เรื่องนี้ห้ามไปหาดูเอง อ่านจากลิ้งค์เค้าเท่านั้น

Software Design Pattern

Parinya Ekparinya

Parinya.Ek@kmitl.ac.th

Software Architecture and Design

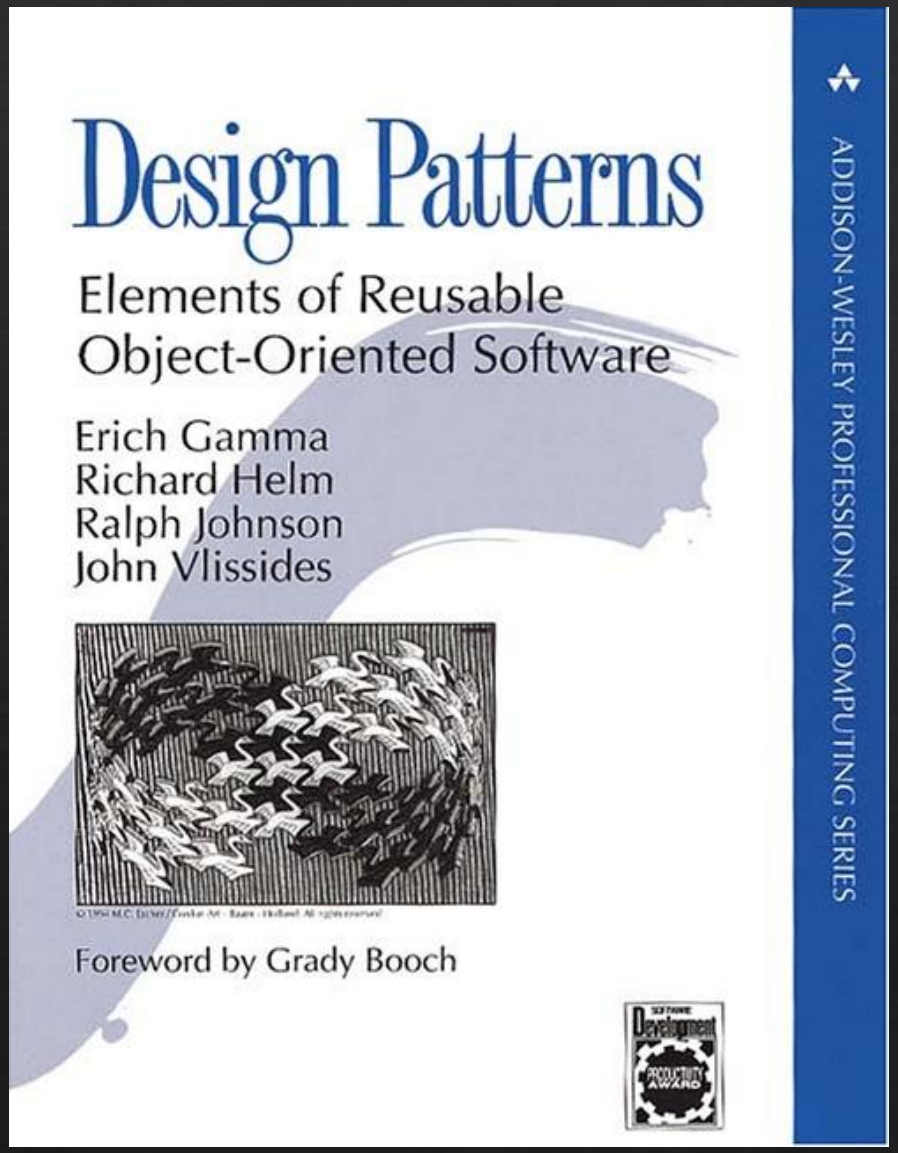
Acknowledgement

The content of the following slides are partially based on the listed material as follows:

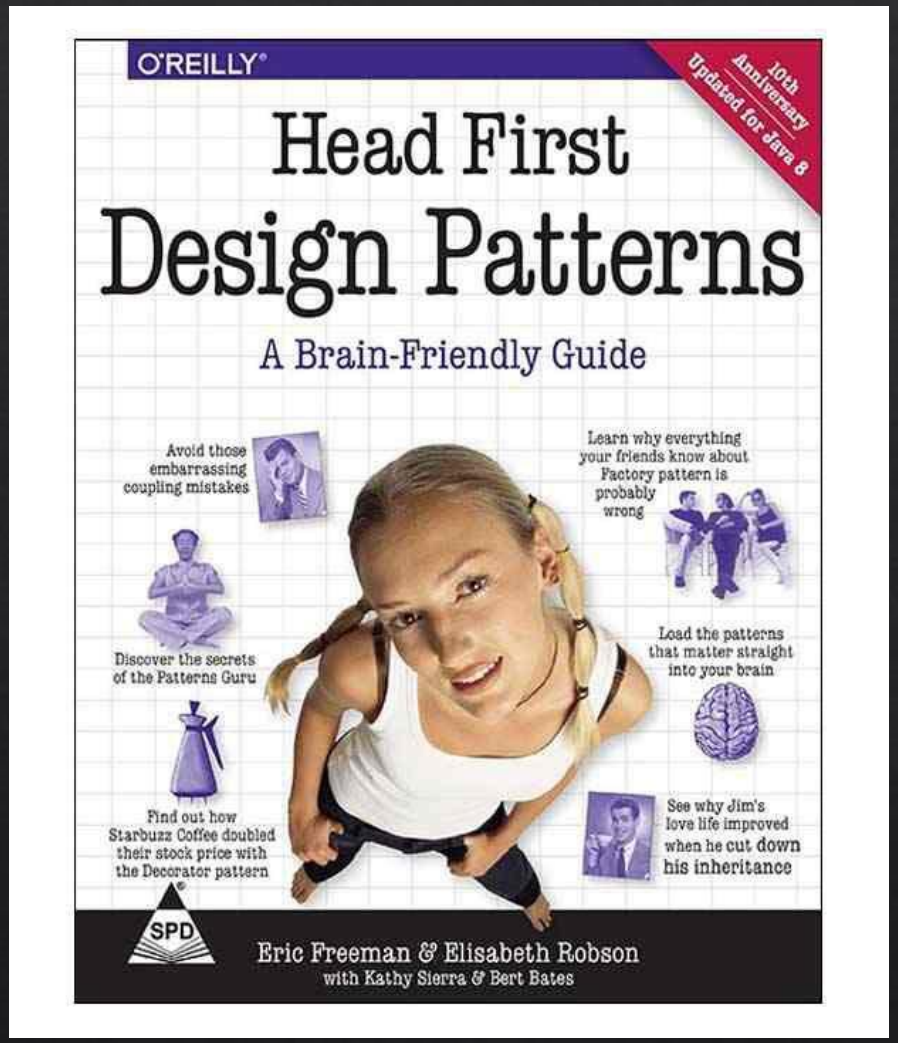
- Object-Oriented Patterns & Frameworks by Dr. Douglas C. Schmidt
- https://en.wikipedia.org/wiki/Factory_method_pattern
- https://en.wikipedia.org/wiki/Abstract_factory_pattern
- https://en.wikipedia.org/wiki/Builder_pattern
- https://en.wikipedia.org/wiki/Prototype_pattern
- https://en.wikipedia.org/wiki/Singleton_pattern
- https://en.wikipedia.org/wiki/Dependency_injection

(Software) Design Pattern

- A software design pattern is a general, reusable solution to a commonly occurring problem in the context of modular software design.
- In 1994, the concept of software design patterns for object-oriented software was published in ***"Design Patterns: Elements of Reusable Object-Oriented Software"*** by Erich Gamma, John Vlissides, Ralph Johnson, and Richard Helm.
- People often call these four authors "the Gang of Four".
- So, their book is often called "the GoF book".
- Originally, GoF design patterns covers 23 classic design patterns.
- After that people keep introducing more design patterns.



The GoF Book (Smalltalk)



Another useful book (Java)

2nd Edition
December 2020

Relationship with Other Design Concepts

Architectural Styles

Reference Architectures / Domain Specific Architectures

Design Patterns

Design/Coding Idioms

Benefits & Limitations of Design Patterns

Benefits

- Design reuse
- Uniform design vocabulary
- Enhance understanding, restructuring, & team communication
- Basis for automation
- Transcends language-centric biases/myopia
- Abstracts away from many unimportant details

Limitations

- Require significant tedious & error-prone human effort to handcraft pattern implementations design reuse
- Can be deceptively simple uniform design vocabulary
- May limit design options
- Leaves important (implementation) details unresolved

WARNING:

Overuse of design patterns can lead to code that is downright over-engineered. Always go with the simplest solution that does the job and introduce patterns only where the need emerges.

Freeman, E., Robson, E., Bates, B., & Sierra, K. (2008). *Head first design patterns*. " O'Reilly Media, Inc."

Four Basic Parts of a Pattern

1. Name
2. Problem (including "applicability")
3. Solution (both visual & textual descriptions)
4. Consequences (the results and trade-offs of applying the pattern)

Classification of Design Patterns

- **Creational patterns** provide object creation mechanisms that increase flexibility and reuse of existing code.
- **Structural patterns** explain how to assemble objects and classes into larger structures, while keeping the structures flexible and efficient.
- **Behavioral patterns** take care of effective communication and the assignment of responsibilities between objects.

Creational Design Patterns

- Factory Method
- Abstract Factory
- Builder
- Prototype
- Singleton
- Dependency Injection

Factory Method

Factory Method: Problem

- How can an object be created so that subclasses can redefine which class to instantiate?
- How can a class defer instantiation to subclasses?

Applicability

- When a class cannot anticipate the objects it must create or a class wants its subclasses to specify the objects it creates

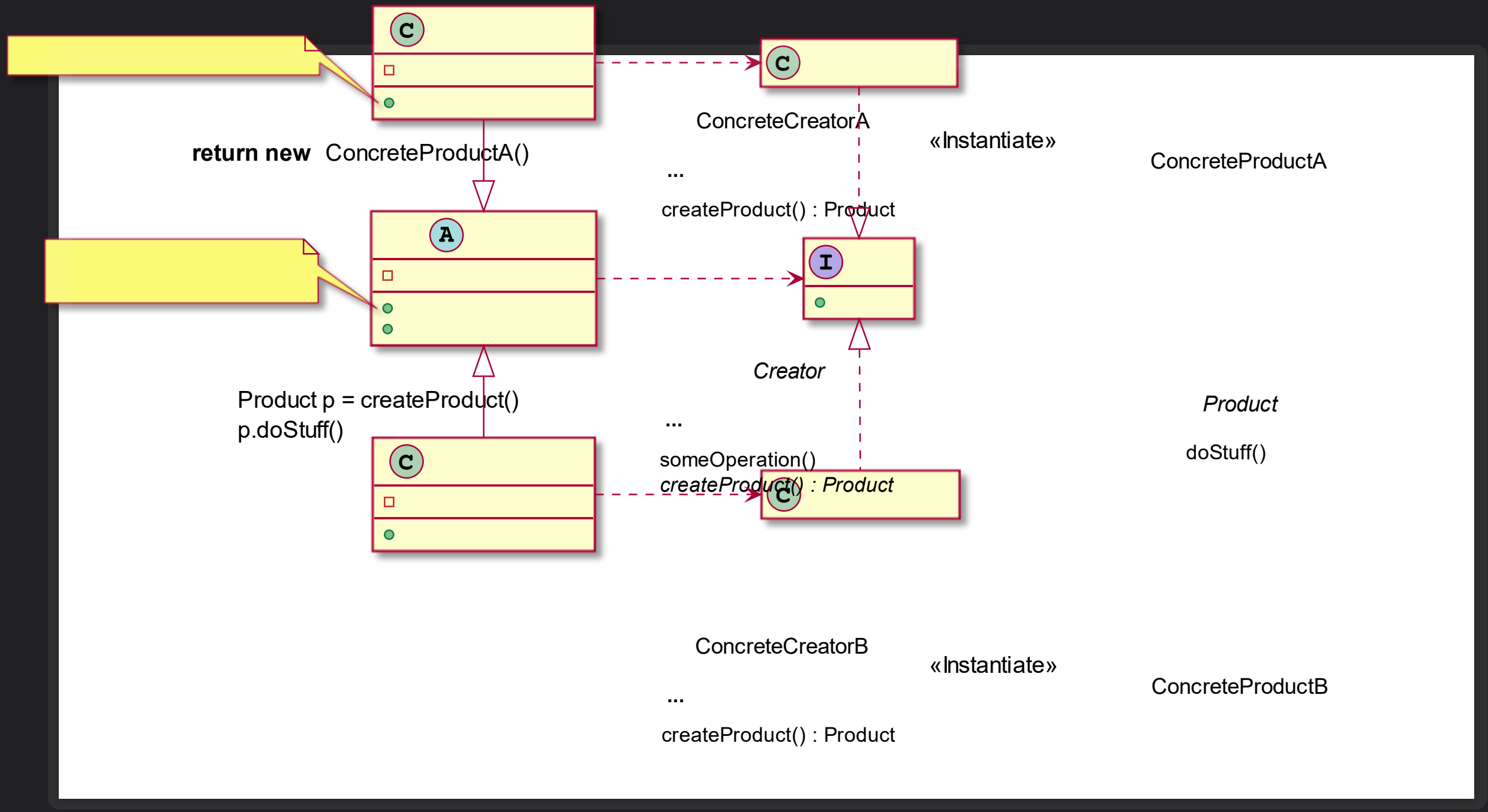
Factory Method: Solution

Intent

- Provide an interface for creating an object, but leave choice of object's concrete type to a subclass

The Factory Method design pattern describes how to solve such problems:

- Define a separate operation (factory method) for creating an object
- Create an object by calling a factory method


C
C
ConcreteCreatorA
«Instantiate»
ConcreteProductA
return new ConcreteProductA()
...
createProduct() : Product
A
I
Creator
Product p = createProduct()
p.doStuff()
...
someOperation()
createProduct() : Product
Product
doStuff()
C
C
ConcreteCreatorB
«Instantiate»
ConcreteProductB
...
createProduct() : Product

Factory Method: Consequences

+ By avoiding to specify the class name of the concrete class & the details of its creation, the client code has become more flexible isolates code for construction & representation.
+ The client is only dependent on the interface.
– Construction of objects requires one additional class in some cases.

Factory Method: Examples

- Factory Method (refactoring.guru)
- Factory method pattern – Wikipedia

Abstract Factory

Abstract Factory: Problem

- How can an application be independent of how its objects are created?
- How can a class be independent of how the objects it requires are created?
- How can families of related or dependent objects be created?

Applicability

- When clients cannot anticipate groups of classes to instantiate

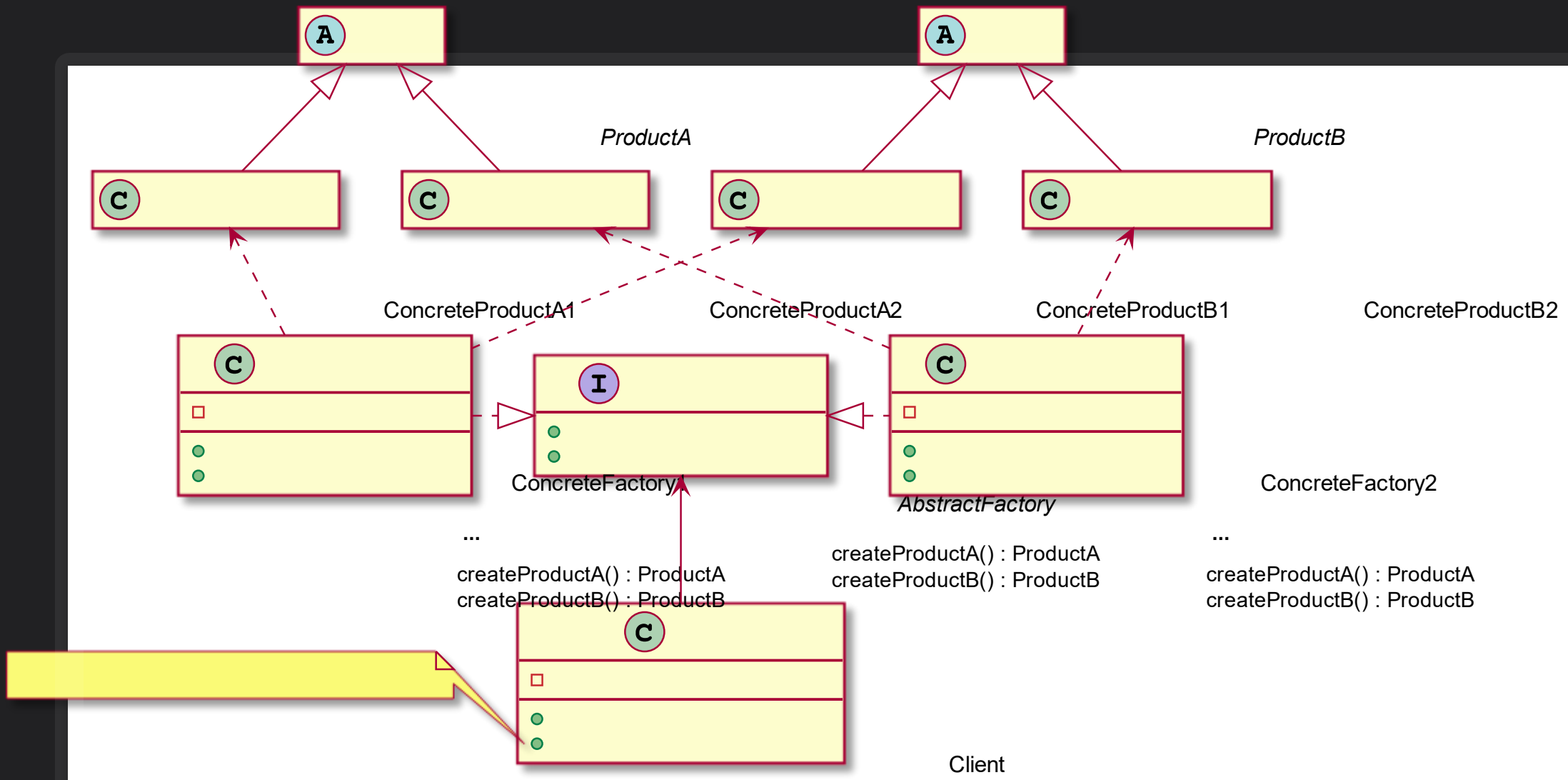
Abstract Factory: Solution

Intent

- Create families of related objects without specifying subclass names

The Abstract Factory design pattern describes how to solve such problems:

- Encapsulate object creation in a separate (factory) object. That is, define an interface (AbstractFactory) for creating objects, and implement the interface.
- A class delegates object creation to a factory object instead of creating objects directly.


A
A
ProductA
ProductB
C
C
C
C
ConcreteProductA1
ConcreteProductA2
ConcreteProductB1
ConcreteProductB2
C
I
C
ConcreteFactory1
AbstractFactory
ConcreteFactory2
...
createProductA() : ProductA
createProductB() : ProductB
...
createProductA() : ProductA
createProductB() : ProductB
...
createProductA() : ProductA
createProductB() : ProductB
C
Client
factory : AbstractFactory
Client(factory : AbstractFactory)
someOperation()
ProductA pa = factory.createProductA()

Abstract Factory: Consequences

+ Flexibility: removes type (i.e., subclass) dependencies from clients
+ Abstraction & semantic checking: hides product's composition
– Hard to extend factory interface to create new products

Abstract Factory: Examples

- Design Patterns: Abstract Factory in Java (refactoring.guru)
- Abstract Factory Design Pattern in Java – JournalDev

Builder

Builder: Problem

- How can a class (the same construction process) create different representations of a complex object?
- How can a class that includes creating a complex object be simplified?

Applicability

- Need to isolate knowledge of the creation of a complex object from its parts
- Need to allow different implementations/interfaces of an object's parts

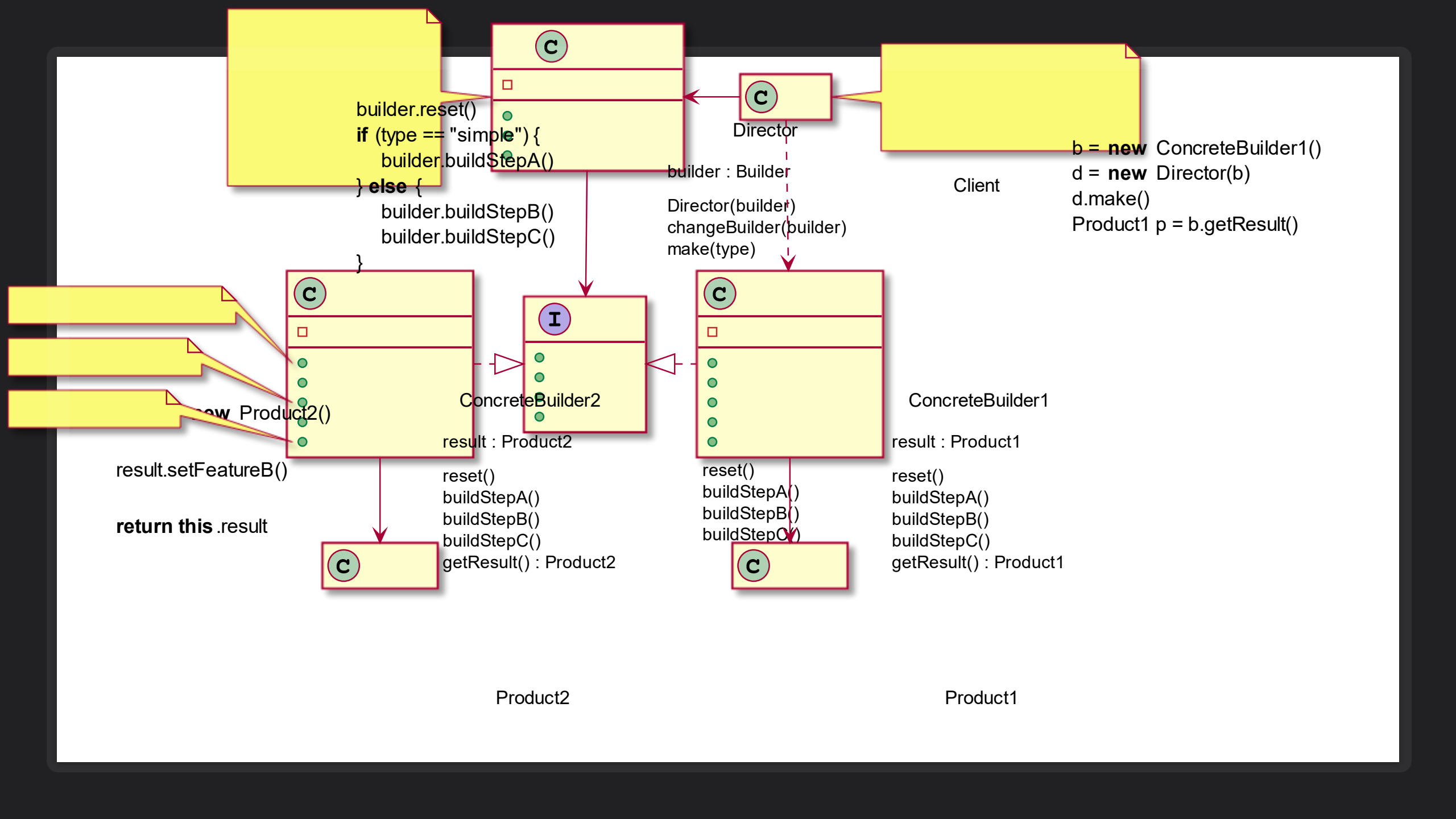
Builder: Solution

Intent

- Separate the construction of a complex object from its representation so that the same construction process can create different representations

The Builder design pattern describes how to solve such problems:

- Encapsulate creating and assembling the parts of a complex object in a separate Builder object
- A class delegates object creation to a Builder object instead of creating the objects directly.


builder.reset()
if (type == "simple") {
builder.buildStepA()
} else {
builder.buildStepB()
builder.buildStepC()
}
Director
builder : Builder
Director(builder)
changeBuilder(builder)
make(type)
Client
b = new ConcreteBuilder1()
d = new Director(b)
d.make()
Product1 p = b.getResult()
ew Product2()
result.setFeatureB()
return this .result
ConcreteBuilder2
result : Product2
reset()
buildStepA()
buildStepB()
buildStepC()
getResult() : Product2
ConcreteBuilder1
result : Product1
reset()
buildStepA()
buildStepB()
buildStepC()
getResult() : Product1
Product2
Product1

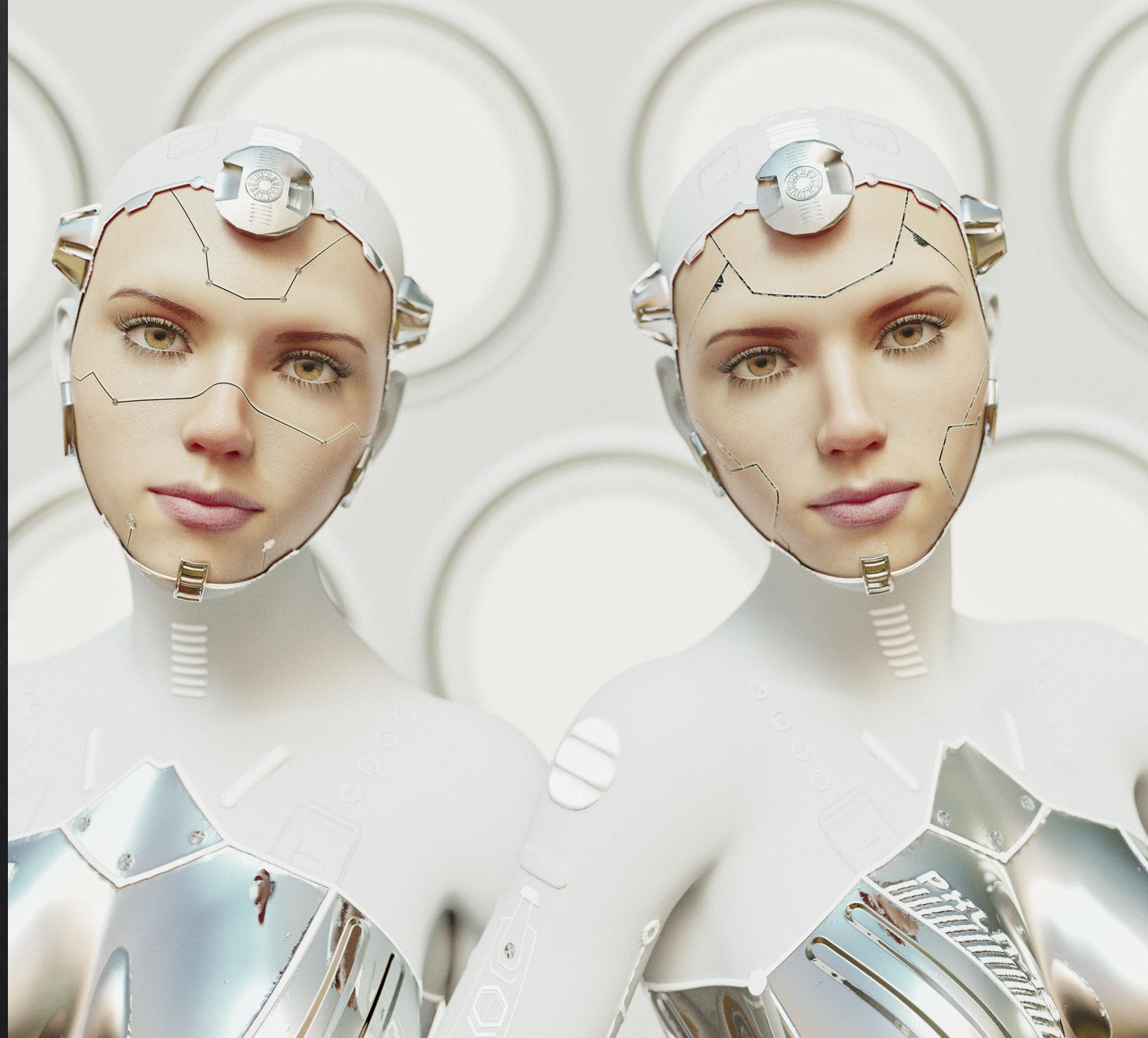
Builder: Consequences

+ Can vary a product's internal representation
+ Isolates code for construction & representation
+ Finer control over the construction process
– May involve a lot of classes

Builder: Examples

- Design Patterns: Builder in Java (refactoring.guru)
- Builder Design Pattern in Java - JournalDev
- Implementing the builder pattern in Java 8 - Tutorial (vogella.com)

Prototype



Prototype: Problem

- How can objects be created so that which objects to create can be specified at run-time?
- How can dynamically loaded classes be instantiated?

Applicability

- When a system should be independent of how its products are created, composed, & represented
- When the classes to instantiate are specified at run-time

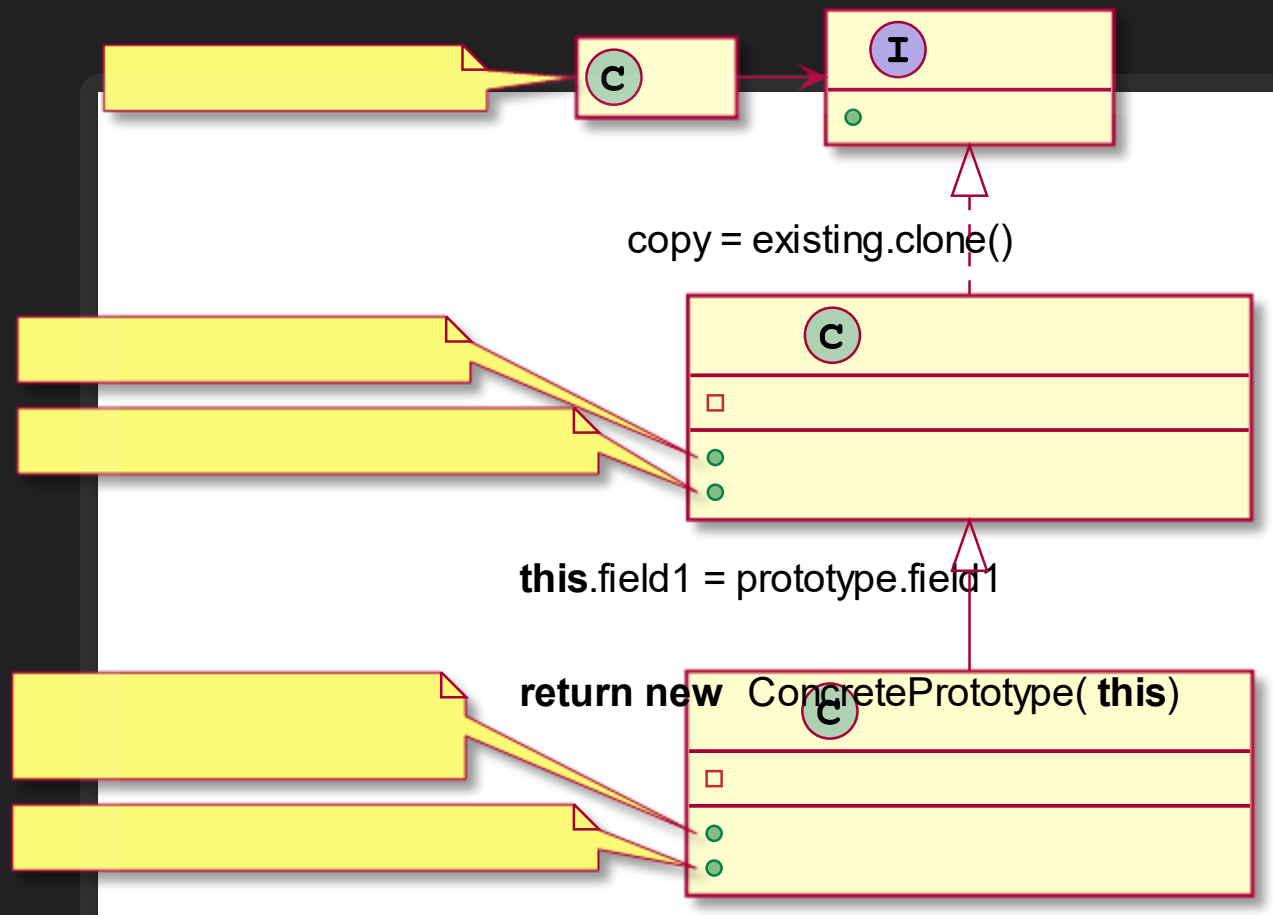
Prototype: Solution

Intent

- Specify the kinds of objects to create using a prototypical instance & create new objects by copying this prototype

The prototype design pattern describes how to solve such problems:

- Define a ***Prototype*** object that returns a copy of itself.
- Create new objects by copying a ***Prototype*** object.



Prototype

clone() : Prototype

Client

ConcretePrototype

field1

ConcretePrototype(prototype : Prototype)
clone() : Prototype

super(prototype)
this.field2 = prototype.field2

return new SubclassPrototype(**this**)

SubclassPrototype

field2

SubclassPrototype(prototype : Prototype)
clone() : Prototype

Prototype: Consequences

+ Can add & remove classes at runtime by cloning them as needed
+ Reduced subclassing minimizes/eliminates need for lexical dependencies at run-time
– Every class that used as a prototype must itself be instantiated
– Classes that have circular references to other classes cannot really be cloned

Prototype: Examples

- Design Patterns: Prototype in Java (refactoring.guru)
- Prototype Design Pattern in Java - JournalDev

Singleton

Singleton: Problem

How to:

- Ensure that a class only has one instance
- Easily access the sole instance of a class
- Control its instantiation
- Restrict the number of instances

Applicability

- When there must be exactly one instance of a class & it must be accessible from a well-known access point
- When the sole instance should be extensible by subclassing & clients should be able to use an extended instance without modifying their code

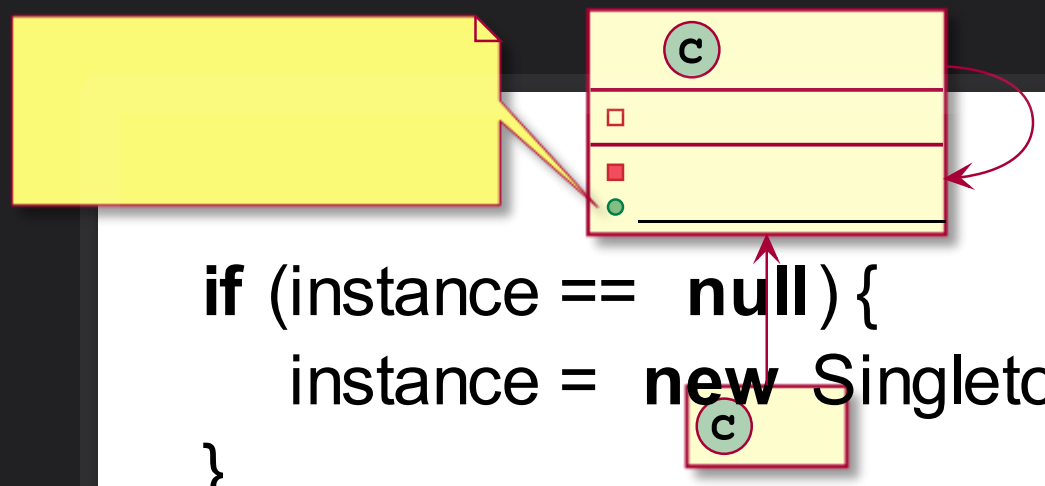
Singleton: Solution

Intent

- Ensure a class only ever has one instance & provide a global point of access

The singleton design pattern describes how to solve such problems:

- Hide the constructors of the class.
- Define a public static operation (getInstance()) that returns the sole instance of the class.




```
if (instance ==  null) {
    instance =  new  Singleton()
}
return  instance
```

Singleton

instance : Singleton

Singleton()
getInstance() : Singleton

Client

Singleton: Consequences

+ Reduces namespace pollution
+ Makes it easy to change your mind & allow more than one instance
+ Allow extension by subclassing
– Same drawbacks of a global if misused
– Implementation may be less efficient than a global
– Concurrency/cache pitfalls & communication overhead

Singleton: Examples

- Design Patterns: Singleton in Java (refactoring.guru)
- Java Singleton Design Pattern Example Best Practices – JournalDev

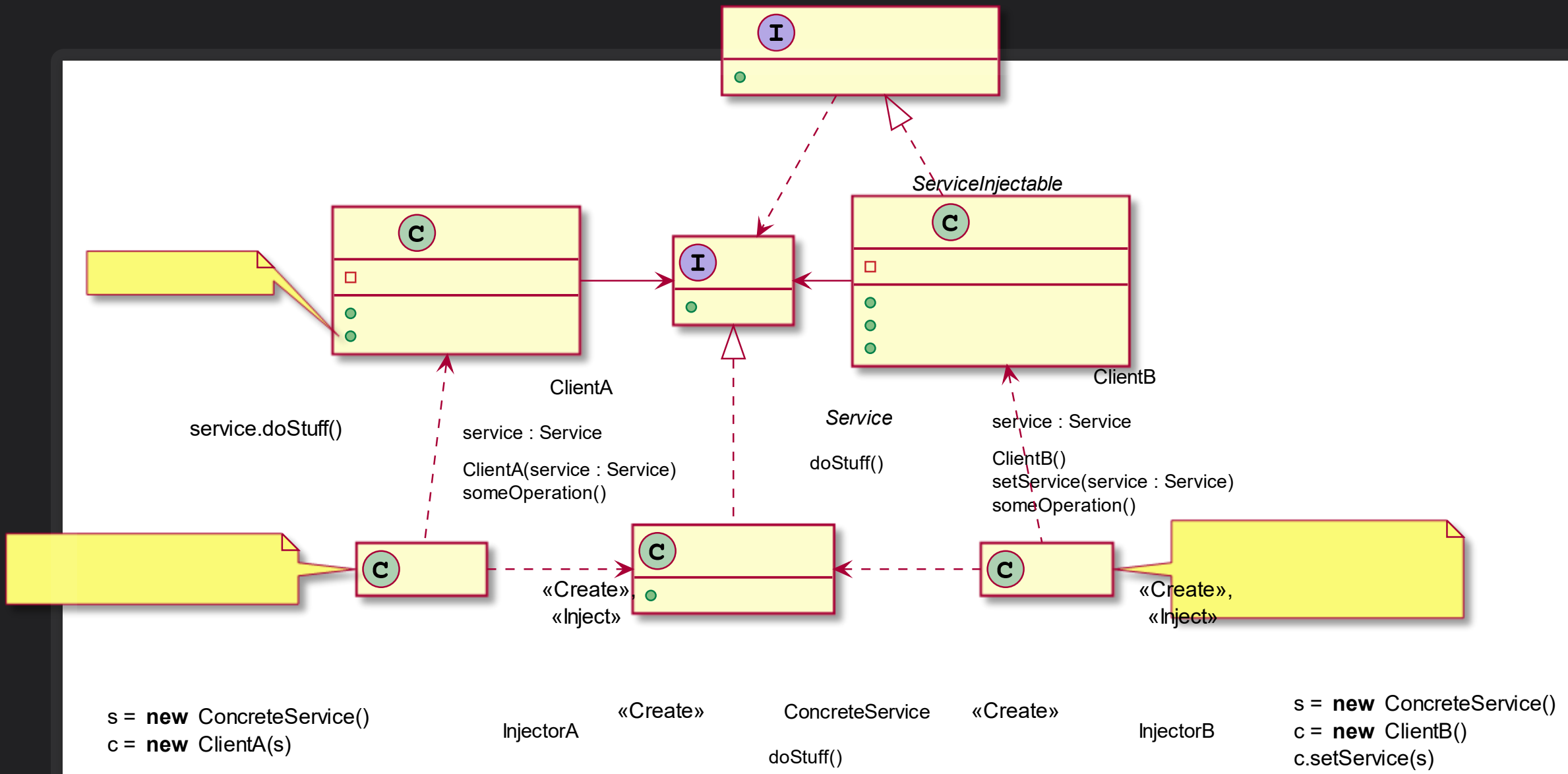
Dependency Injection

Dependency Injection: Problem

- How can a class be independent of how the objects on which it depends are created?
- How can the way objects are created be specified in separate configuration files?
- How can an application support different configurations?

Dependency Injection: Solution

- Dependency injection separates the creation of a client's dependencies from the client's behavior, which promotes loosely coupled programs and the dependency inversion and single responsibility principles.
- Fundamentally, dependency injection is based on passing parameters to a method.
- Dependency injection is an example of the more general concept of inversion of control.
- For details: Inversion of Control Containers and the Dependency Injection pattern (martinfowler.com)


I
ServiceInjectable
C
I
C
ClientA
ClientB
service.doStuff()
service : Service
ClientA(service : Service)
someOperation()
Service
doStuff()
service : Service
ClientB()
setService(service : Service)
someOperation()
C
C
C
«Create»,
«Inject»
«Create»,
«Inject»
s = new ConcreteService()
c = new ClientA(s)
InjectorA
«Create»
ConcreteService
doStuff()
«Create»
InjectorB
s = new ConcreteService()
c = new ClientB()
c.setService(s)

Dependency Injection: Consequences

+ Helps in Unit testing.
+ Boiler plate code is reduced, as initializing of dependencies is done by the injector component.
+ Helps to enable loose coupling, which is important in application programming.
– It’s a bit complex to learn, and if overused can lead to management issues and other problems.
– Many compile time errors are pushed to run-time.

Dependency Injection: Examples

- Using dependency injection in Java - Introduction - Tutorial (vogella.com)
- Java Dependency Injection - DI Design Pattern Example Tutorial – JournalDev
- A quick intro to Dependency Injection: what it is, and when to use it (freecodecamp.org)