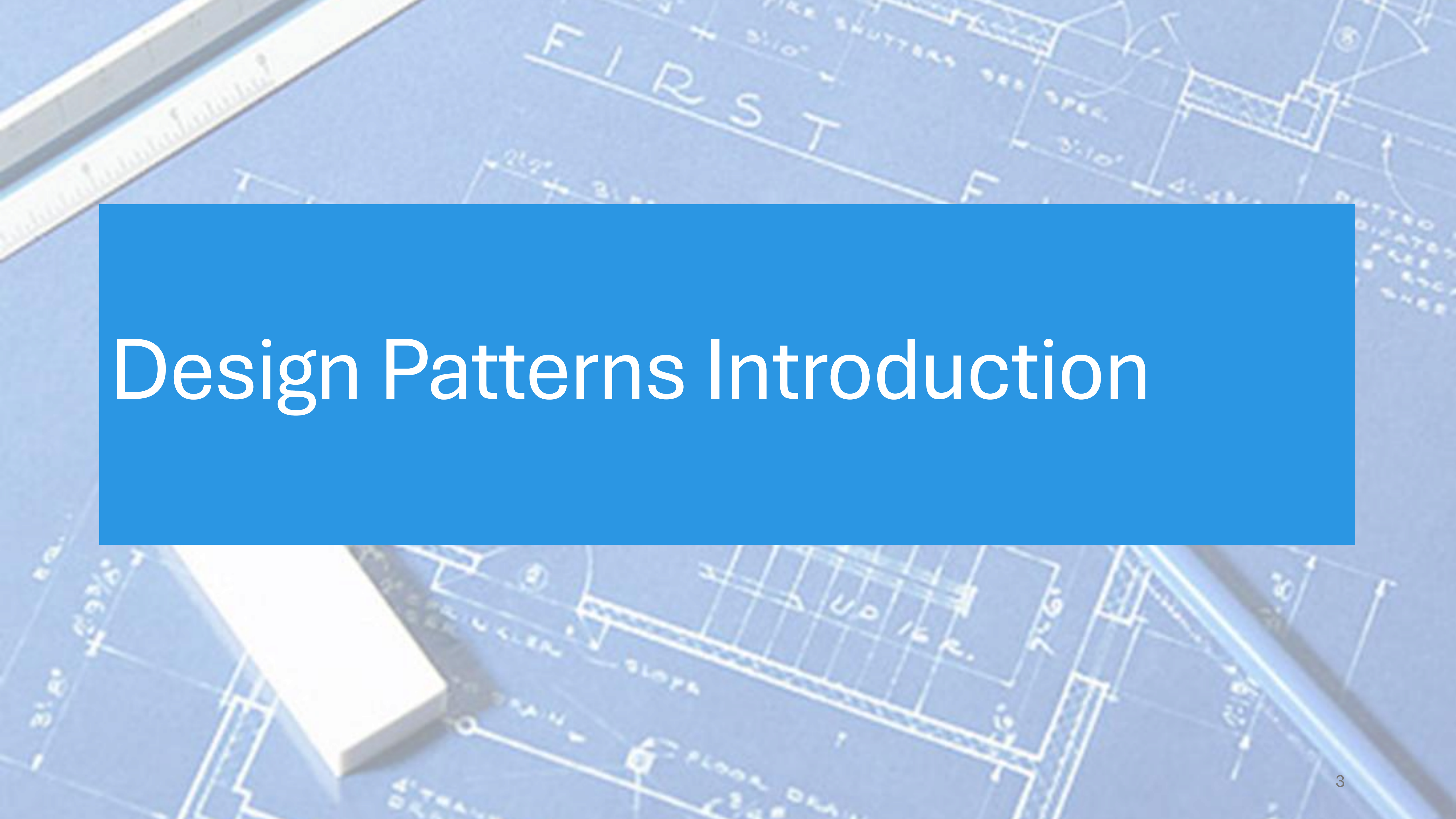
Software Design & Architecture
Design Patterns/
Creational Design Patterns

Pengyu Nie

Acknowledgements: slides adapted from previous versions by Mei Nagappan and Shane McIntosh, which are adapted from previous versions by Zhen Ming Jiang, Ahmed E. Hassan, Reid Holmes.

Agenda

- Design patterns introduction, benefits, category
- Creational design patterns
  - Singleton
  - Factory Method
  - Abstract Factory



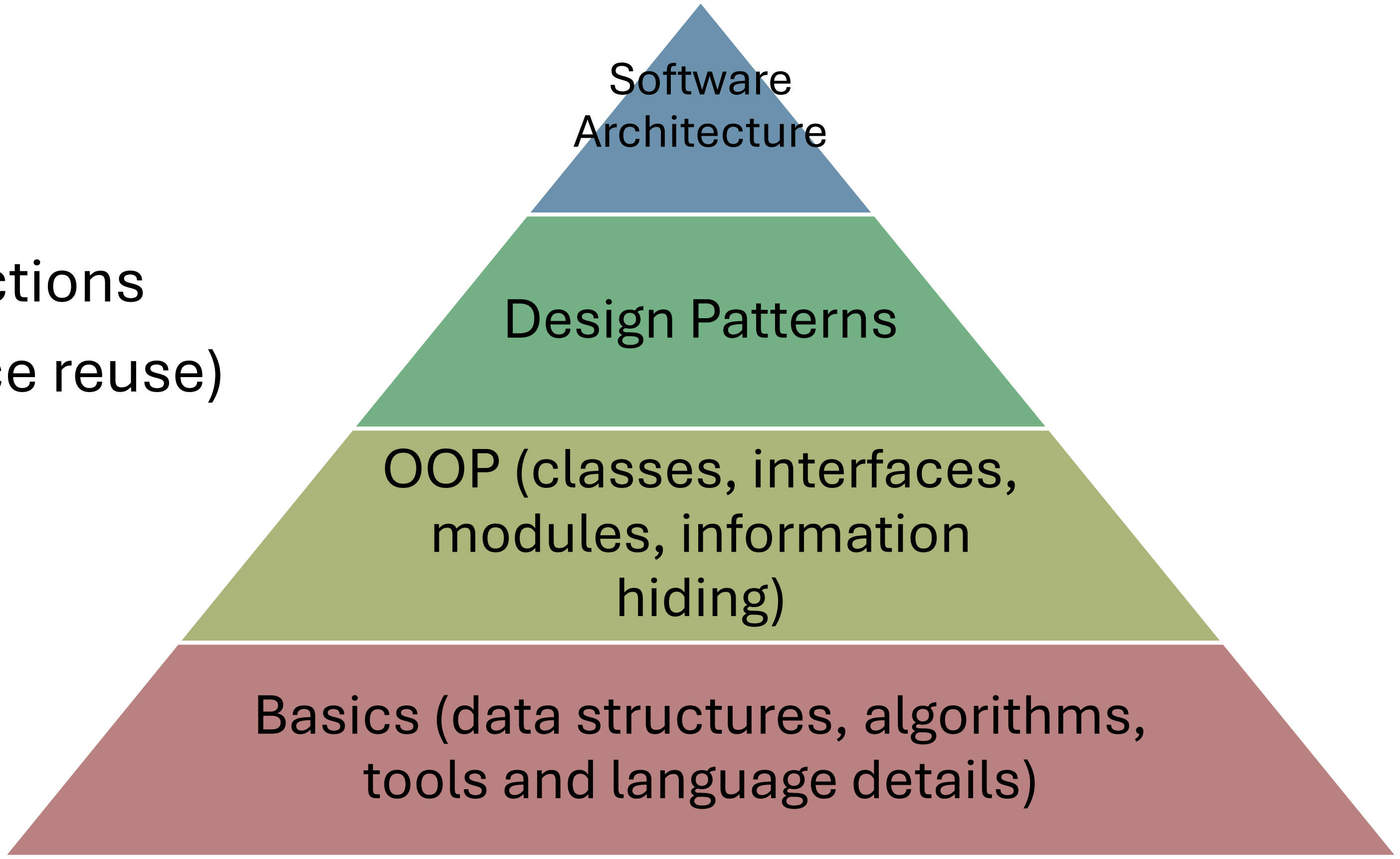
Design Patterns Introduction

Design Patterns Introduction

- Reusable solutions to common problems in object-oriented programming
  - A design pattern typically involves a small set of classes co-operating to achieve a desired end
  - This is done via adding a level of indirection in some clever way, and
  - The new improved solution provides the small functionality as an existing approach, but in the some more desirable way (elegance, efficiency, adaptability)

Design Patterns Introduction (cont.)

- Think of design patterns as…
  - high-level programming abstractions
  - a form of code reuse (experience reuse)


Software Architecture
Design Patterns
OOP (classes, interfaces, modules, information hiding)
Basics (data structures, algorithms, tools and language details)

Design Patterns Benefits (1)

- Leveraging existing design knowledge: other people have faced similar situations

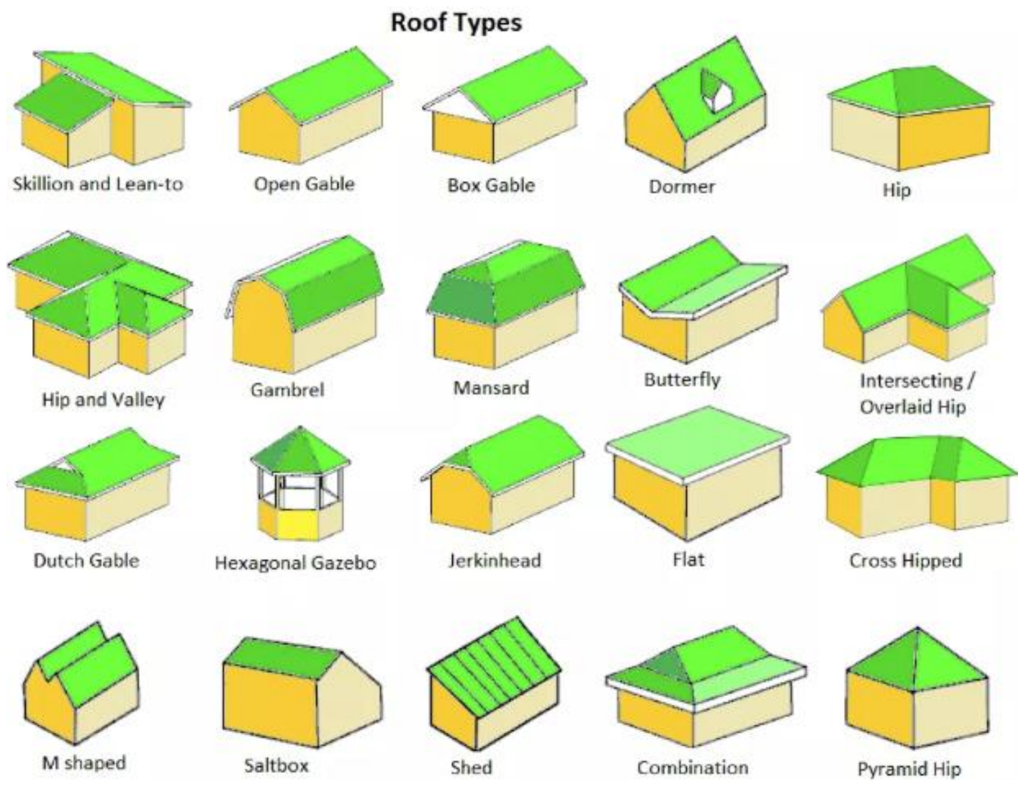
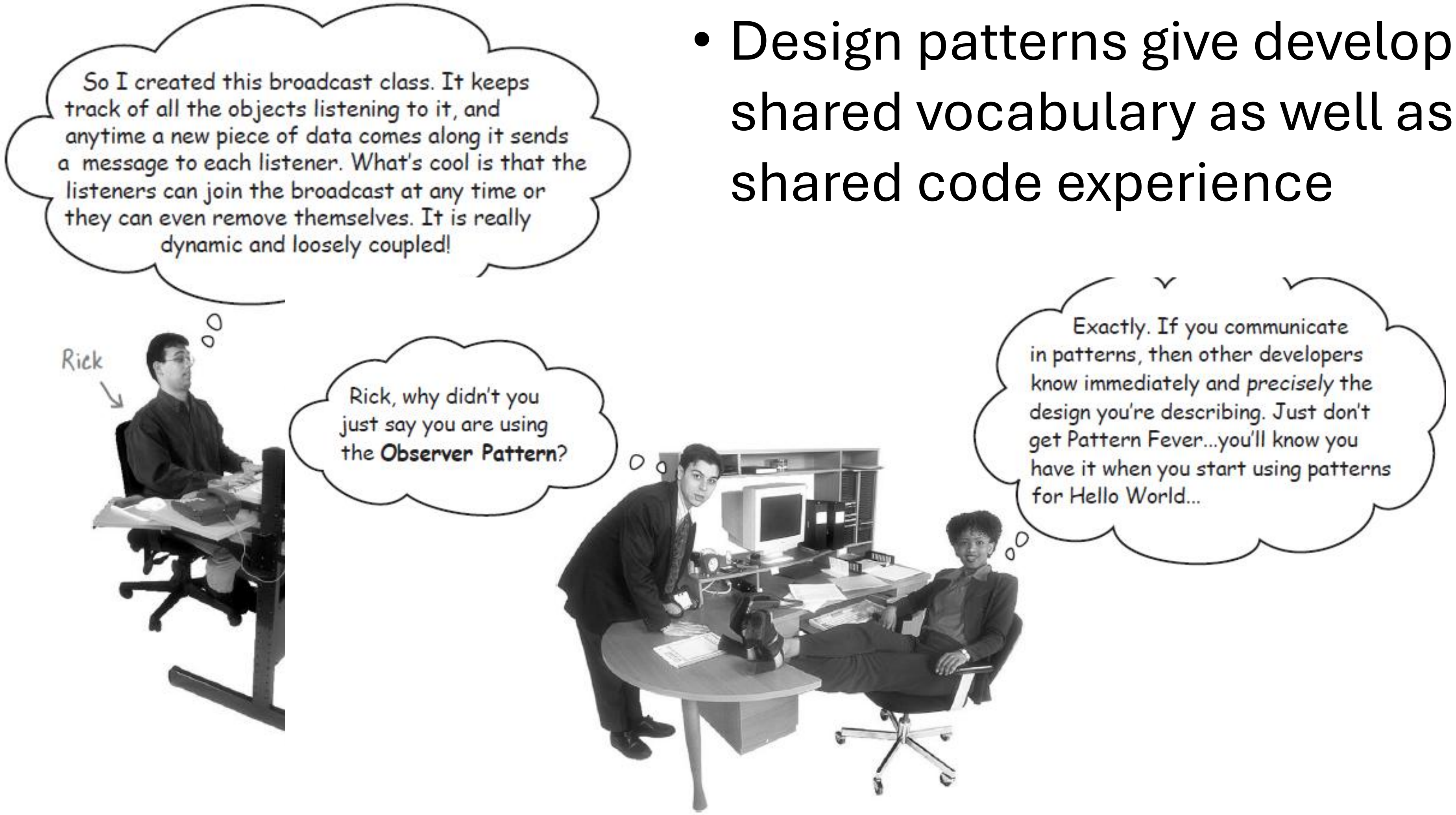


Image source: https://i1.wp.com/www.roofcalc.net/wp-content/uploads/2014/06/Roof-Types-Diagram.png

Design Pattern Benefits (2)

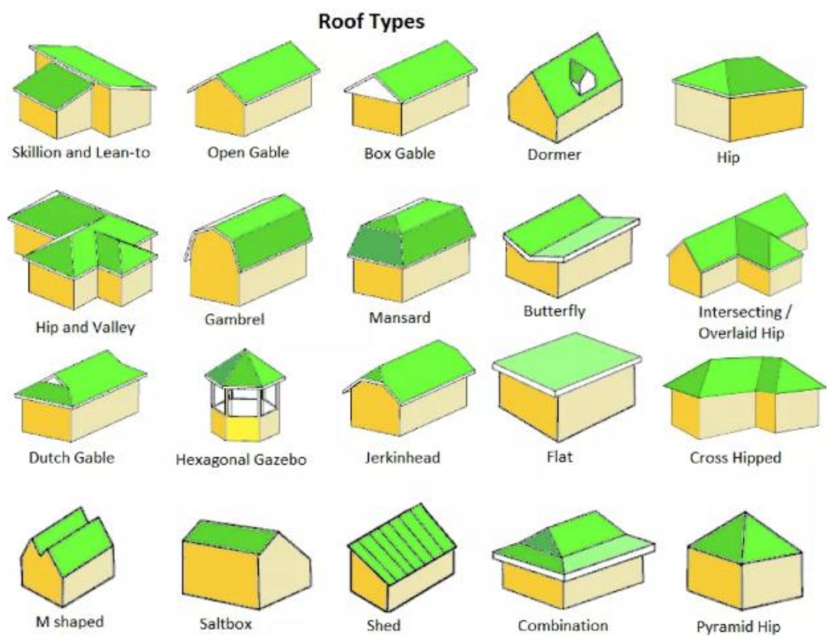

So I created this broadcast class. It keeps track of all the objects listening to it, and anytime a new piece of data comes along it sends a message to each listener. What's cool is that the listeners can join the broadcast at any time or they can even remove themselves. It is really dynamic and loosely coupled!
Rick
Rick, why didn't you just say you are using the **Observer Pattern**?
Exactly. If you communicate in patterns, then other developers know immediately and *precisely* the design you're describing. Just don't get Pattern Fever...you'll know you have it when you start using patterns for Hello World...

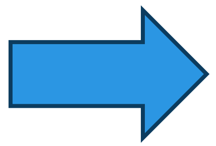
- Design patterns give developers a shared vocabulary as well as a shared code experience

Image source: Eric Freeman and Elisabeth Robson. Head First Design Patterns.

Design Patterns Benefits (3)

- Enhancingg flexibility for change: when maintainer looks at the code and design patterns choices, they know what changes they can make without breaking the design





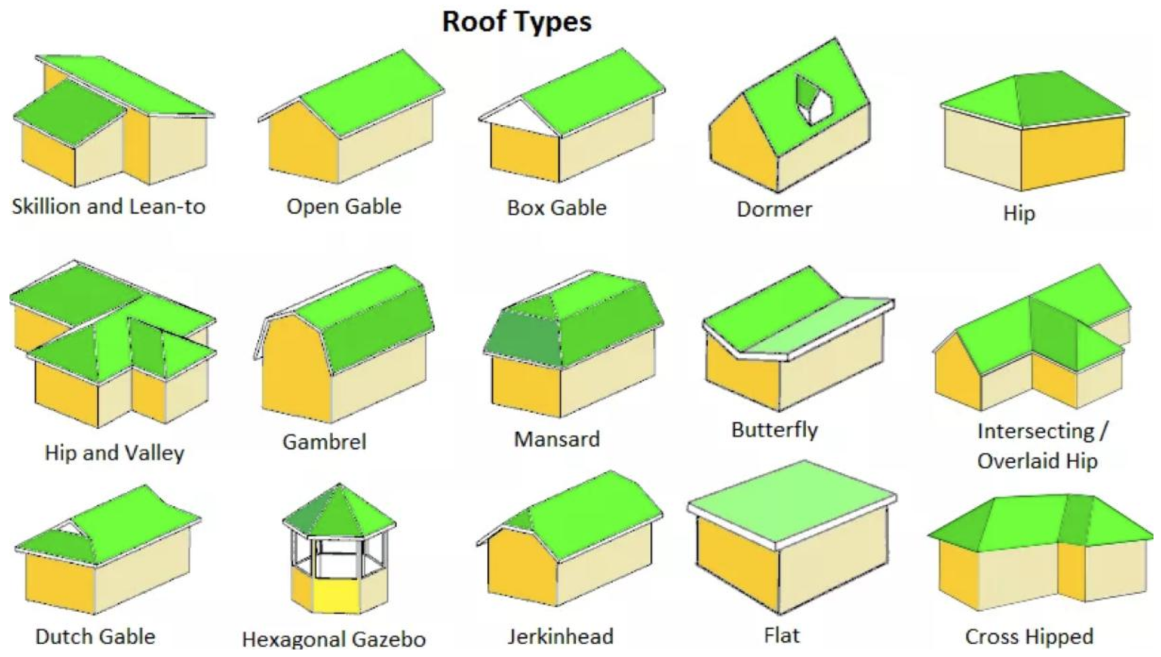
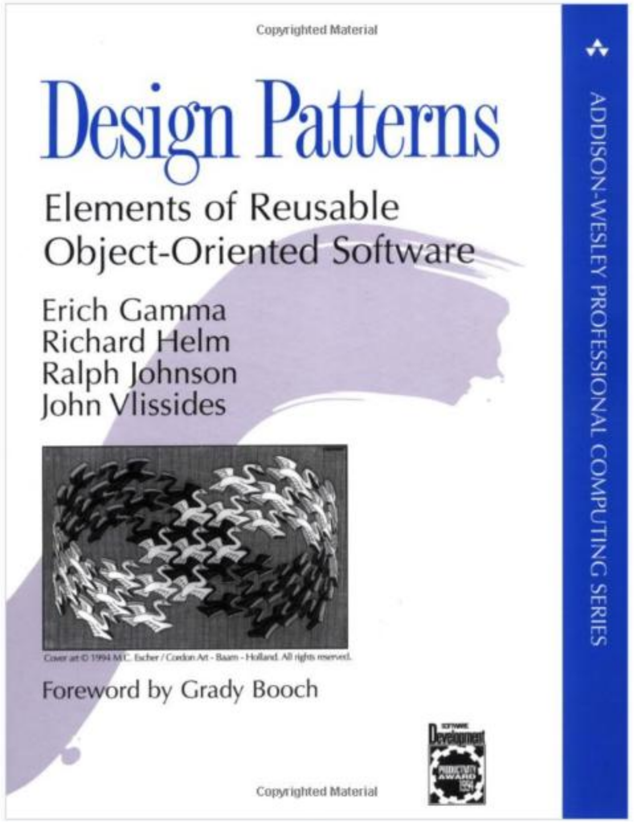


Image source: https://i1.wp.com/www.roofcalc.net/wp-content/uploads/2014/06/Roof-Types-Diagram.png

Design Patterns Benefits (4)

- Design new systems using higher-level abstractions than variables, procedures, and classes
- Understand relative tradeoffs, appropriateness, (dis)advantages of patterns
- Communicate about systems with other developers
- Give guidance in resolving non-functional requirements and trade-offs
- Avoid known traps, pitfalls, and temptations
- Ease restructuring, refactoring
- Foster coherent, directed system evolution and maintenance

Design Patterns Resources



Copyrighted Material

Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

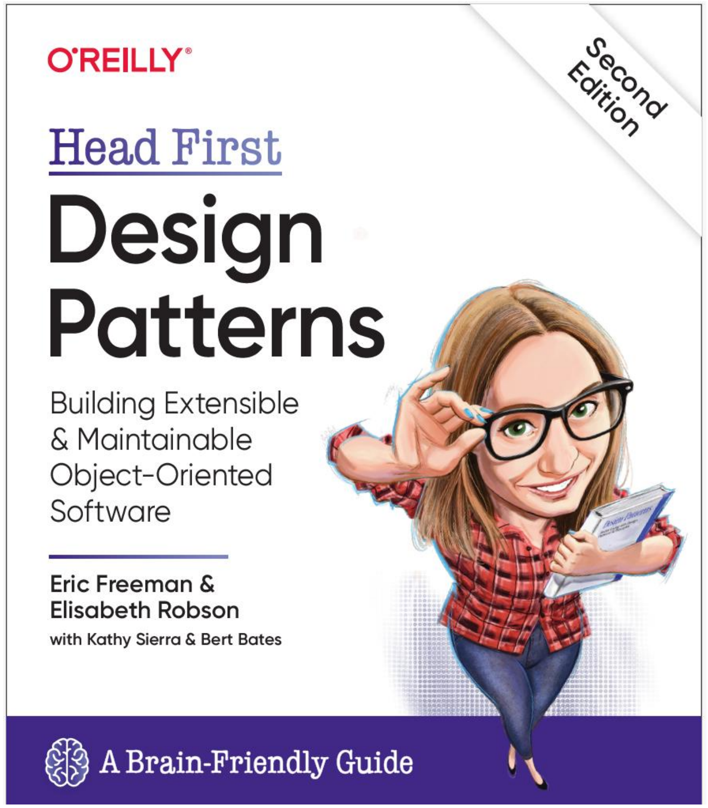
Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.

Foreword by Grady Booch

Copyrighted Material

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

"Gang of Four" Design Patterns



O'REILLY®

Second Edition

Head First

Design Patterns

Building Extensible & Maintainable Object-Oriented Software

Eric Freeman & Elisabeth Robson
with Kathy Sierra & Bert Bates

A Brain-Friendly Guide

Head First Design Patterns

Online resources

- https://refactoring.guru/design-patterns
- https://www.geeksforgeeks.org/software-design-patterns/
- https://hillside.net/patterns/
- etc.

Design Patterns Categories

- Creational: concern the process of object creation
  - Singleton, Factory Method, Abstract Factory, *today*
    Builder, Prototype, Object Pool
- Structural: concern the process of assembling objects and classes
  - Adapter, Composite, Decorator, *design patterns 2*
    Façade, Bridge, Flyweight, Proxy
- Behavioral: concern the interaction between classes or objects
  - Observer, Strategy, Template Method, *design patterns 3*
    Iterator, State, Chain of Responsibility,
    Command, Mediator, Memento *design patterns 4 – your pick from the remaining ones*

Singleton

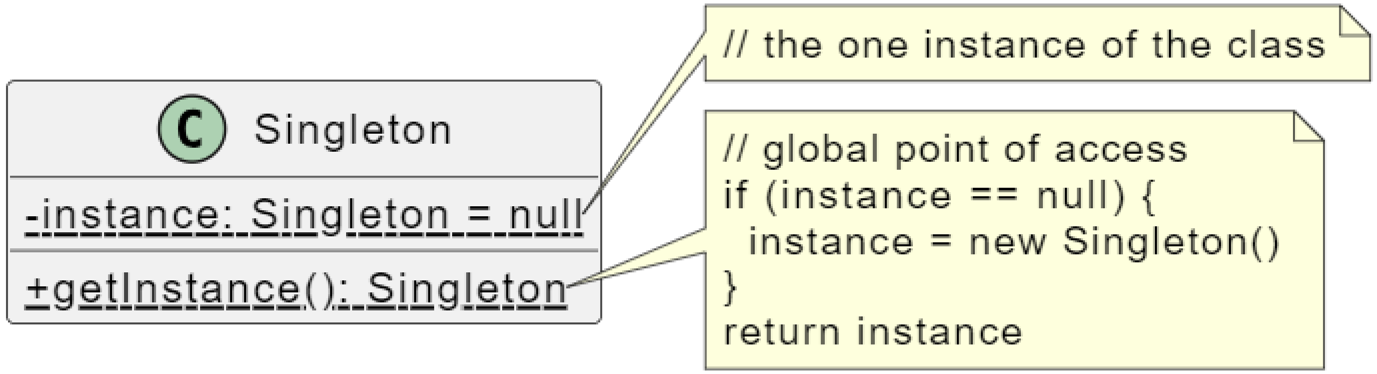
Singleton: Motivation and Intent

- Motivation: some classes must only have one instance (e.g., file system, database connection, window manager)
- Intent: ensure a class has only one instance; provide a global point of access

I tell ya she's ONE OF A KIND. Look at the lines, the curves, the body, the headlights!
You talkin' to me or the car? Oh, and when can I get my oven mitt back?

Image source: Eric Freeman and Elisabeth Robson. Head First Design Patterns.

Singleton: Solution


Singleton
-instance: Singleton = null
+getInstance(): Singleton
// the one instance of the class
// global point of access
if (instance == null) {
  instance = new Singleton()
}
return instance

no public constructor!
instantiation happens only once

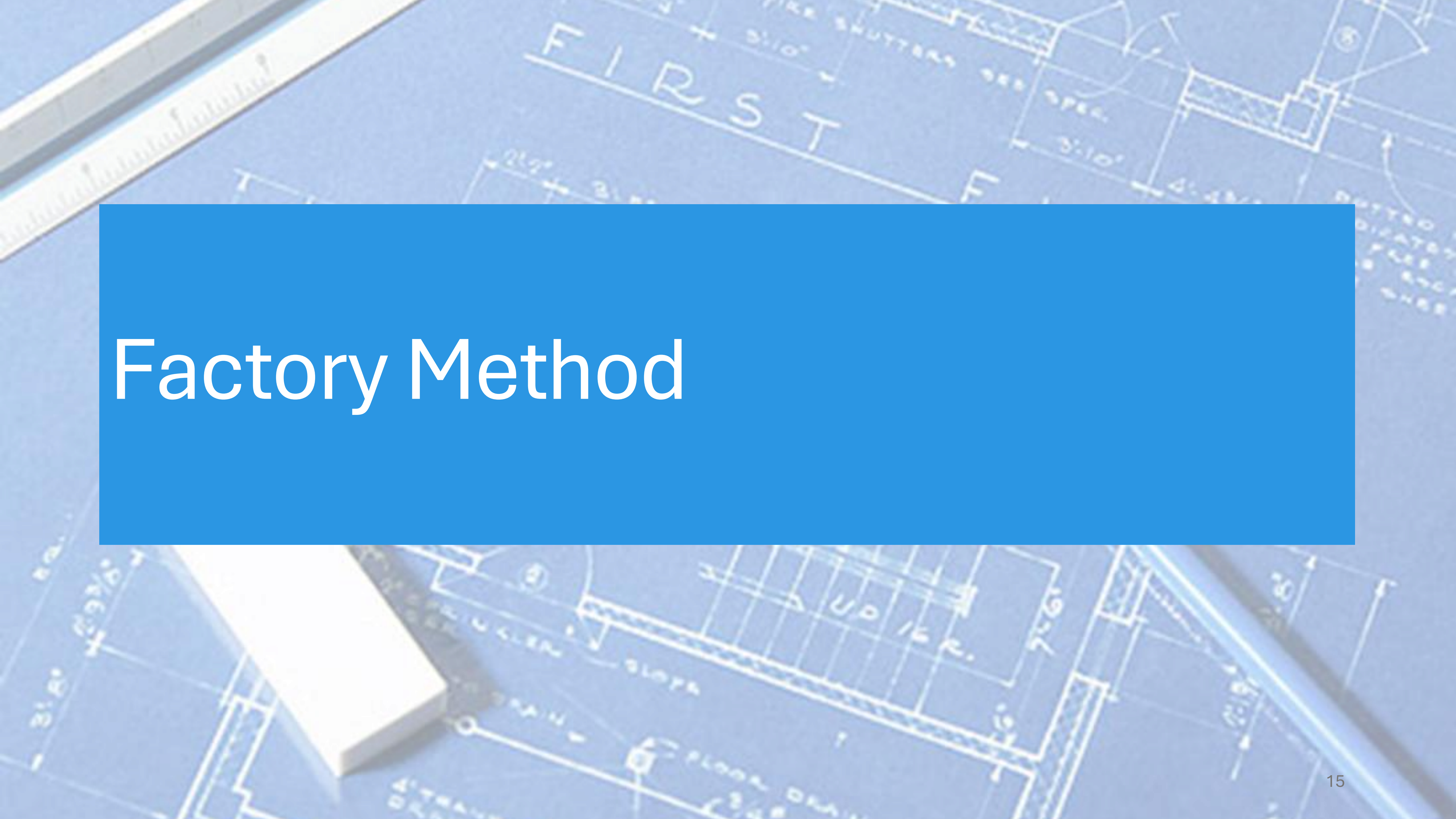
Kotlin has built-in support for Singleton with object keyword

```
object Singleton {
  // … (other fields or methods)
}
```

… or the more traditional way

```
class Singleton private constructor() {
  companion object {
    private var instance: Singleton? = null
    fun getInstance(): Singleton {
      if (instance == null) { instance == Singleton() }
      return instance!!
    }
  }
  // … (other fields or methods)
}
```

Demo: https://github.com/pengyunie/CS446Demo1251/tree/main/app/src/main/java/ca/uwaterloo/cs446/dp/singleton



Factory Method

Factory Method: Motivation and Intent

- Motivation:
  - we want to create an object of (a subclass of) an abstract class
  - we don't care which subclass is used
- Intent:
  - define an interface for creating objects in the superclass
  - but let subclasses alter the type of objects that will be created

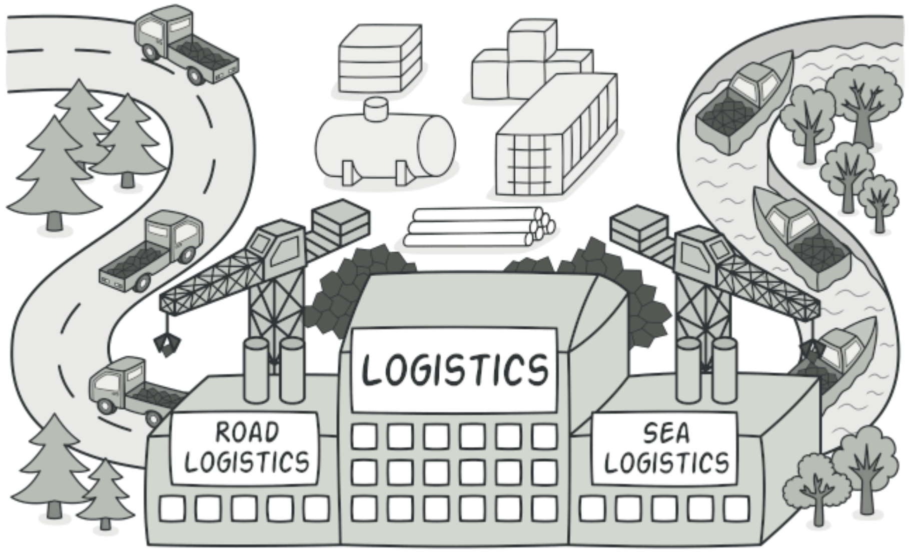
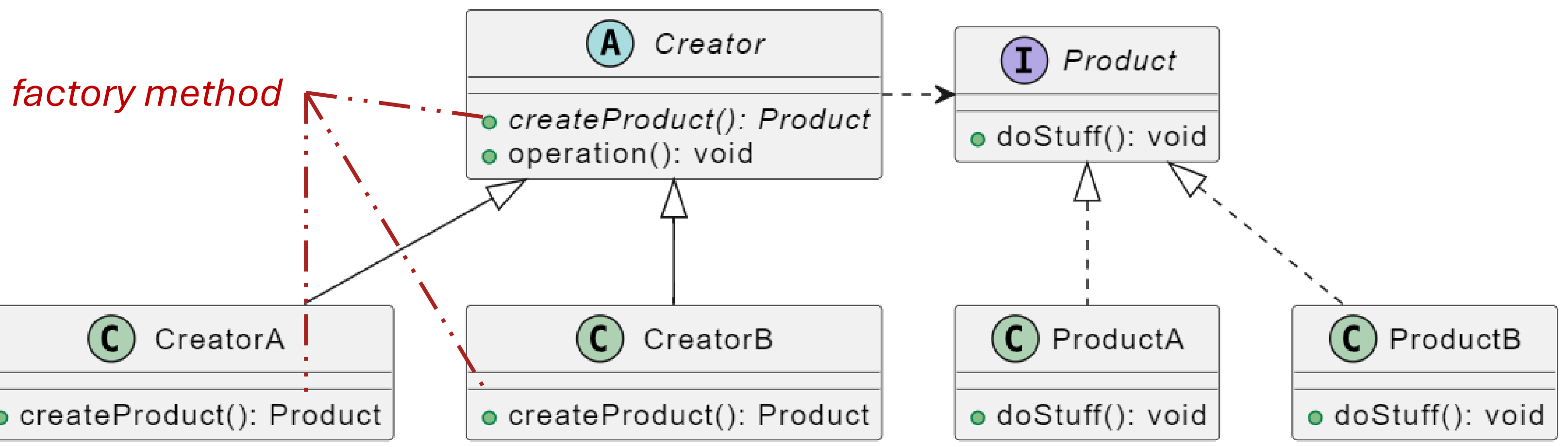


Image source: https://refactoring.guru/design-patterns/factory-method

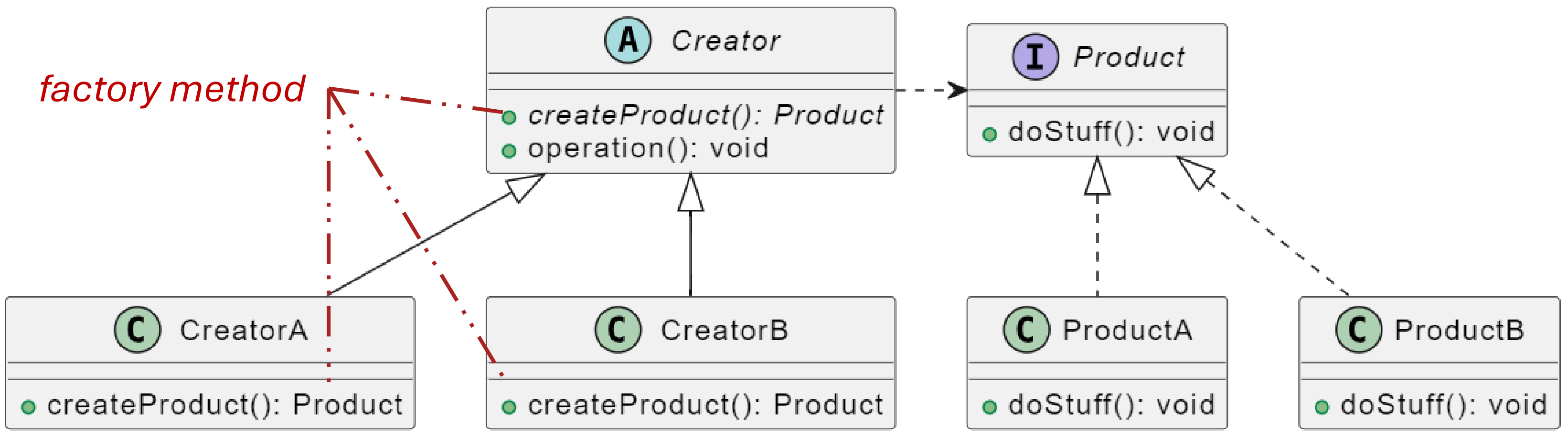
Factory Method: Solution



✓ Single responsibility principle
- (abstract) Creator: define the common operation steps
- (concrete) CreatorA/B: define which product being used
- (abstract) Product: declare common interface
- (concrete) ProductA/B: implement each operation

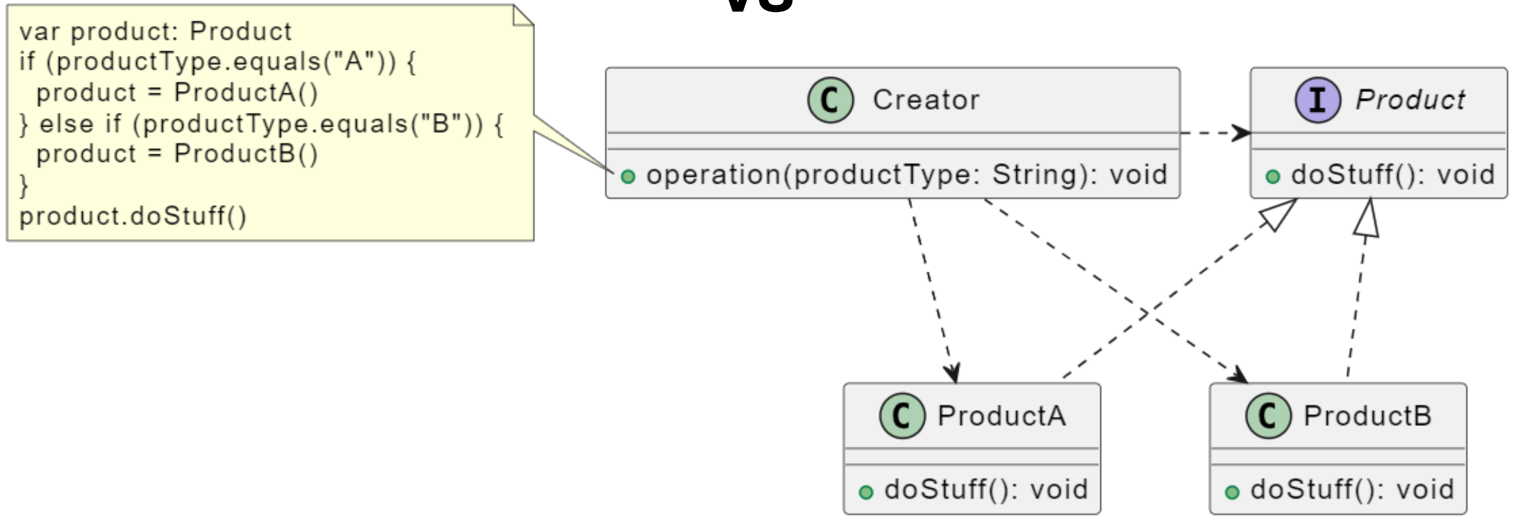
✓ Open-closed principle
- client can extend to CreatorC, ProductC, etc.

Factory Method: Solution (cont.)



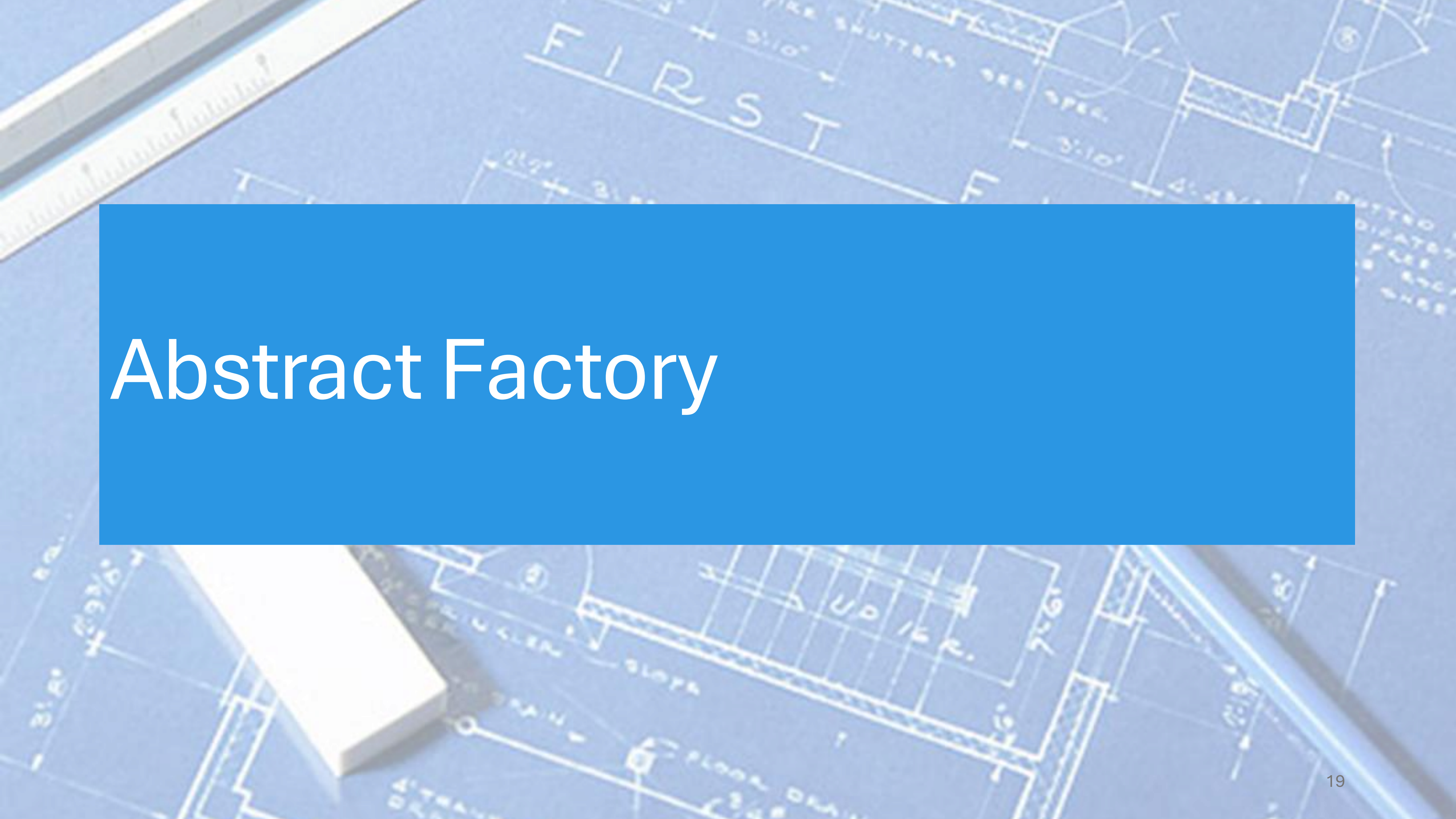
✓ Dependency inversion principle

VS



```
var product: Product
if (productType.equals("A")) {
  product = ProductA()
} else if (productType.equals("B")) {
  product = ProductB()
}
product.doStuff()
```

Demo: https://github.com/pengyunie/CS446Demo1251/tree/main/app/src/main/java/ca/uwaterloo/cs446/dp/factorymethod



Abstract Factory

Abstract Factory: Motivation and Intent

- Motivation:
  - we want to create objects of (some subclasses of) several abstract classes (e.g., following a theme)
  - we don't care which subclasses are used
- Intent:
  - provide an interface for creating families of related/dependent objects without specifying their concrete classes

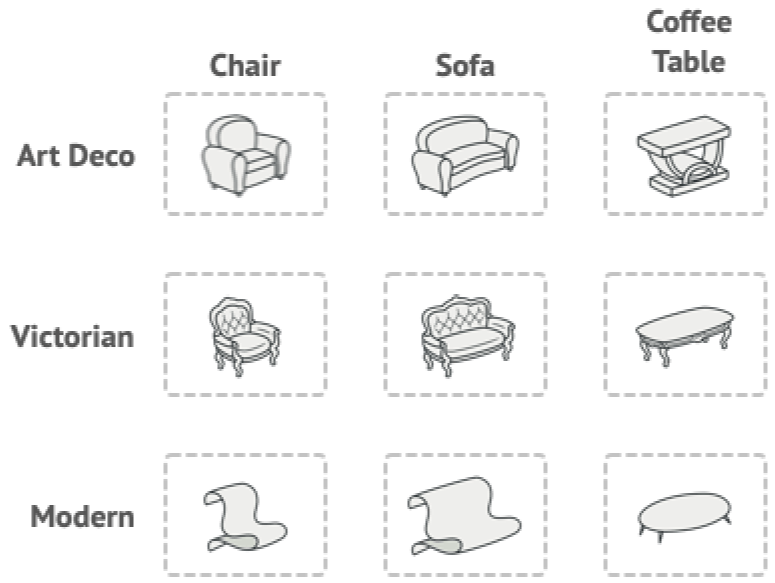
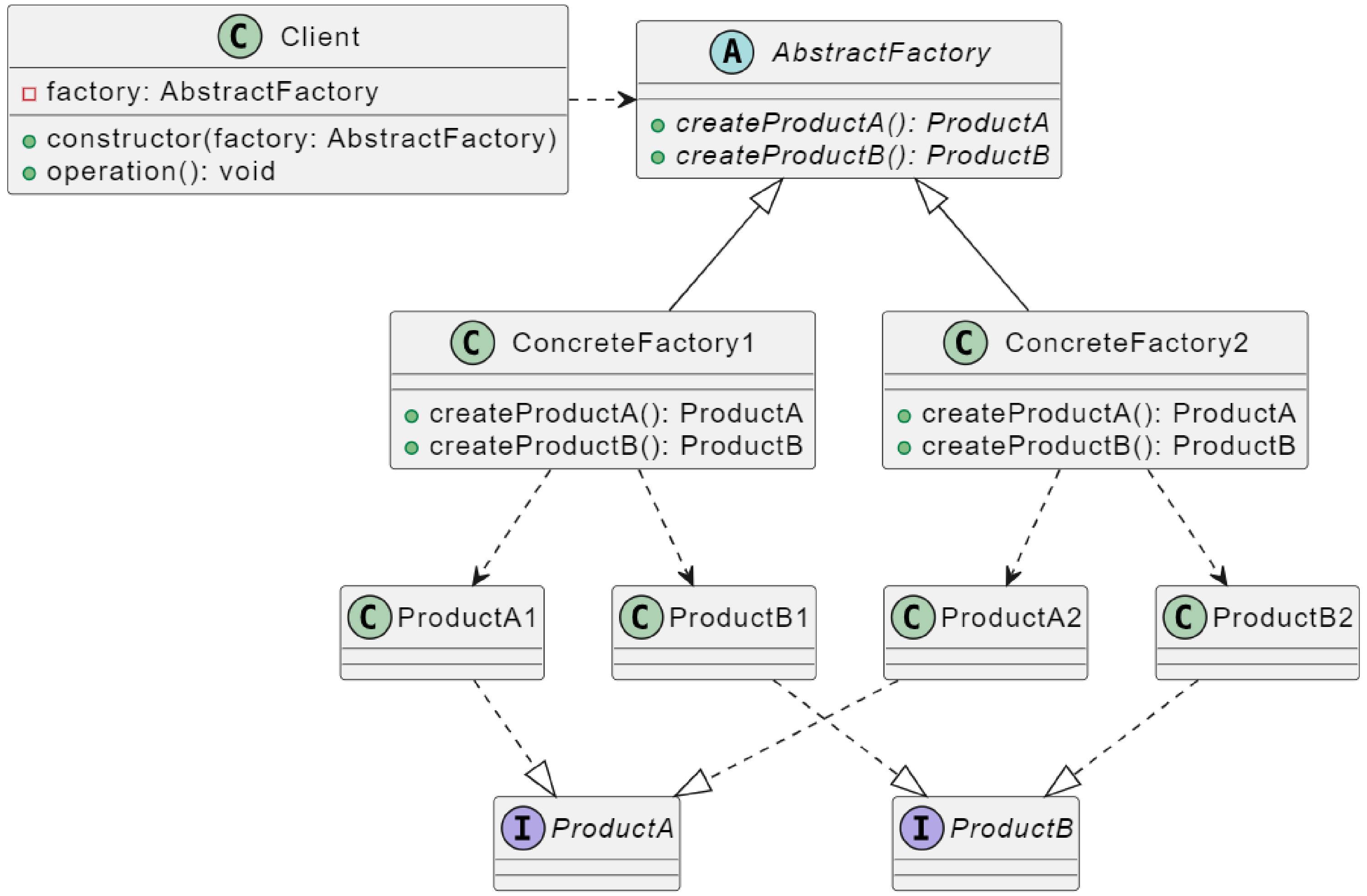


Image source: https://refactoring.guru/design-patterns/abstract-factory

Abstract Factory: Solution



C Client
□ factory: AbstractFactory
● constructor(factory: AbstractFactory)
● operation(): void
A *AbstractFactory*
● *createProductA(): ProductA*
● *createProductB(): ProductB*
C ConcreteFactory1
● createProductA(): ProductA
● createProductB(): ProductB
C ConcreteFactory2
● createProductA(): ProductA
● createProductB(): ProductB
C ProductA1
C ProductB1
C ProductA2
C ProductB2
I *ProductA*
I *ProductB*

scaling up the *factory method* design pattern

Demo: https://github.com/pengyunie/CS446Demo1251/tree/main/app/src/main/java/ca/uwaterloo/cs446/dp/abstractfactory

Agenda (recap)

- Design patterns introduction, benefits, category
- Creational design patterns
  - Singleton
  - Factory Method
  - Abstract Factory
- Review P4: Iteration 2 Demo requirements