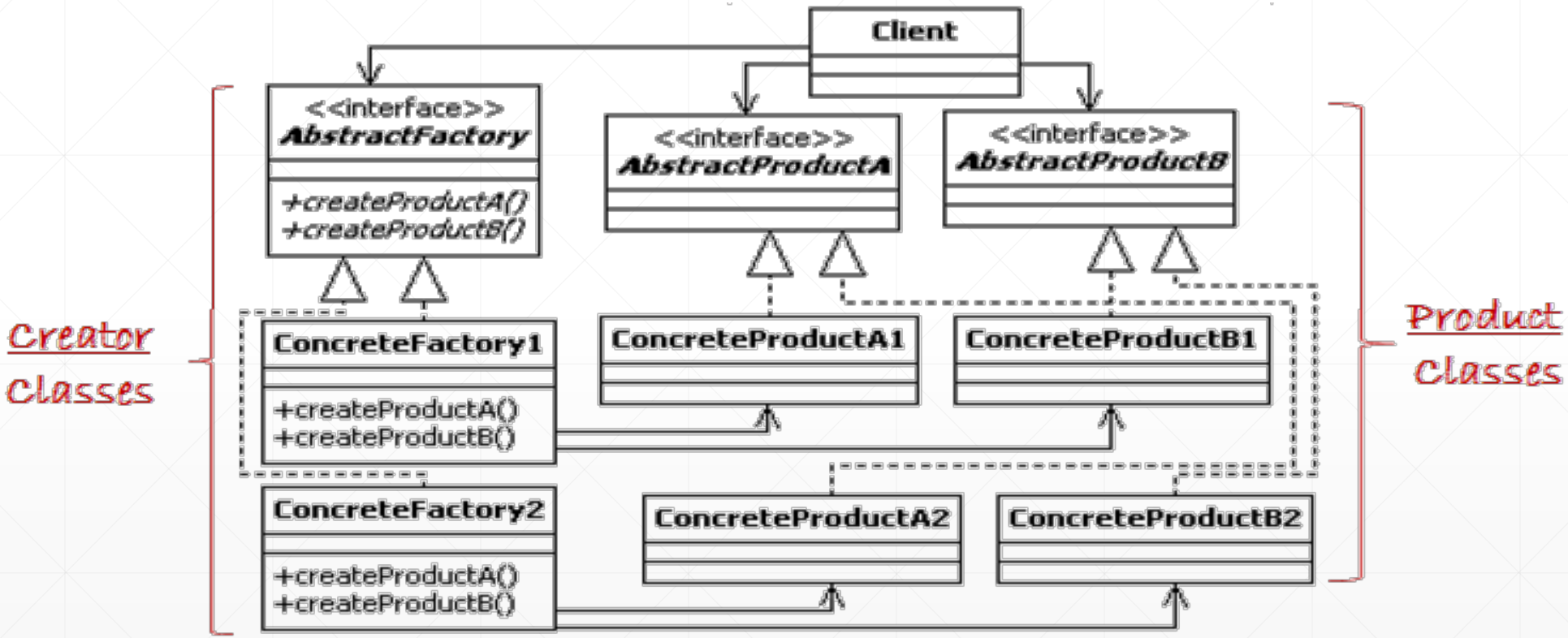
Software Design and Architecture

[Creational Design Patterns] – Chapter 06, L03

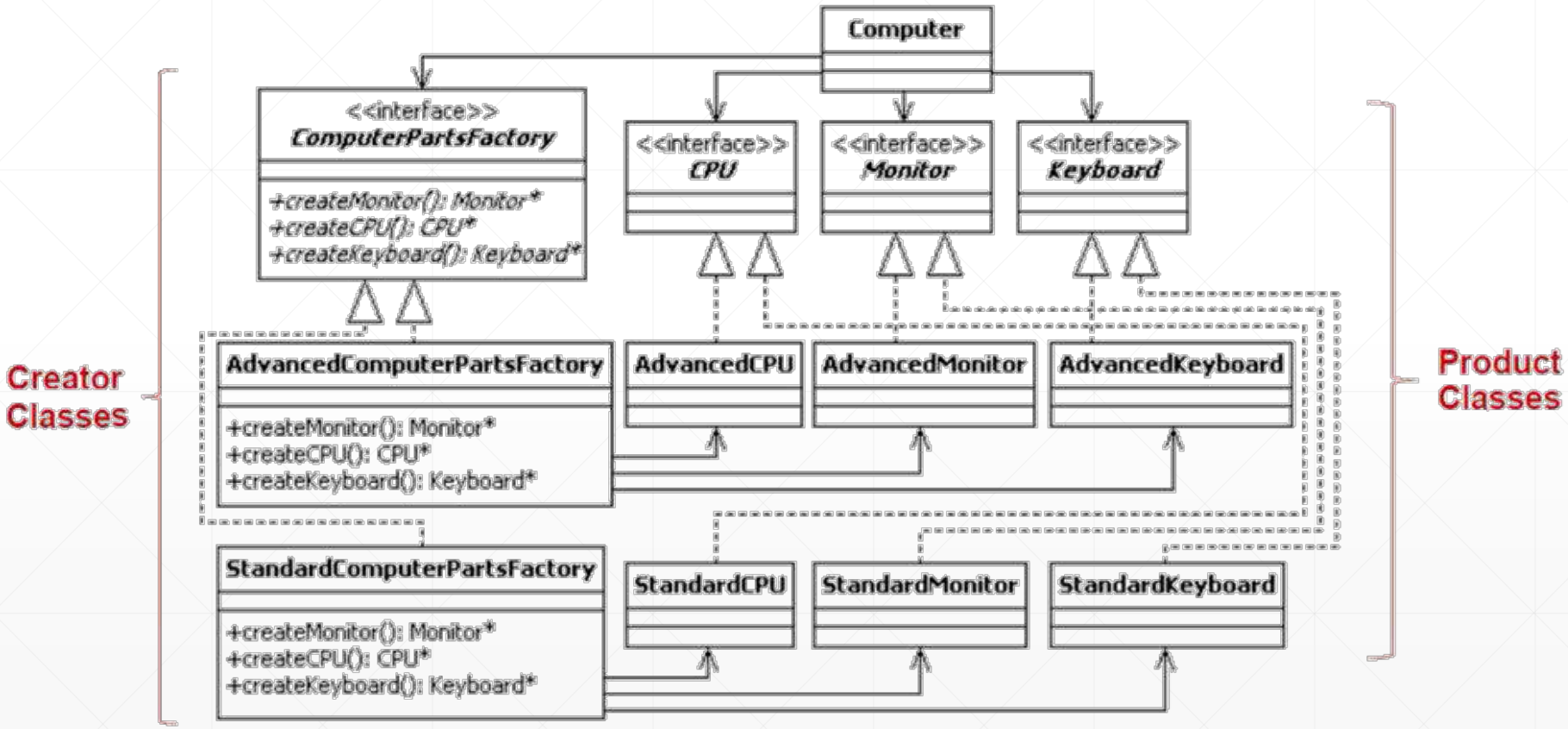
Lecture Outlines

- **Creational Patterns in Detailed Design**
  - ✓ **Previously:**
    - Abstract Factory Pattern
    - Factory Method Pattern
  - ✓ **Builder**
    - Code generator example
  - ✓ **Singleton**
    - Event manager example
- **What's next…**

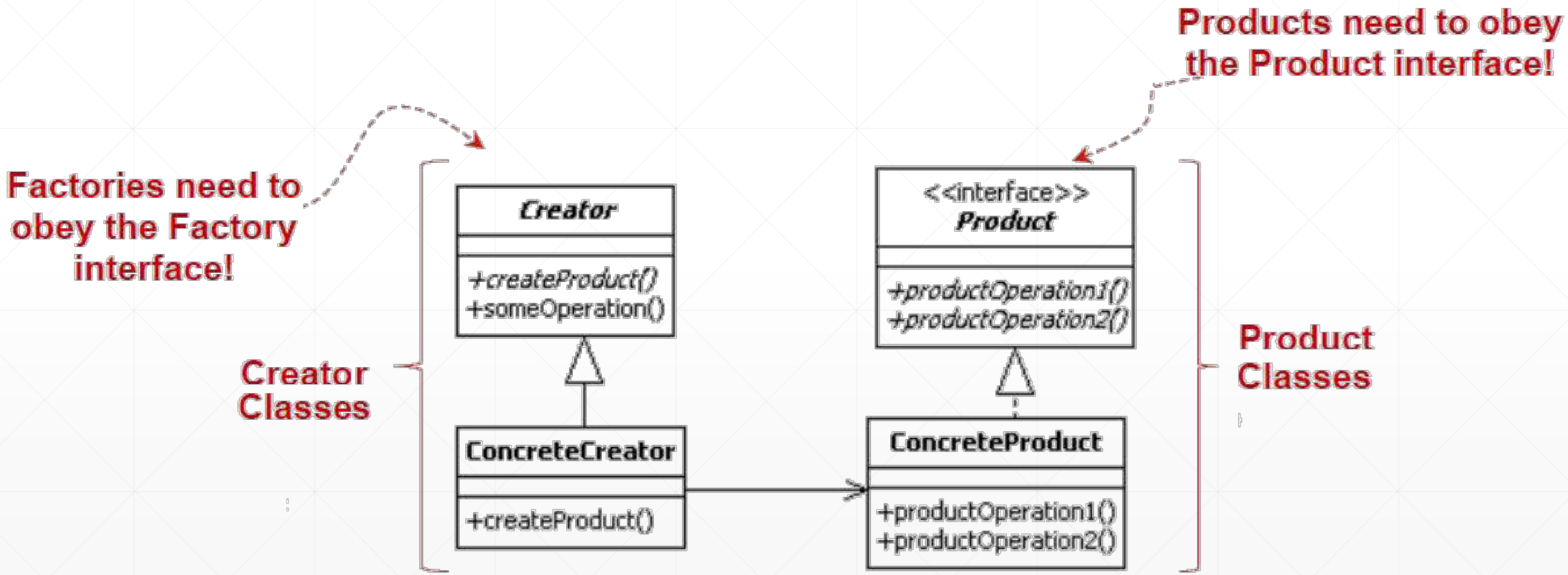
Abstract Factory Pattern – Revised



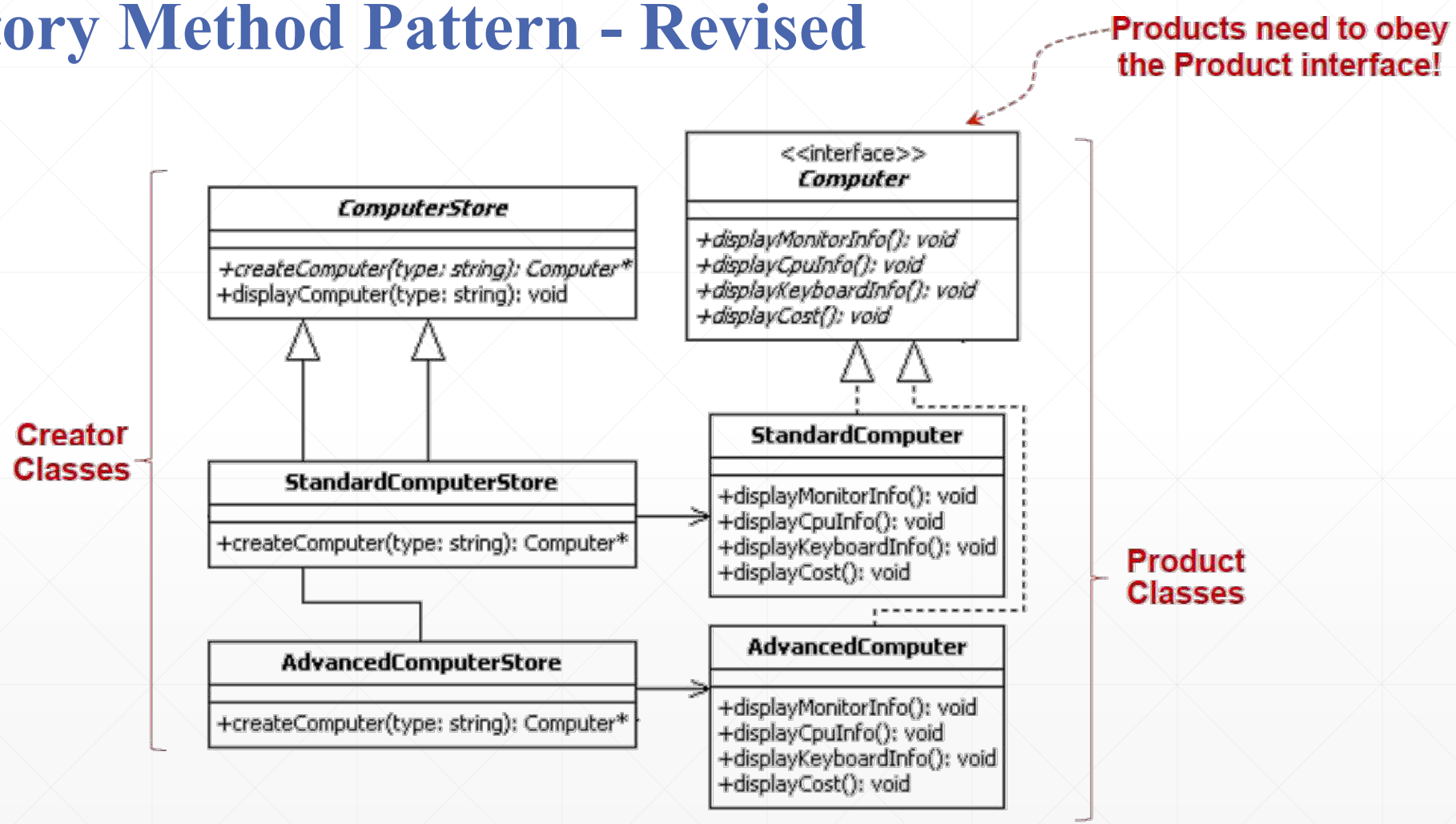
Abstract Factory Pattern – Revised



Factory Method Pattern - Revised



Factory Method Pattern - Revised



The Builder Pattern

- The **intent** of the Builder is to:
  - **Separate** the **construction** of a complex object from its **representation** so that the same construction process can create different representations.

Builder Design Pattern Example

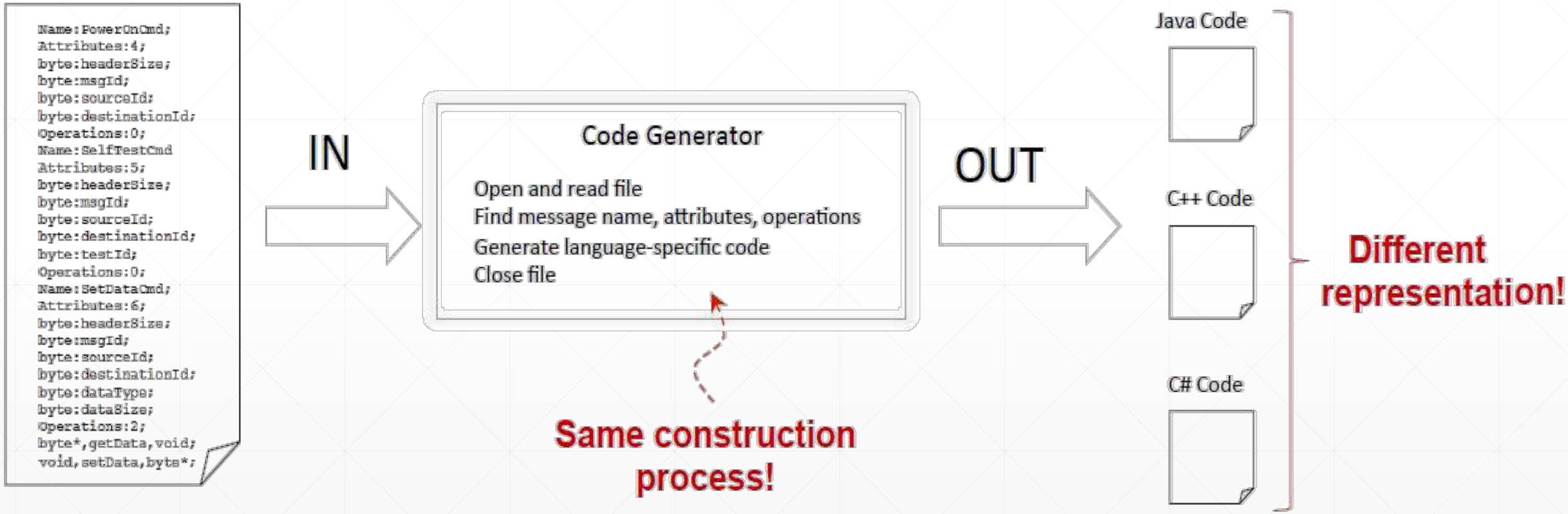
- Steps for designing using the Builder pattern:
  - ✓ Identify and design the product class (e.g., **GeneratedProduct**)
  - ✓ Identify the product's creational process and algorithm, and design a class for its execution (e.g., **CodeGenerator**).
  - ✓ Using the knowledge acquired from steps 1 and 2, design the builder interface, which specifies the parts that need to be created for the whole product to exist. These are captured as abstract interface methods that need to be implemented by derived concrete builders.
  - ✓ Identify and design classes for the different representations of the product (e.g., CppMessageBuilder, JavaMessageBuilder, etc.)

Extra:

Builder pattern

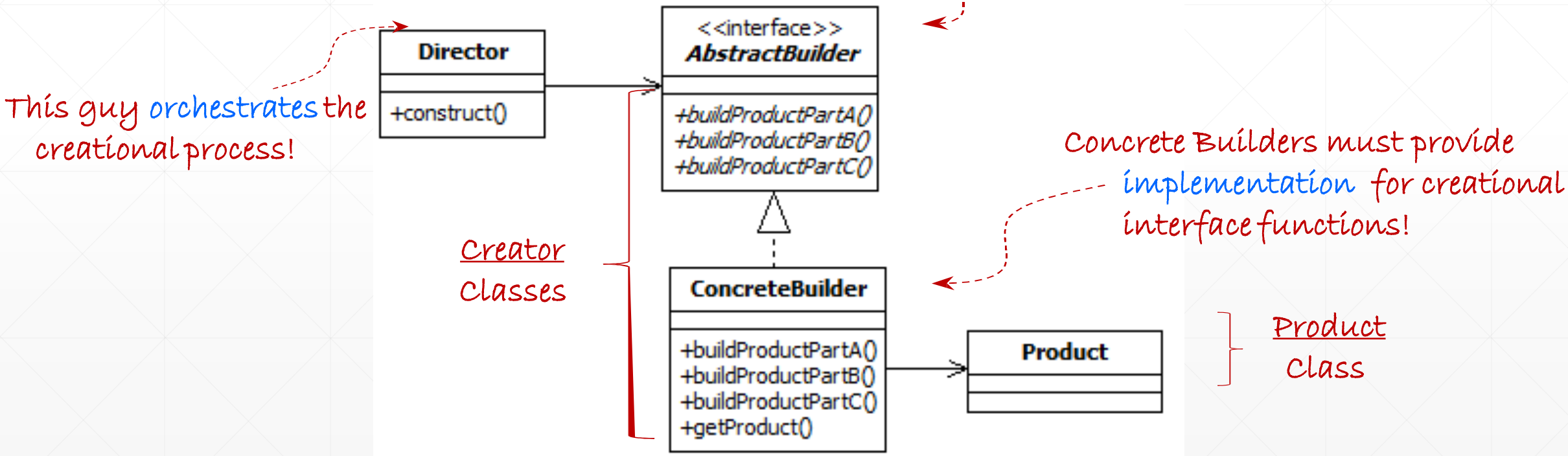
- **Intent:** separate *how we build* a complex object (steps) from *what the final object looks like* (representation).
- **When useful:** same construction steps, but you want **different outputs** (e.g., generate Java vs. C# code; make small vs. large burger).
- **Key roles:**
  - **Director** — orchestrates steps in order (doesn't know details).
  - **Builder (interface)** — declares multistep methods (buildPartA/B/C…, getProduct()).
  - **Concrete Builders** — implement those steps to produce *different representations*.
  - **Product** — the thing being built.
- **Flow (always the same):**
  - Client chooses a ConcreteBuilder and gives it to the Director
  - Director runs the steps in sequence
  - Client asks builder for the final Product
- **Builder vs Abstract Factory :**
  - **Builder** builds **one complex object step-by-step**.
  - **Abstract Factory** creates **a family of related objects all at once**.

Builder Design Pattern Example



The Builder Pattern

- Builder Structure



Extra:

➢ Builder Pattern — How to Solve (General Steps)

➢ 0) Recognize it's a Builder problem

- ✓ Same **sequence of steps**, different **final representations** (outputs).
- ✓ Need to **construct a complex object step-by-step**.
- ✓ You want to **hide construction** from the client and make adding new variants easy.

➢ 1) Define the Product

- ✓ Name the final thing being built (e.g., Report, Burger, Form, Query, House).
- ✓ List its parts/fields (what varies between variants).

➢ 2) Declare the Builder interface (steps)

- ✓ Write the **multistep API** that builds parts of the Product.

Typical methods:

buildPartA(...)
buildPartB(...)
buildPart……
getResult() (returns the Product)

Steps should describe *what to build*, not how.

3) Implement Concrete Builders (variants)

- ✓ One class per variant (e.g., PdfReportBuilder, HtmlReportBuilder).
- ✓ Each implements the steps **its own way** and stores intermediate state.
- ✓ getResult() returns the fully built Product.

4) Create the Director (optional but recommended)

- ✓ Orchestrates **order** of steps.
- ✓ Has a method like construct() that calls the steps in sequence.
- ✓ Keeps the construction process reusable and consistent.

5) Client wiring

- ✓ Client picks the **ConcreteBuilder** (the variant desired).
- ✓ Pass it to the **Director**, call construct().
- ✓ Get the Product via builder.getResult() (or director.construct() returns it).

6) Verify extensibility

- ✓ To add a new variant, create a **new ConcreteBuilder** only—client and director stay the same.

1. Product

```java
class Product {
    // the final object being built
}
```

2. Builder Interface

```java
interface Builder {
    void buildPartA();
    void buildPartB();
    Product getResult();
}
```

3. Concrete Builders

```java
class JavaBuilder implements Builder {
    public void buildPartA() { /* build step A for Java */ }
    public void buildPartB() { /* build step B for Java */ }
    public Product getResult() { return new Product(); }
}

class CSharpBuilder implements Builder {
    public void buildPartA() { /* build step A for C# */ }
    public void buildPartB() { /* build step B for C# */ }
    public Product getResult() { return new Product(); }
}
```

4. Director

```java
class Director {
    private Builder builder;

    Director(Builder builder) {
        this.builder = builder;
    }

    Product construct() {
        builder.buildPartA();
        builder.buildPartB();
        return builder.getResult();
    }
}
```

5. Client Code

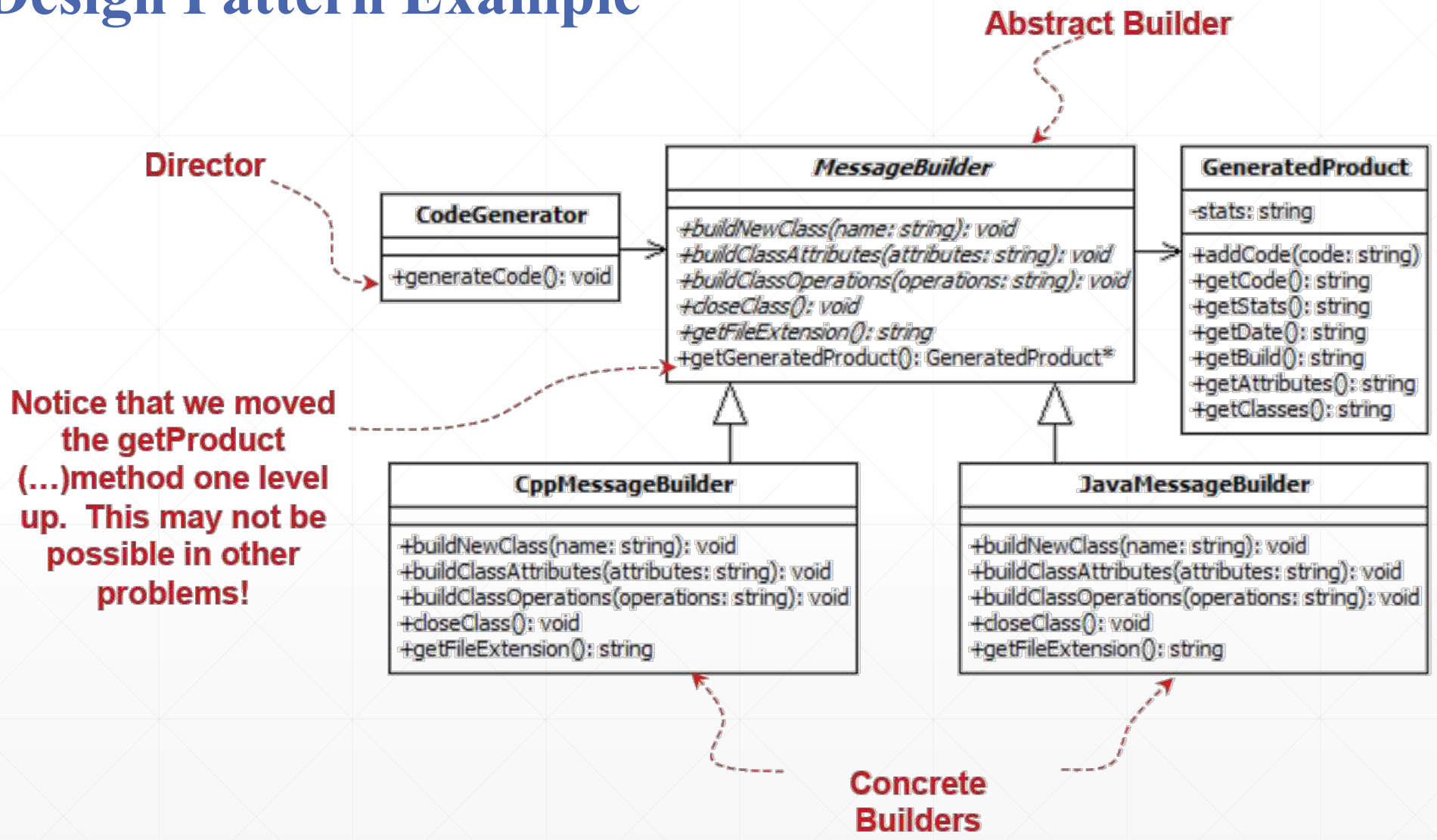
```java
public class Demo {
    public static void main(String[] args) {
        // Use Java builder
        Builder javaBuilder = new JavaBuilder();
        Director director = new Director(javaBuilder);
        Product javaProduct = director.construct();

        // Use C# builder
        Builder csBuilder = new CSharpBuilder();
        Director director2 = new Director(csBuilder);
        Product csProduct = director2.construct();
    }
}
```

•Points to Remember

- **Product** = the final object.
- **Builder interface** = declares building steps.
- **Concrete Builders** = implement steps differently (Java vs. C#).
- **Director** = controls the order of building steps.
- **Client** = chooses which builder to use.

☞ Same **construction steps**, but **different final representations**.

Builder Design Pattern Example



The Builder provides the following benefits

- Separates an object's construction process from its representation; therefore future representations can be added easily to the software.
- Changes to existing representations can be made without modifying the code for the creational process.
- Provides finer control over the construction process so that parts of objects can be created at discrete points in time (which differs from the Abstract Factory pattern)
- Improves maintainability, testability, and reusability.

Extra:

🔹 Idea in Simple Words

The **Builder Pattern** is used when you want to create a **complex object step by step**, instead of building it all at once.

👉 Think of it like ordering food:

- You don't get the meal instantly
- You choose parts (bread, meat, sauce...)
- Then it gets assembled step by step

🔹 Real-Life Example: 🍔 Burger Builder

Imagine you are building a burger:

❌ Without Builder (hard way)

You create everything at once:

- Burger("big bun", "double meat", "cheese", "ketchup", "lettuce")

➡️ Problem:

- Hard to read
- Hard to customize
- Easy to make mistakes

✅ With Builder Pattern

You build it step by step:

```java
Burger burger = new BurgerBuilder()
                    .addBun("Big Bun")
                    .addPatty("Beef")
                    .addCheese()
                    .addLettuce()
                    .addSauce("Ketchup")
                    .build();
```

Question:

A restaurant's system needs to prepare different types of burgers such as **Beef Burger** and **Chicken Burger**. Each burger must always be prepared in the same sequence of steps:

1. Add Bun
2. Add Patty
3. Add Sauce

The ingredients, however, differ depending on the type of burger:

- **Beef Burger:** Sesame Bun + Beef Patty + BBQ Sauce
- **Chicken Burger:** Whole Wheat Bun + Chicken Patty + Mayo Sauce

You are asked to **design and implement** this system in Java such that:

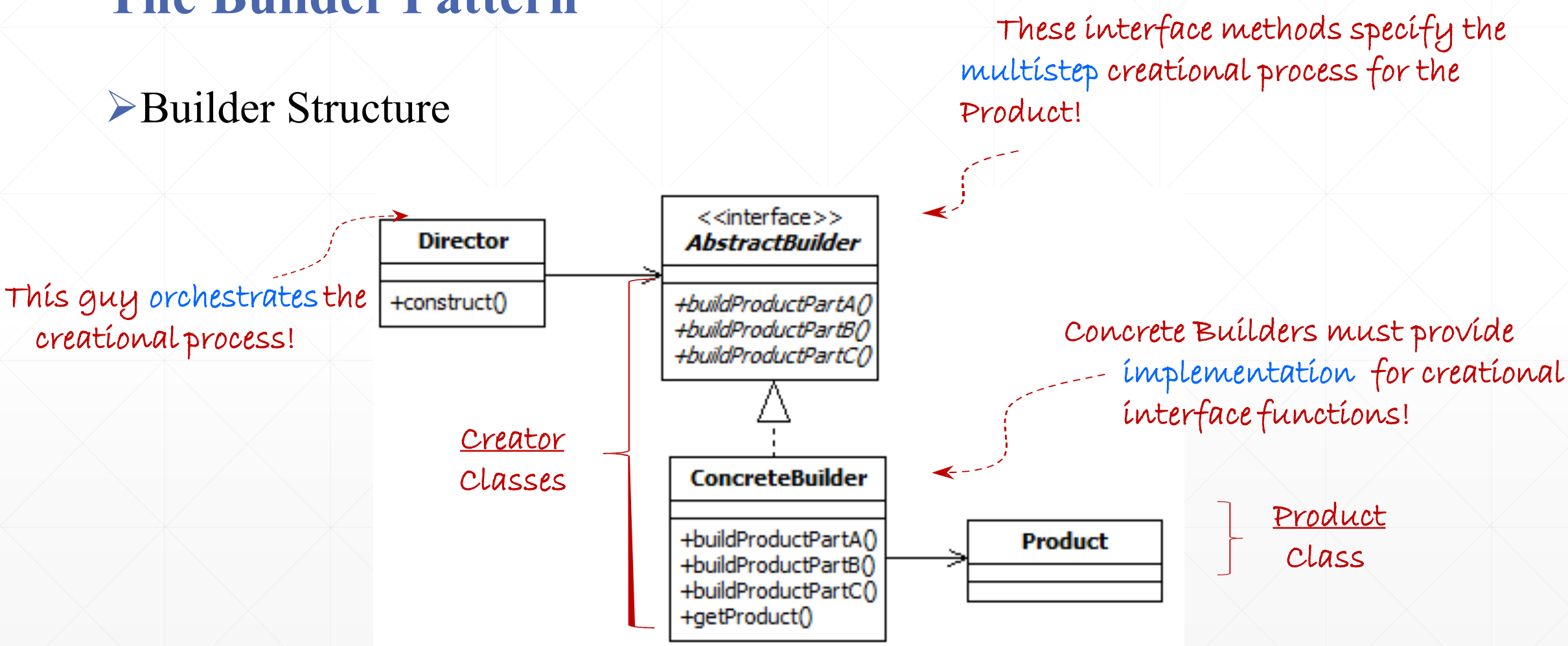
- The construction steps are the same for all burgers.
- New types of burgers can be added easily in the future.
- The client should only decide the burger type, while the actual construction process remains hidden.

Tasks:1- draw class diagram based on builder pattern

2- write the java code

The Builder Pattern

- Builder Structure



1. Product (Burger)

```java
class Burger {
    String bun;
    String patty;
    String sauce;

    public String toString() {
        return bun + " + " + patty + " + " + sauce;
    }
}
```

2. Builder Interface

```java
interface BurgerBuilder {
    void addBun();
    void addPatty();
    void addSauce();
    Burger getBurger();
}
```

3. Concrete Builders (Different Recipes)

```java
class BeefBurgerBuilder implements BurgerBuilder {
    Burger burger = new Burger();

    public void addBun()   { burger.bun = "Sesame Bun"; }
    public void addPatty() { burger.patty = "Beef Patty"; }
    public void addSauce() { burger.sauce = "BBQ Sauce"; }
    public Burger getBurger() { return burger; }
}

class ChickenBurgerBuilder implements BurgerBuilder {
    Burger burger = new Burger();

    public void addBun()   { burger.bun = "Whole Wheat Bun"; }
    public void addPatty() { burger.patty = "Chicken Patty"; }
    public void addSauce() { burger.sauce = "Mayo Sauce"; }
    public Burger getBurger() { return burger; }
}
```

4. Director

```java
class Chef {
    private BurgerBuilder builder;

    Chef(BurgerBuilder builder) {
        this.builder = builder;
    }
// construct() = step-by-step process
public Burger construct() {
    builder.addBun();
    builder.addPatty();
    builder.addSauce();
    return builder.getBurger();
}
}
```

5. Client Code

```java
public class Demo {
    public static void main(String[] args) {
        Chef beefChef = new Chef(new BeefBurgerBuilder());
        Burger beefBurger = beefChef.construct();
        System.out.println("Built: " + beefBurger);

        Chef chickenChef = new Chef(new ChickenBurgerBuilder());
        Burger chickenBurger = chickenChef.construct();
        System.out.println("Built: " + chickenBurger);
    }
}
```

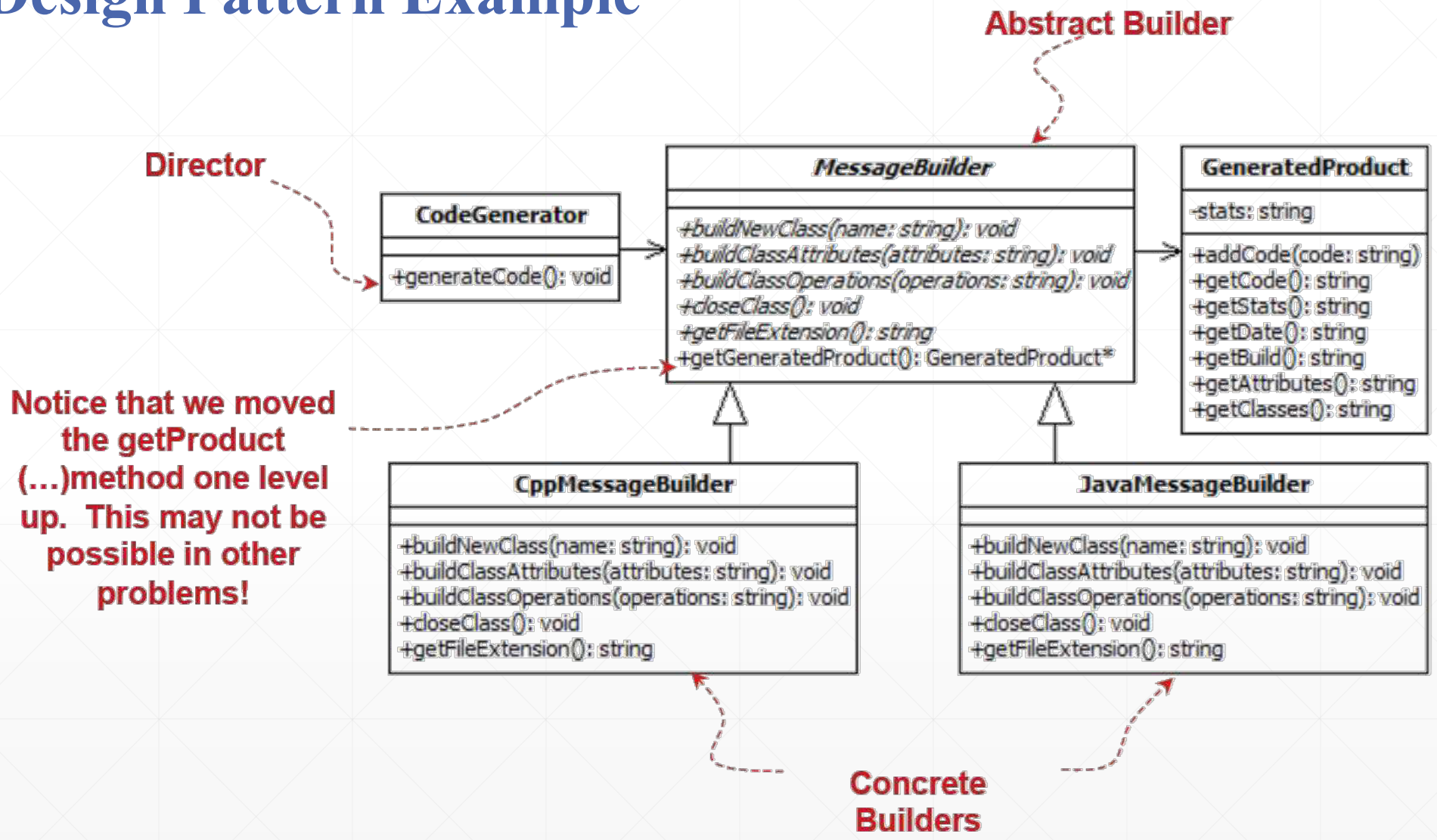
Builder Design Pattern

- The Builder Design pattern is an object **creational pattern** that encapsulates both the creational process and the representation of product objects.
- In the **Abstract Factory** Pattern, various product objects are created all at once,
- The **Builder** Pattern allows clients to control the (**multistep**) creational process of a **single product object.**
- The Builder dictates the creation of **individual parts** of the object at **discrete points** throughout software operations.

Builder Design Pattern Example

- Consider a **code generator**:
  - ✓ Code generated from an interface specification document (ICD) used to monitor and control custom developed hardware.
  - ✓ Company sells hardware, but also includes library to monitor and control its hardware.
  - ✓ Generator reads a text document containing information about a messaging specification. It then generates C++ classes that can be used to monitor and control hardware. This is also shipped to customers so that they can use the hardware easily.
  - ✓ The ICD is quite complex, and customers are demanding support for other languages, such as Java and C#.

Builder Design Pattern Example



Builder Design Pattern Example

The Builder Interface for generating code!

The same steps used to create products in all target languages!

```
// Forward reference.
class GeneratedProduct;

class MessageBuilder {

public:
  // The interface method for building a new class.
  virtual void buildNewClass(string name) = 0;

  // The interface method for building class attributes.
  virtual void buildClassAttributes(string attributeList) = 0;

  // The interface method for building class operations.
  virtual void buildClassOperations(string operationList) = 0;

  // The interface method for closing a new class.
  virtual void closeClass() = 0;

  // The file extension for the target programming language.
  virtual string getFileExtension() = 0;

  // Return the generated product.
  GeneratedProduct* getGeneratedProduct() {

    return _codeProduct;
  }

private:
  // The product containing generated code and stats about code generated.
  GeneratedProduct* _codeProduct;
};
```

Builder Design Pattern Example

```
class CppMessageBuilder : public MessageBuilder {

public:
```

C++ Builder

```
  // The interface method for building a new class.
  virtual void buildNewClass(string name) {

    // Generate code for creating a class using CPP style and the name
    // argument.

    // Once code is generated, add it to the product.
    getGeneratedProduct()->addCode(/*new C++ class code*/);
  }

  // The interface method for building class attributes.
  virtual void buildClassAttributes(string attributeList) {

    // For all items in attributeList, generate attributes using CPP style
    // and add them to the generated code.

    // Once code is generated, add it to the product.
    getGeneratedProduct()->addCode(/*C++ attributes*/);
  }

  // The interface method for building class operations.
  virtual void buildClassOperations(string operationList) {

    // For all items in operationList, generate operations using CPP style
    // and add them to the generated code.

    // Once code is generated, add it to the product.
    getGeneratedProduct()->addCode(/*C++ operations*/);
  }

  // The interface method for closing a new class.
  virtual void closeClass() {

    // Generate code to close a class in Cpp, and add it to the generated
    // code.

    // Once code is generated, add it to the product.
    getGeneratedProduct()->addCode("\n};\n\n");
  }
};
```

Step 1

```
class MessageOne
{
```

Step 2

```
class MessageOne
{
  private:
    int attributeOne;
    int attributeTwo;
```

Step 3

```
class MessageOne
{
  private:
    int attributeOne;
    int attributeTwo;

  public:
    void setValueOne(unsigned char x);
```

Step 4

```
class MessageOne
{
  private:
    int attributeOne;
    int attributeTwo;

  public:
    void setValueOne(unsigned char x);
};
```

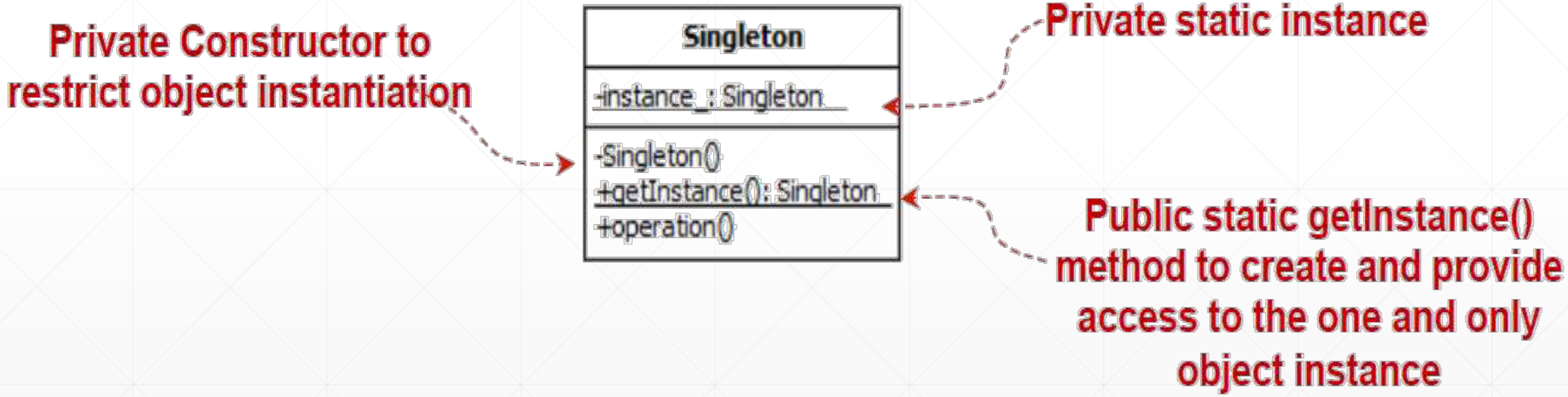
C++ Product Created!

Singleton Design Pattern

- The Singleton design pattern is an object **creational** design pattern used to prevent objects from being instantiated more than once in a running program.

- The **intent** of the Singleton is to
  - ✓ **Ensure** a **class** only has **one instance**, and provide a **global** point of access to it.

Singleton Design Pattern



Singleton Design Pattern Example

Problem:

Consider an application that requires **event logging** capabilities. The application consists of many different objects that generate events to keep track of their actions, status of operations, errors, or any other information of interest. A decision is made to **create an event manage** that can be accessed by all objects and used to **manage all events in the system**. Upon instantiation, the event manager creates an **event list** that gets updated as events are logged. At specific points during the software system's operation, these events are written to a file. To prevent conflicts, it is desirable that at any given time, there is **only one instance of the event manager** executing.

Extra:

- The application needs **event logging** (recording actions, statuses, and errors).
- Many objects in the system generate events, but instead of each one handling logging separately, a **central Event Manager** is created.
- This Event Manager keeps an **event list**, updates it when events happen, and later writes the events to a file.
- To avoid problems (like multiple versions writing at the same time), the design ensures there is **only one Event Manager instance** in the whole system.

Extra:

Why Is There a Problem?

In many software systems:

•Multiple **parts of the application** (e.g. login system, payment system, notification system) need to **log events** — such as user actions, errors, system updates.

•If each part **creates its own logger**, this leads to:

- ❌ **Multiple log files** or conflicting writes.
- ❌ **Confusing or inconsistent data**.
- ❌ **Harder to debug** or audit the system.

● **The real problem:** If every part creates a new logger (event manager), it causes **conflicts**, **redundancy**, and **loss of control**.

Why Singleton Fixes This?

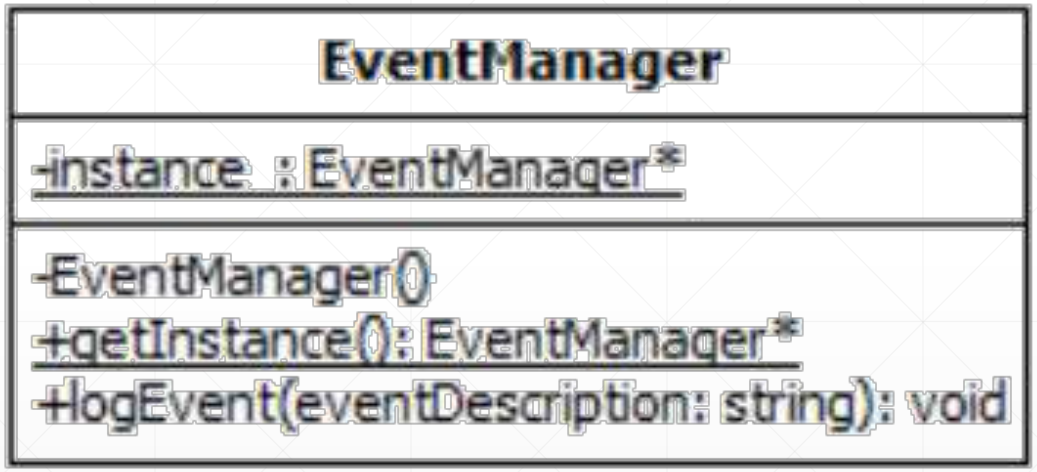
•Singleton ensures that **only one instance** of the **EventManager (Logger)** exists.

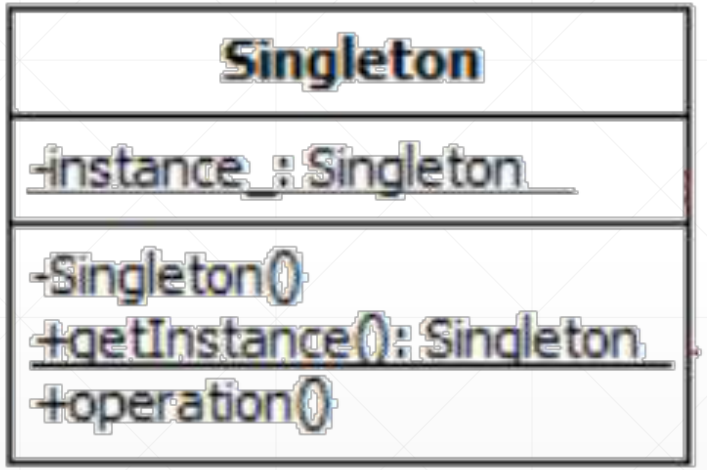
•All parts of the system **share the same logger**.

•You maintain **one log file**, centralized control, and avoid conflicts.

✅ One object → One responsibility → No conflict.

Singleton Design Pattern Example





Java Code for the Singleton Pattern

A **static** variable is a variable that has been allocated "statically", meaning that its lifetime (or "extent") is the entire run of the program.

The Singleton class

```
// File Name: Singleton.java
public class Singleton {

   private static Singleton singleton = new Singleton( );

   /* A private Constructor prevents any other
    * class from instantiating.
    */
   private Singleton() { }

   /* Static 'instance' method */
   public static Singleton getInstance( ) {
      return singleton;
   }

   /* Other methods protected by singleton-ness */
   protected static void demoMethod( ) {
      System.out.println("demoMethod for singleton");
   }
}
```

Java Code for the Singleton Pattern

```
// File Name: SingletonDemo.java
public class SingletonDemo {

   public static void main(String[] args) {
      Singleton tmp = Singleton.getInstance( );
      tmp.demoMethod( );
   }
}
```

The main program file where we will create a singleton object

Singleton Design Pattern Example

- Steps to design using the Singleton design pattern:
  1. Set the visibility of the **constructor** to **private**
  2. Define a private static member attribute that can store a reference (i.e., a pointer) to the one instance of the singleton.
  3. Create a public static getInstance() method that can access the private constructor to instantiate one object of the singleton type and return it to clients.

Benefits of the Singleton Design Pattern

➢Benefits:

- ✓It provides controlled access to a single instance of a given type.
- ✓It can be customized to permit a variable number of instances.

➢Drawback:

- ✓It does not work well in multithreading environments

Extra: Real-World Scenario
Let's say you're building a Banking App:
Login System logs when a user logs in.
Transaction System logs when money is transferred.
Notification System logs when a message is sent.
☞ You want all events to go to the same log, in the correct order.

Step 1: Define the Singleton Class `EventLogger`

```java
public class EventLogger {
    private static EventLogger instance;

    // Private constructor to prevent outside creation
    private EventLogger() {}

    // Global access point
    public static EventLogger getInstance() {
        if (instance == null) {
            instance = new EventLogger();
        }
        return instance;
    }
   // Method to log messages
   public void log(String message) {
       System.out.println("Log: " + message);
       // In real use: write to a file
   }
}
```

Step 2: Use Singleton in Different Parts of the App

Login System

```java
public class LoginSystem {
    public void login(String username) {
        // Do Login work...
        EventLogger.getInstance().log("User " + username + " logged in.");
    }
}
```

Why don't we write `instance.log()` directly?

Because **`instance` is private and static** inside the class, and we're **outside** the class when we want to use it.

- `getInstance()` is a **`public static method`** → you can access it from anywhere.
- It **returns the `instance`** (the one and only object).
- Then you can call `.log()` on that object.

Extra: Transaction System

```
public class TransactionSystem {
    public void transfer(String from, String to, double amount) {
        // Do transfer work...
        EventLogger.getInstance().log("Transferred $" + amount + " from " + from + " to " + to);
    }
}
```

Notification System

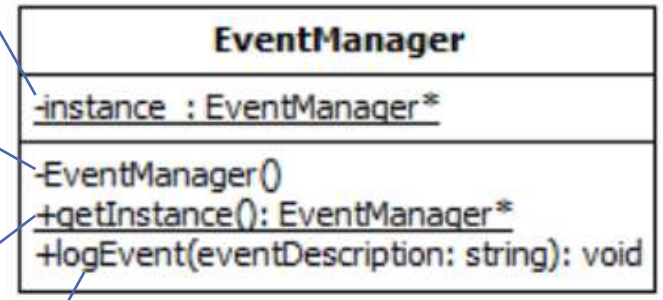
```
public class NotificationSystem {
    public void sendNotification(String user, String message) {
        // Do notification work...
        EventLogger.getInstance().log("Notification sent to " + user + ": " + message);
    }
}
```

Step 3: Main Class to Run the App

```
public class Main {
    public static void main(String[] args) {
        LoginSystem login = new LoginSystem();
        login.login("Alice");

        TransactionSystem transaction = new TransactionSystem();
        transaction.transfer("Alice", "Bob", 250.0);

        NotificationSystem notify = new NotificationSystem();
        notify.sendNotification("Alice", "Transfer completed");

        // Proof: All use the same logger instance
        System.out.println("Logger check: " +
            (EventLogger.getInstance() == EventLogger.getInstance())); // t
    }
}
```

Singleton Design Pattern Example

```
#include "EventManager.h"
#include <iostream>

using namespace std;

// Initialize the instance_ static member attribute.
EventManager* EventManager::_instance = 0;

// Private constructor.
EventManager::EventManager(void)
{
    // Intentionally left blank.
}

// The global point of access to the EventManager.
EventManager* EventManager::getInstance(void) {

  // Determine if an instance of the EventManager has been created.
  if( _instance == 0 ) {

    // Create the one and only instance.
    _instance = new EventManager;
  }
  return _instance;
}

// The method that logs events.
void EventManager::logEvent(string eventDescription) {
  // Code to log event.
  cout<<eventDescription<<endl;
}
```



Extra: Example A drawing application needs to create many copies of two types of shapes: **Circle** and **Square**.
Each shape has one attribute (Circle → `radius`, Square → `side`).
The system should allow creating new shapes **by copying existing ones** instead of creating them from scratch every time.

Tasks:
1.Draw a class diagram that solves this problem.
2.Write Java code that:

1. Defines a common interface for all shapes.
2. Allows each shape to make a copy of itself.
3. Demonstrates copying a Circle and a Square in the `main` method.

(Hint for students: Think of a design pattern where objects can create duplicates of themselves.)

How Students Should See It

Step 1 → Prototype interface with clone().
Step 2 → Concrete Prototype A (Circle).
Step 3 → Concrete Prototype B (Square).
Step 4 → Client (PrototypeDemo) that uses cloning instead of new.

Step 1: Define the common interface

java

```java
// Step 1: Interface for all shapes
interface Shape {
    Shape clone();   // method to duplicate the object
}
```

Step 2: Create the first concrete class (Circle)

java

```java
// Step 2: Concrete class - Circle
class Circle implements Shape {
    int radius;

    Circle(int r) { this.radius = r; }

    @Override
    public Shape clone() {
        return new Circle(this.radius);  // make a copy of Circle
    }

    public void show() {
        System.out.println("Circle with radius " + radius);
    }
}
```

Step 3: Create the second concrete class (Square)

```java
// Step 3: Concrete class - Square
class Square implements Shape {
    int side;

    Square(int s) { this.side = s; }

    @Override
    public Shape clone() {
        return new Square(this.side);  // make a copy of Square
    }

    public void show() {
        System.out.println("Square with side " + side);
    }
}
```

Step 4: Write the client (main method)

```java
// Step 4: Client code to test cloning
public class PrototypeDemo {
    public static void main(String[] args) {
        Shape c1 = new Circle(10);    // create original Circle
        Shape c2 = c1.clone();        // clone it

        Shape s1 = new Square(5);     // create original Square
        Shape s2 = s1.clone();        // clone it

        ((Circle)c1).show();
        ((Circle)c2).show();
        ((Square)s1).show();
        ((Square)s2).show();
    }
}
```

📜 Expected Output:

```pgsql
Circle with radius: 10
Circle with radius: 10
Square with side: 5
Square with side: 5
```

🎯 Key Idea:

- Clone creates **identical copy**
- So:
  - `c1` = `c2`
  - `s1` = `s2`

Summary

- In this session, we continued the discussion on **creational** design patterns, including:
  - ✓ Builder
  - ✓ Singleton

- Next … we will present **structural** and **behavioral** design patterns in detailed design.