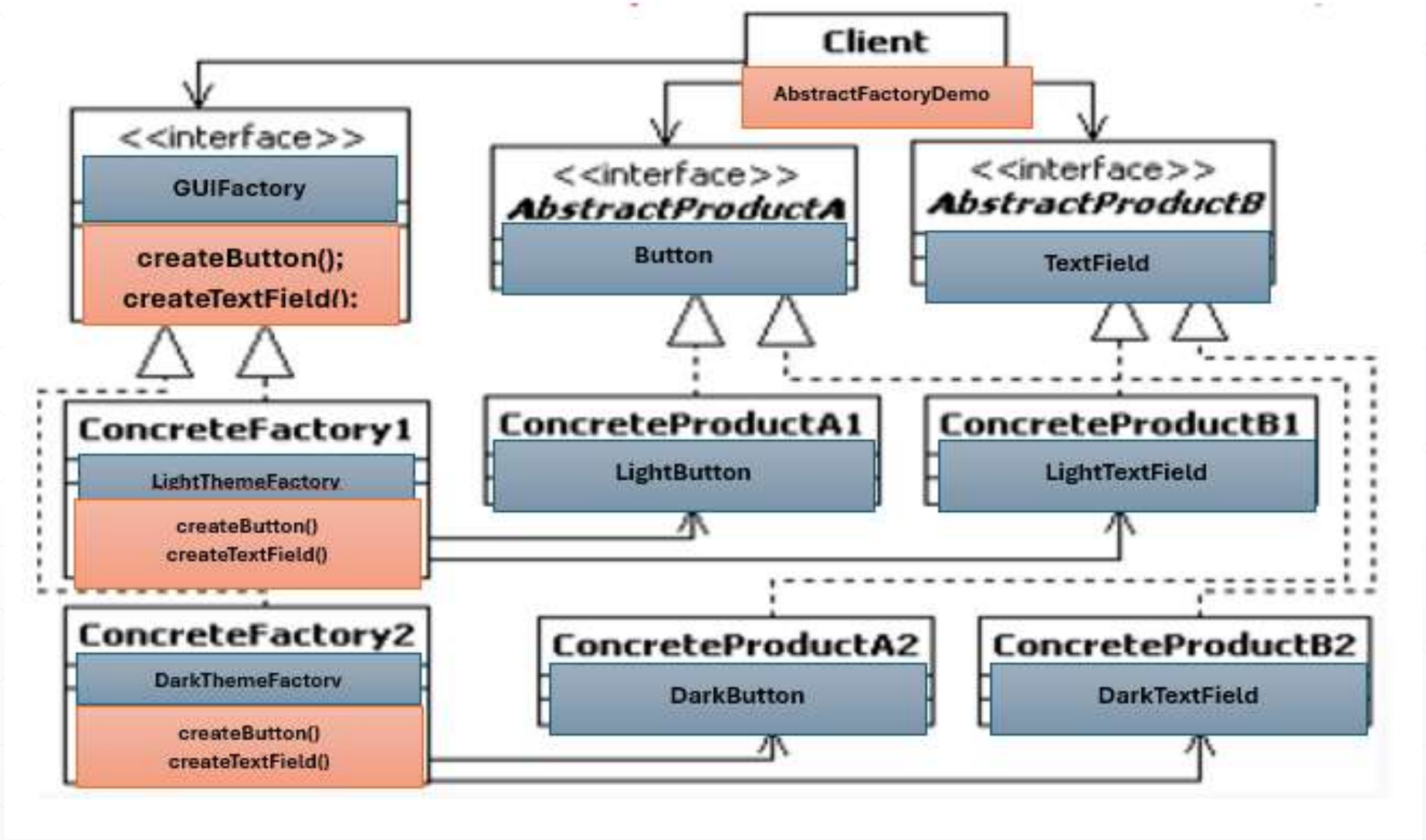
- **So: Abstract Factory =** ***"One stop shop for creating related objects (family), while hiding the details of their concrete classes."***


Abstract Factory - Theme Demo
Choose Theme:
Dark
Light
Apply Theme
Dark Button
Dark Textbox


Abstract Factory - Theme Demo
Choose Theme:
Dark
Light
Apply Theme
Light Button
Light Textbox

Extra:

Extra


Client
AbstractFactoryDemo
<<interface>>
GUIFactory
createButton();
createTextField():
<<interface>>
AbstractProductA
Button
<<interface>>
AbstractProductB
TextField
ConcreteFactory1
LightThemeFactory
createButton()
createTextField()
ConcreteProductA1
LightButton
ConcreteProductB1
LightTextField
ConcreteFactory2
DarkThemeFactory
createButton()
createTextField()
ConcreteProductA2
DarkButton
ConcreteProductB2
DarkTextField

Step 1: Define Abstract Factory

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

Step 2: Implement Concrete Factories

```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();
    }
}
```

Step 3: Define Abstract Products

```java
interface Button {
    void draw();
}

interface TextField {
    void display();
}
```

Extra:

Step 4: Create Concrete Products

java

```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

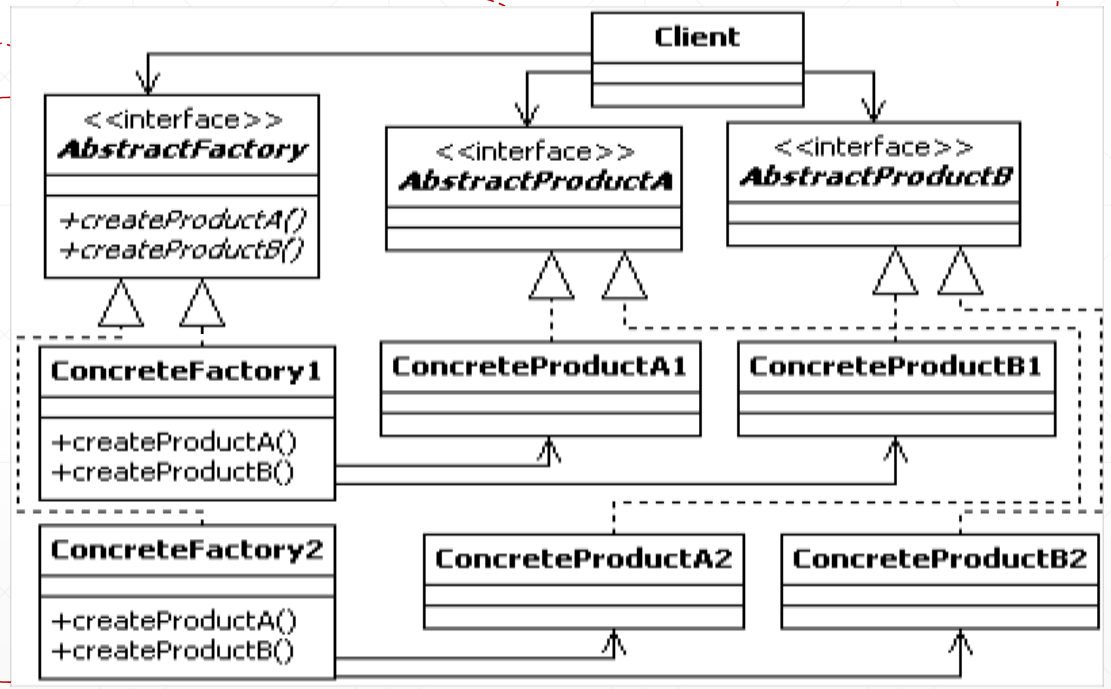
```java
public class AbstractFactoryDemo {
    public static void main(String[] args) {

        // ====== 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();
    }
}
```

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

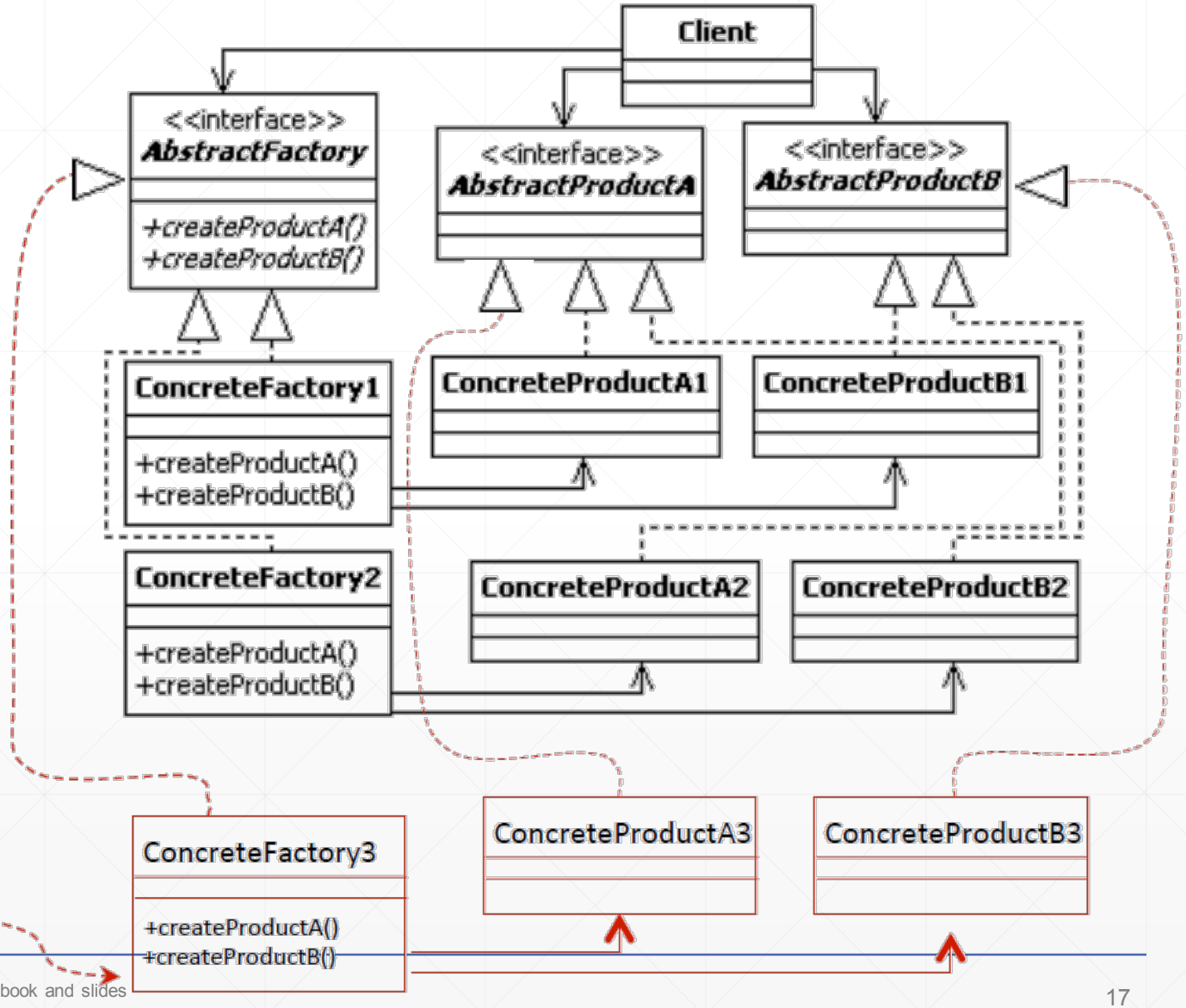
Product Classes

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!

The Abstract Factory Pattern

Important: Adding a new family of products!

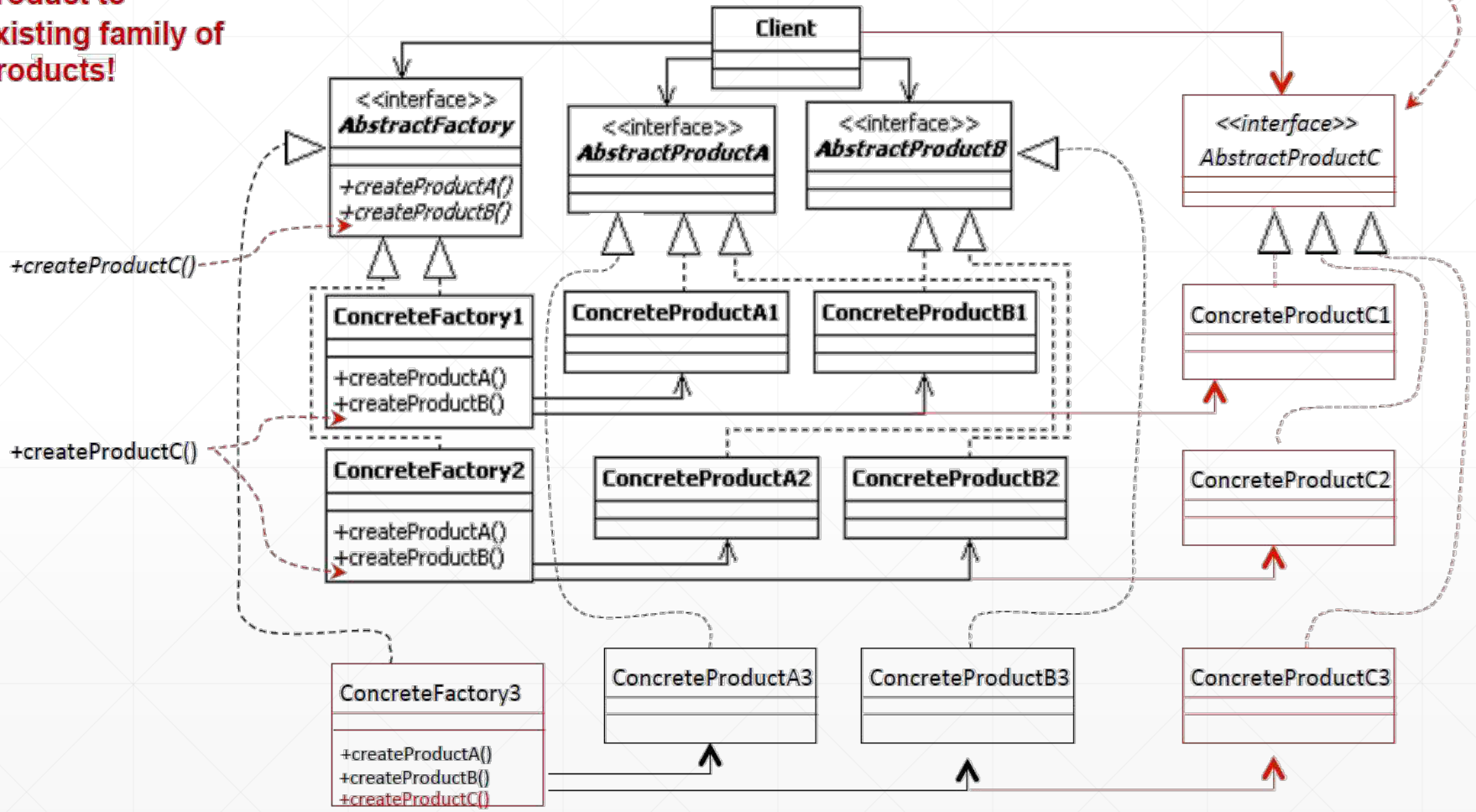


A New Family of Products has been added!

The Abstract Factory Pattern

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

A New Product has been added to existing families!



The Abstract Factory Pattern - Example

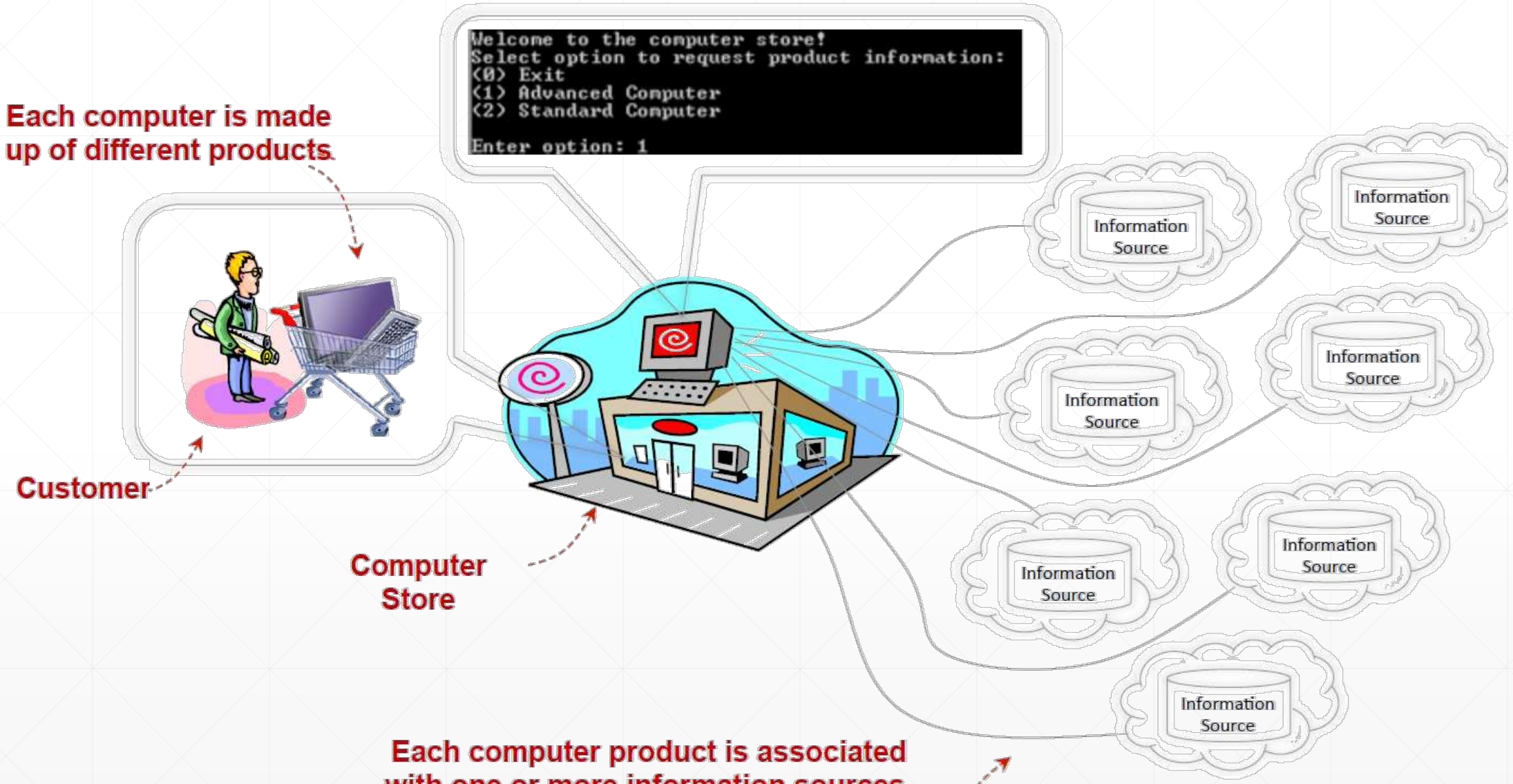
- Consider a software system for a computer store, where the store carries only two types of computers for sale:
  - ✓ Top of the line computer, we'll call these advanced computers
  - ✓ Inexpensive computers, we'll call these standard computers
  - ✓ Obviously, a computer store will need to carry more 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.

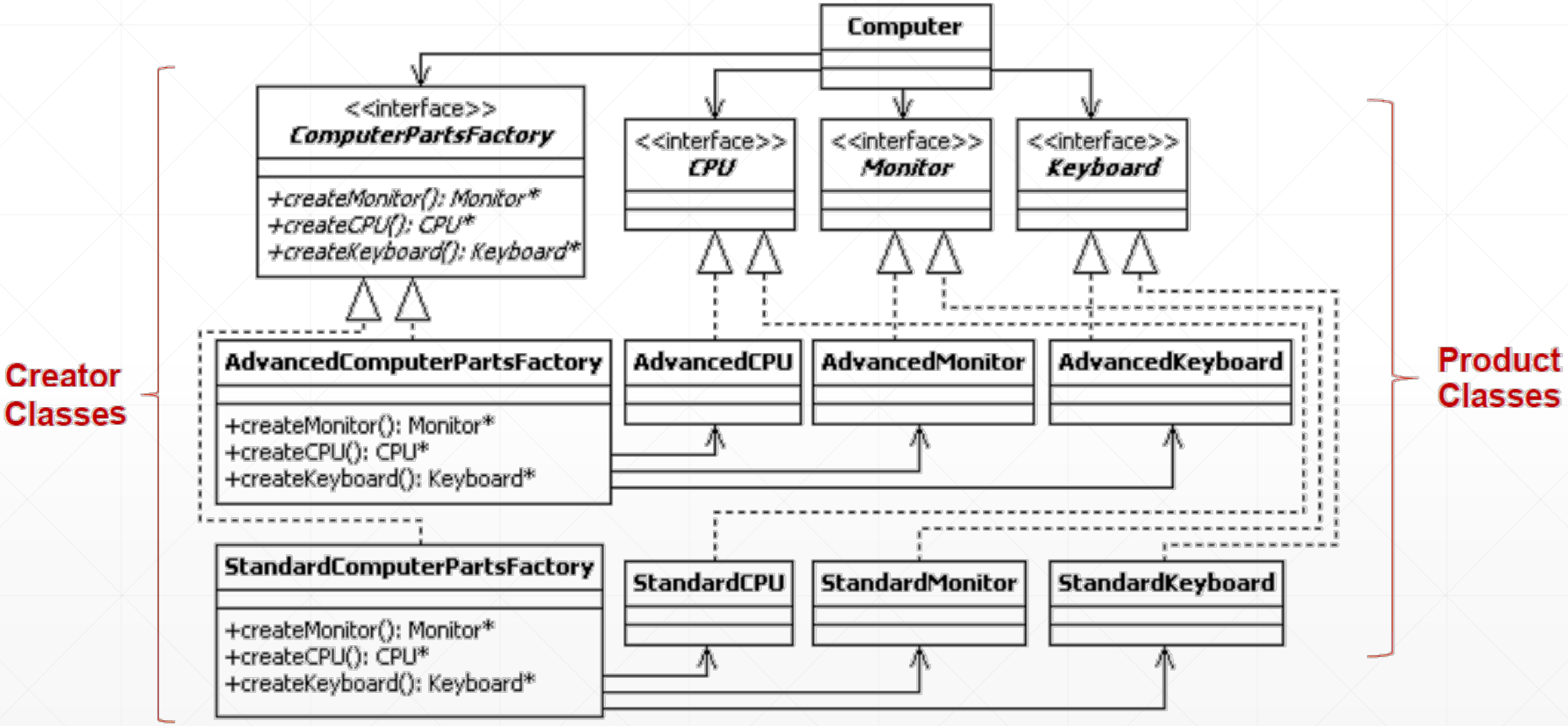
- Standard computers are made up of "standard computer products," e.g., single core CPU, wired keyboard, small screen monitor, low-grade graphics and sound, etc. ü For simplicity, we'll only use CPU, keyboard, and Monitor for our example.

- 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
  - ✓ …

The Abstract Factory Pattern - Example

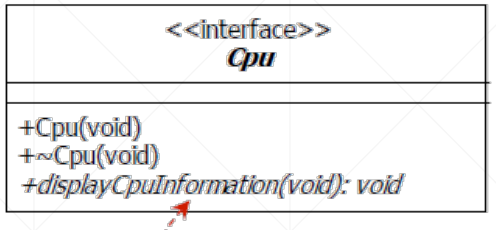


The Abstract Factory Pattern - Example



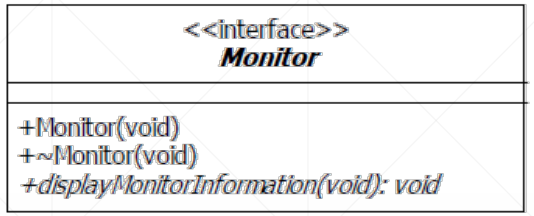
Let's break it down in the next slides...

The Abstract Factory Pattern - Example

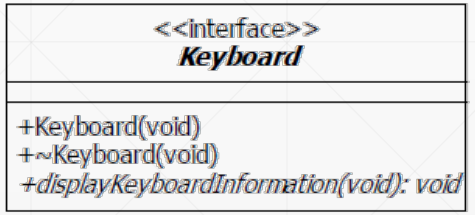


Notice the italics to denote the abstract method

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



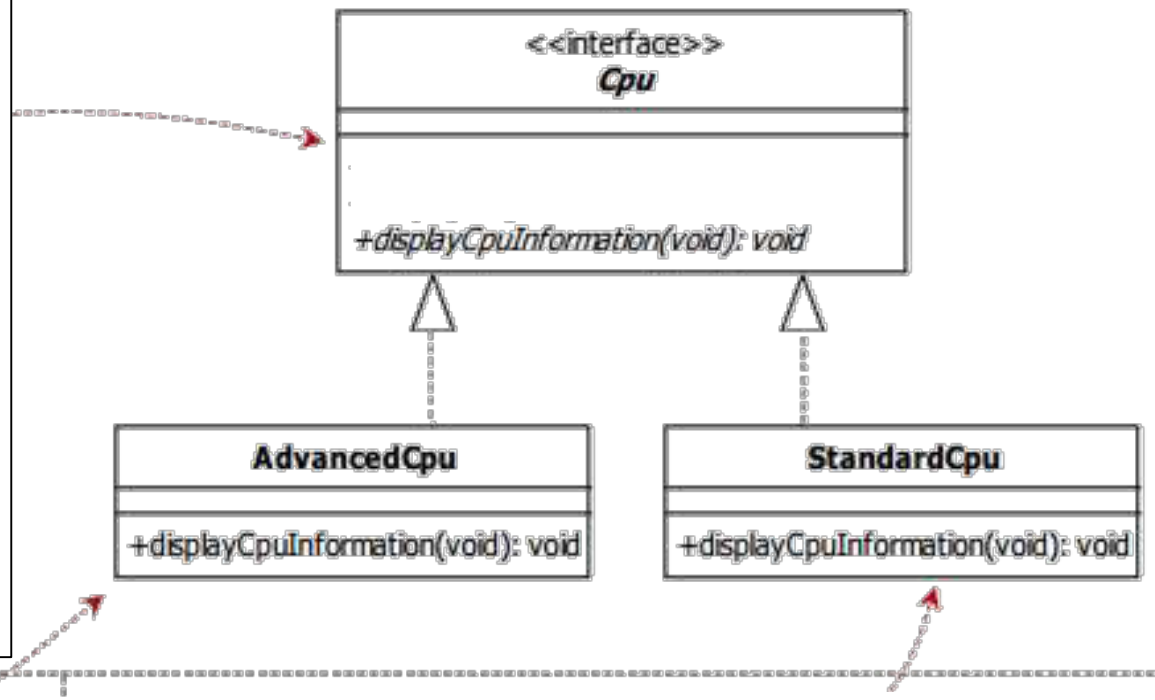

```java
// Interface for Monitor
public interface Monitor {
    // Method for displaying Monitor information.
    void displayMonitorInformation();
}
```




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

The CPU Product Design

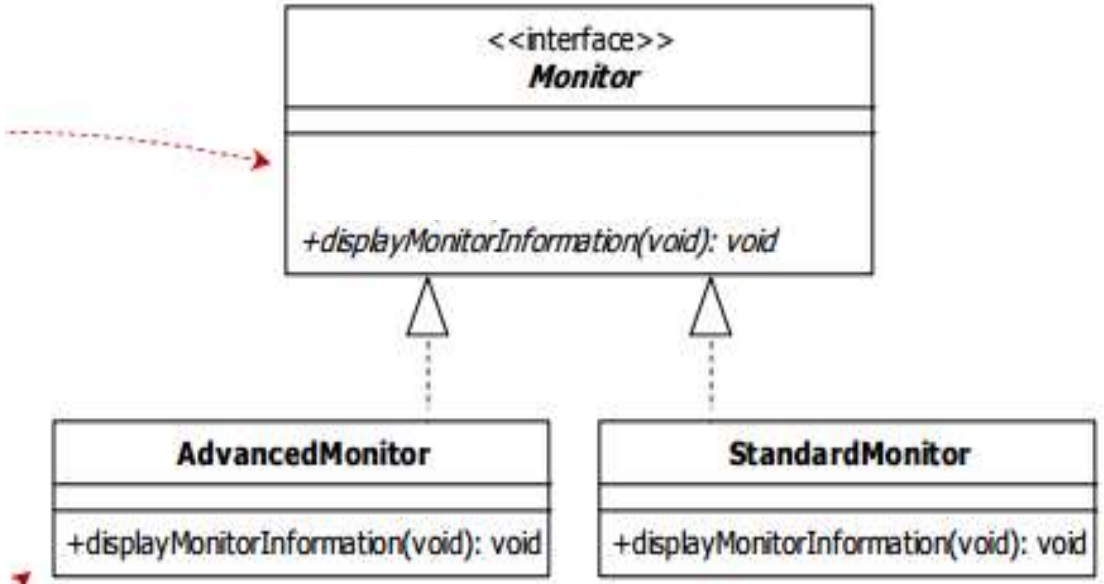
```java
// Cpu.java
public interface Cpu {
    void displayCpuInformation();
    // ... other methods for the Cpu class
}
```




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

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

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

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

The monitor Product Design



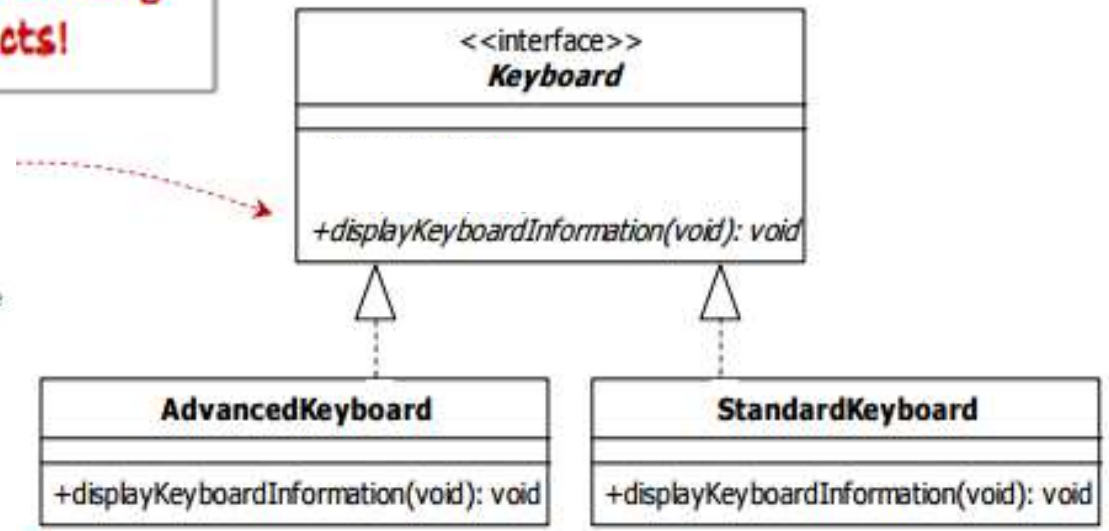

```java
// Monitor.java
public interface Monitor {
    void displayMonitorInformation();
    // ... other methods for the Monitor interface
}
```

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

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

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

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

The keyboard Product Design

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




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

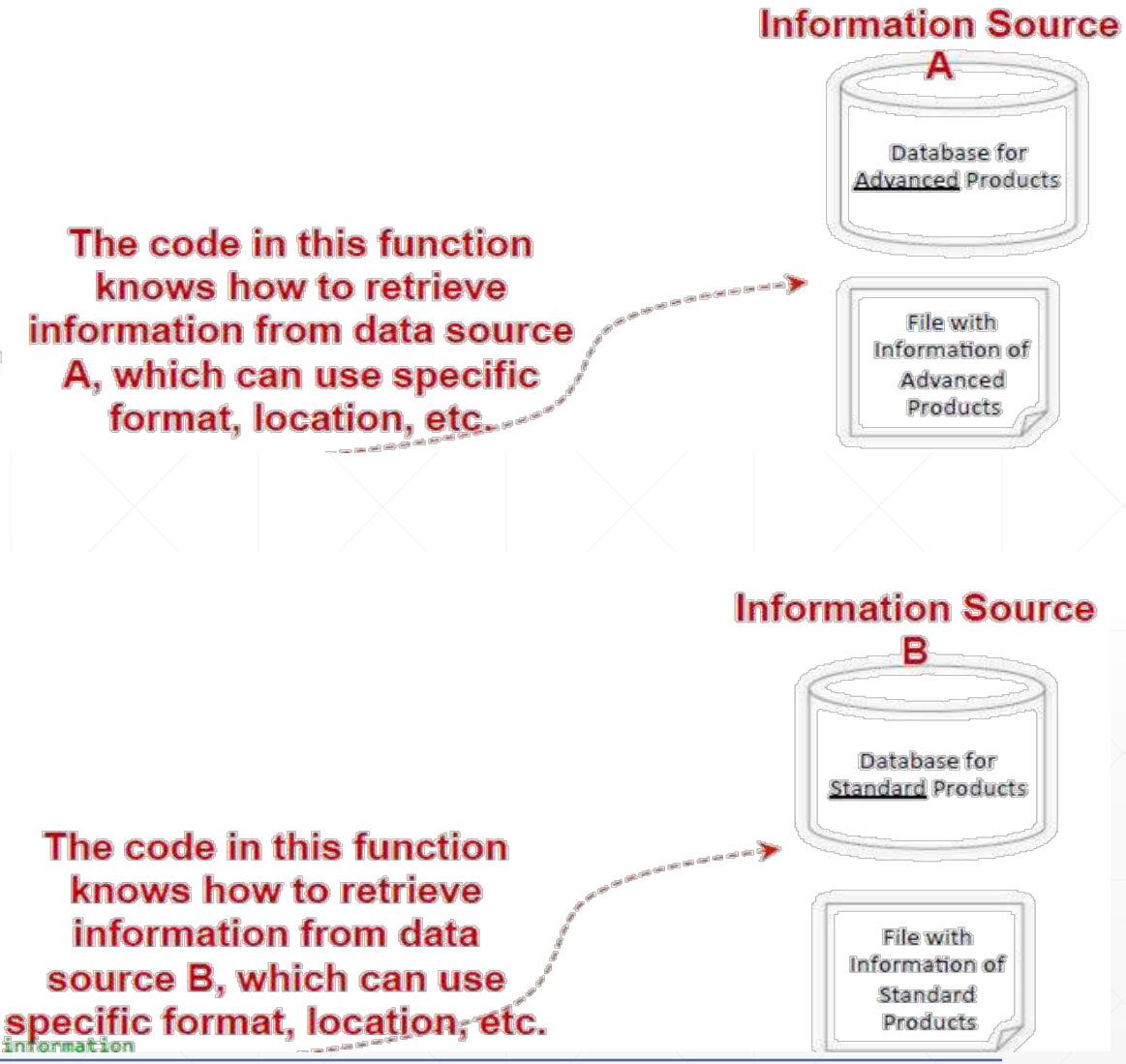
```java
// AdvancedKeyboard.java
public class AdvancedKeyboard implements Keyboard {
    public AdvancedKeyboard() {
        // Constructor
    }

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

```java
// 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
}
```

The CPU Product Implementation

```java
// 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");
    }
}
```

```java
// 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");
    }
}
```



Other Product Implementation

```java
// 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");
}
```

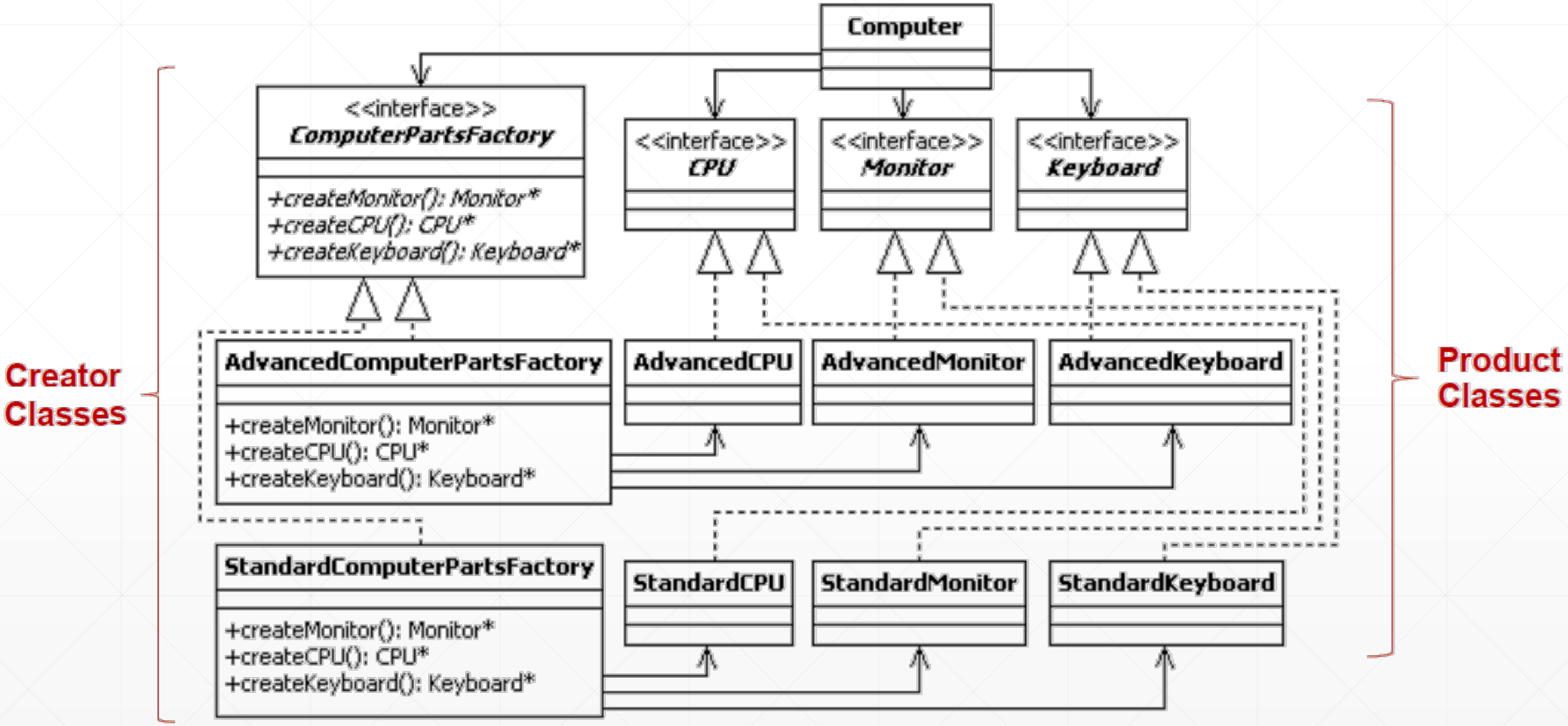
```java
// Method for retrieving the keyboard's information
public void displayKeyboardInformation() {
    // Since this is an example, we will assume that the standard keyboard's information
    // will be retrieved from information source B, e.g., database B, file B, etc.
    System.out.println("\nInformation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

All other products are implemented using the same pattern!

```java
// 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");
}
```

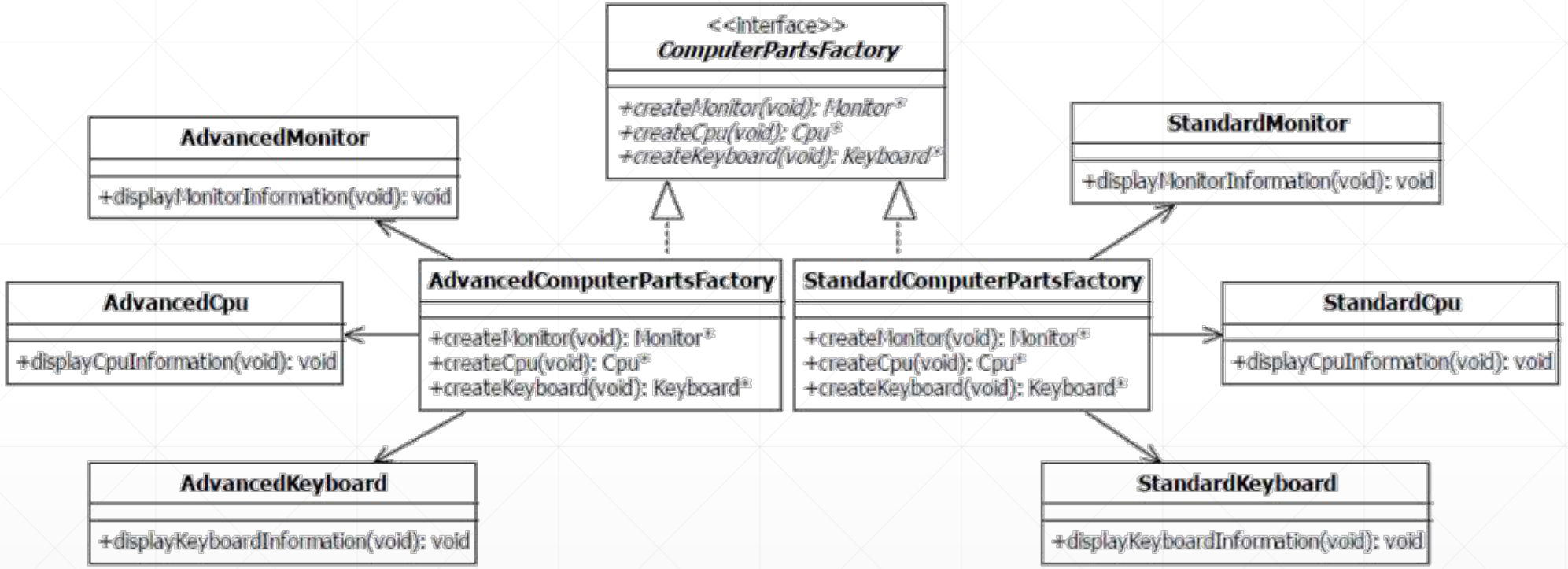
```java
// Method for retrieving the monitor's information
public void displayMonitorInformation() {
    // Since this is an example, we will assume that the standard monitor's information
    // will be retrieved from information source B, e.g., database B, file B, etc.
    System.out.println("\nInformation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

The Abstract Factory Pattern - Example



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

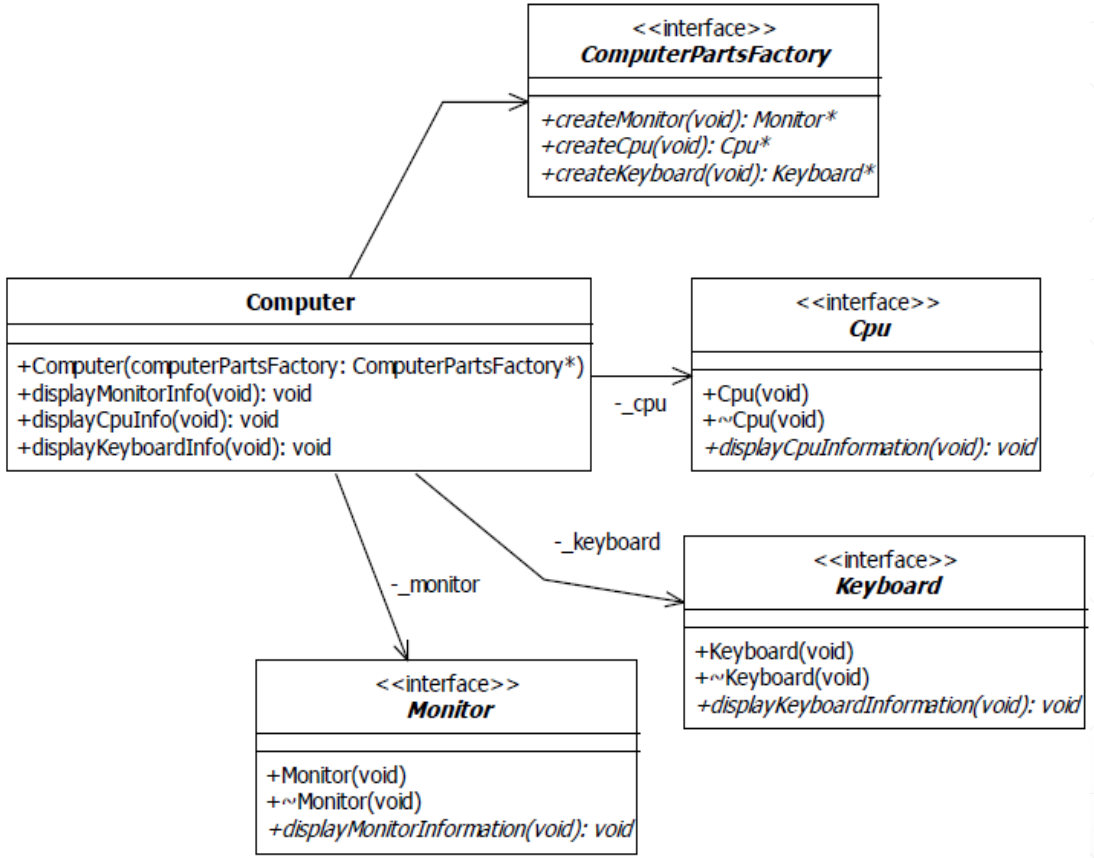
Factory Interfaces and Concrete Factories



Important:
This design connects the products designed in the previous slides with the factories used to 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.
}
```

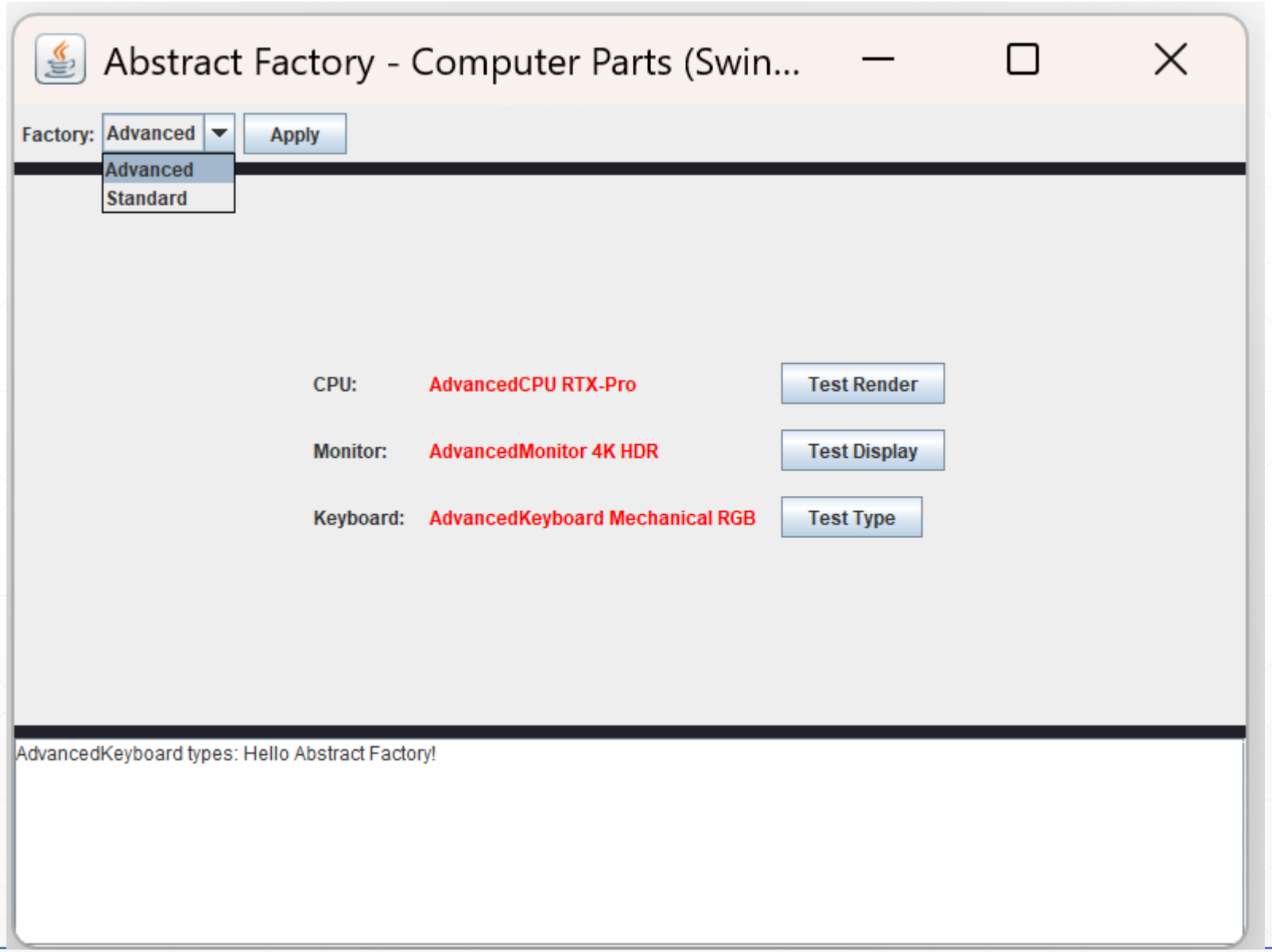


The Abstract Factory Example – Putting it all together

```
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 retreived from source A.
Displaying the advanced monitor's information.


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


Information retreived 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:
```

Extra:

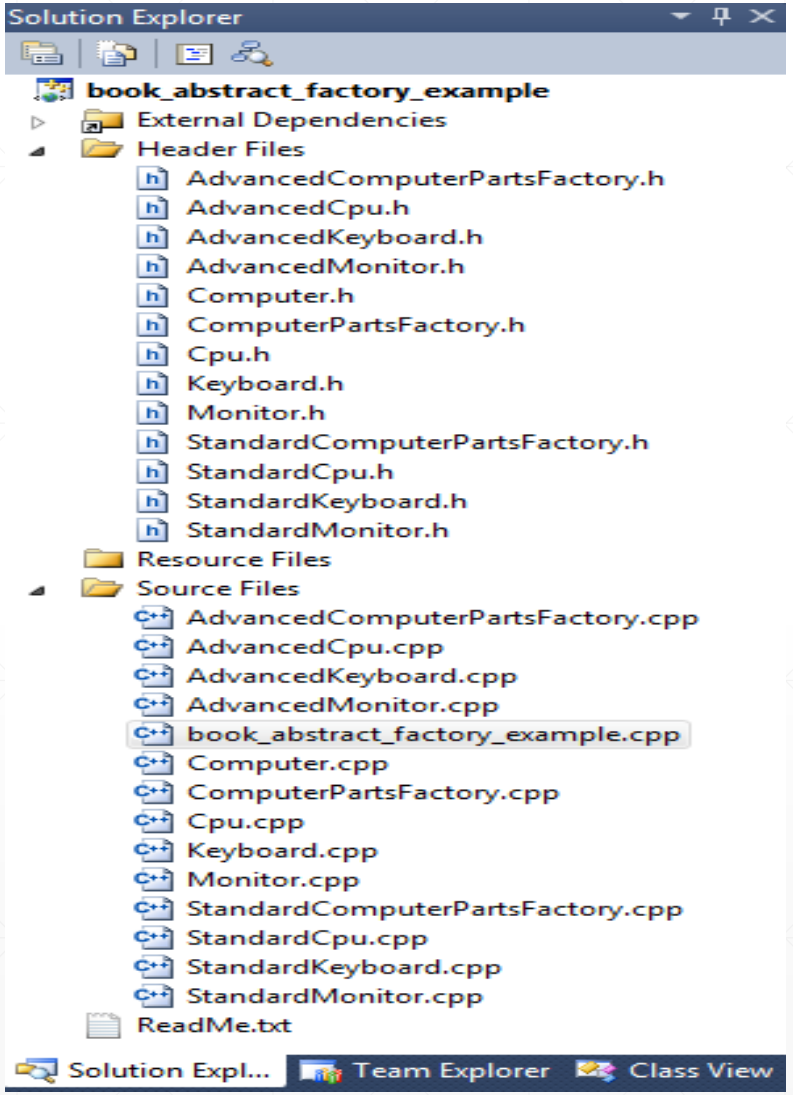

Abstract Factory - Computer Parts (Swin...
Factory:
Advanced
Apply
Advanced
Standard
CPU:
AdvancedCPU RTX-Pro
Test Render
Monitor:
AdvancedMonitor 4K HDR
Test Display
Keyboard:
AdvancedKeyboard Mechanical RGB
Test Type
AdvancedKeyboard types: Hello Abstract Factory!

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.

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

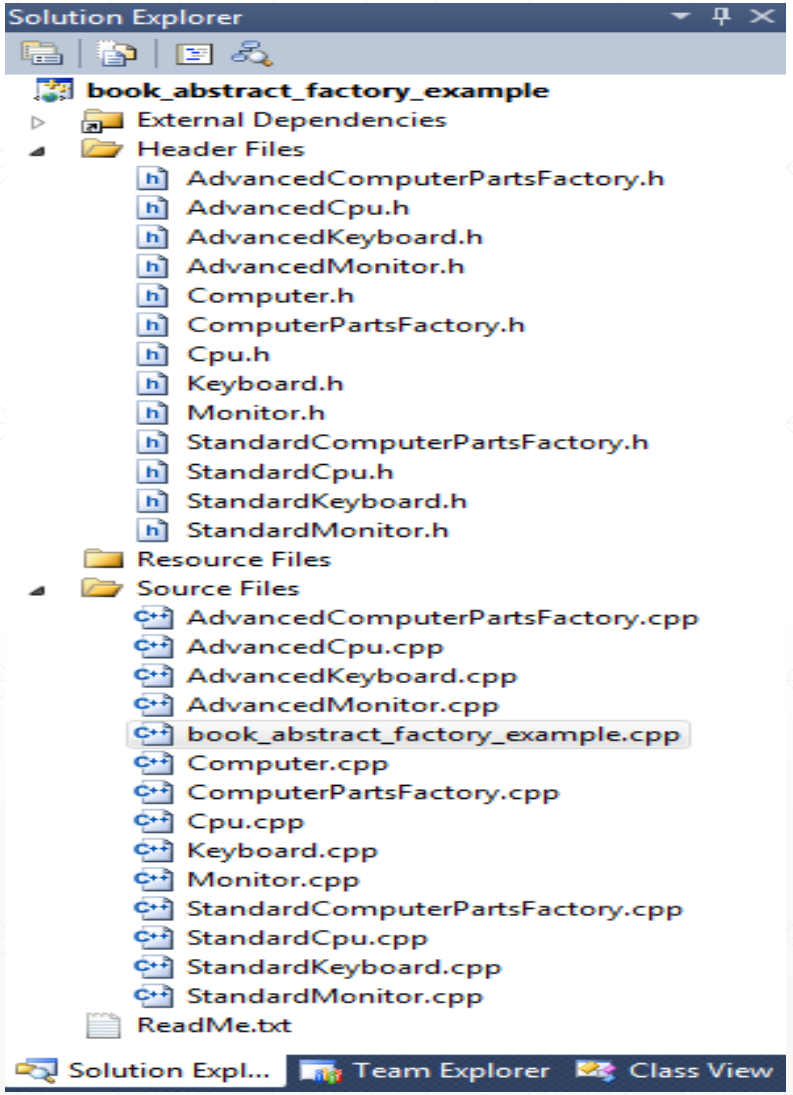


Abstract Factory – Step by Step Summary

- 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.

Cons & Pros of Abstract Factory

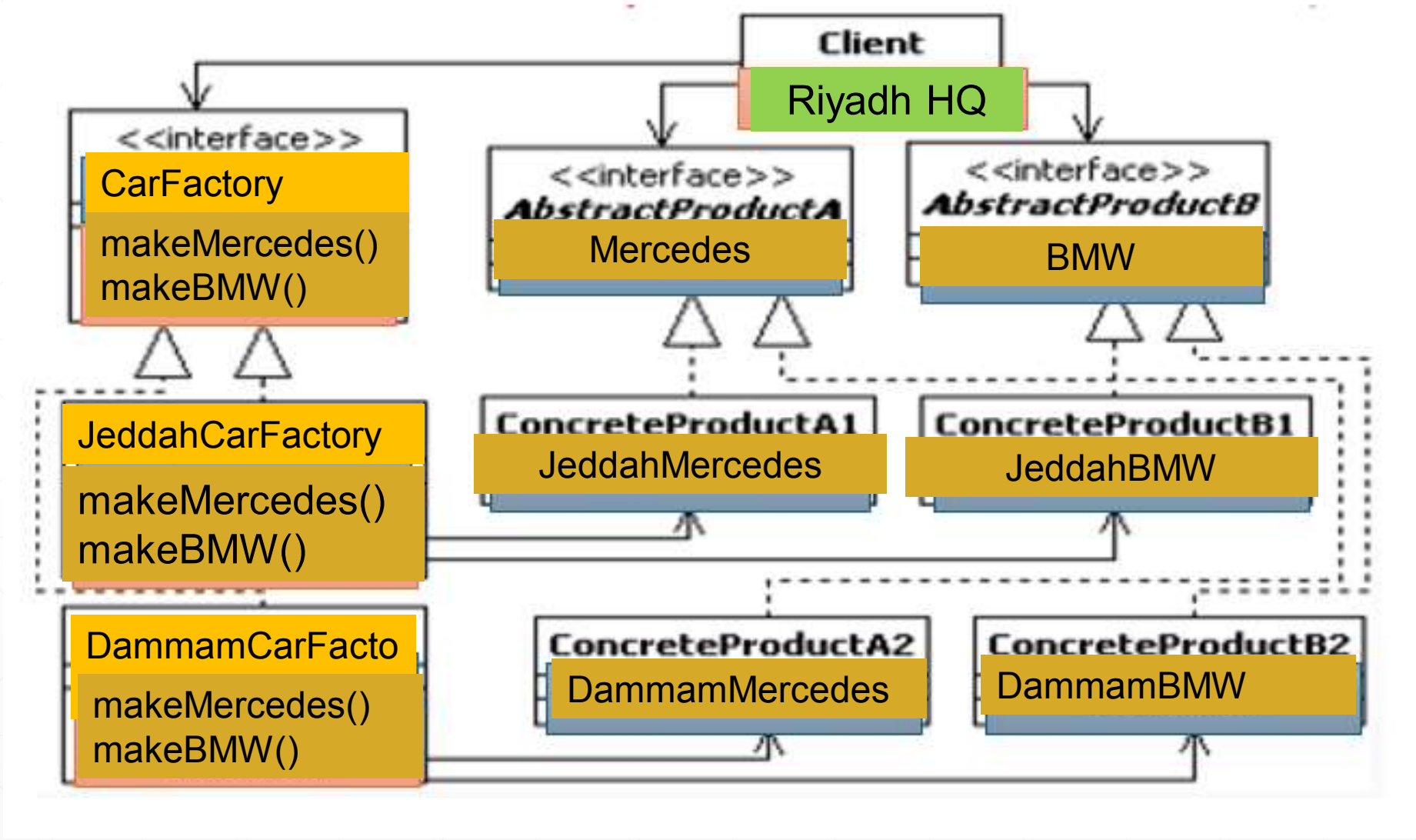
- **Cons**
  - ✓ Large number of classes are required
- **Pros**
  - ✓ Isolates concrete product classes so that reusing them becomes easier
  - ✓ Promotes consistency within specific product families.
  - ✓ Adding new families of products require no modification to existing code.
    - Additions are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



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**.

Extra


Client
Riyadh HQ
<<interface>>
CarFactory
makeMercedes()
makeBMW()
<<interface>>
AbstractProductA
Mercedes
<<interface>>
AbstractProductB
BMW
JeddahCarFactory
makeMercedes()
makeBMW()
ConcreteProductA1
JeddahMercedes
ConcreteProductB1
JeddahBMW
DammamCarFacto
makeMercedes()
makeBMW()
ConcreteProductA2
DammamMercedes
ConcreteProductB2
DammamBMW

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.

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