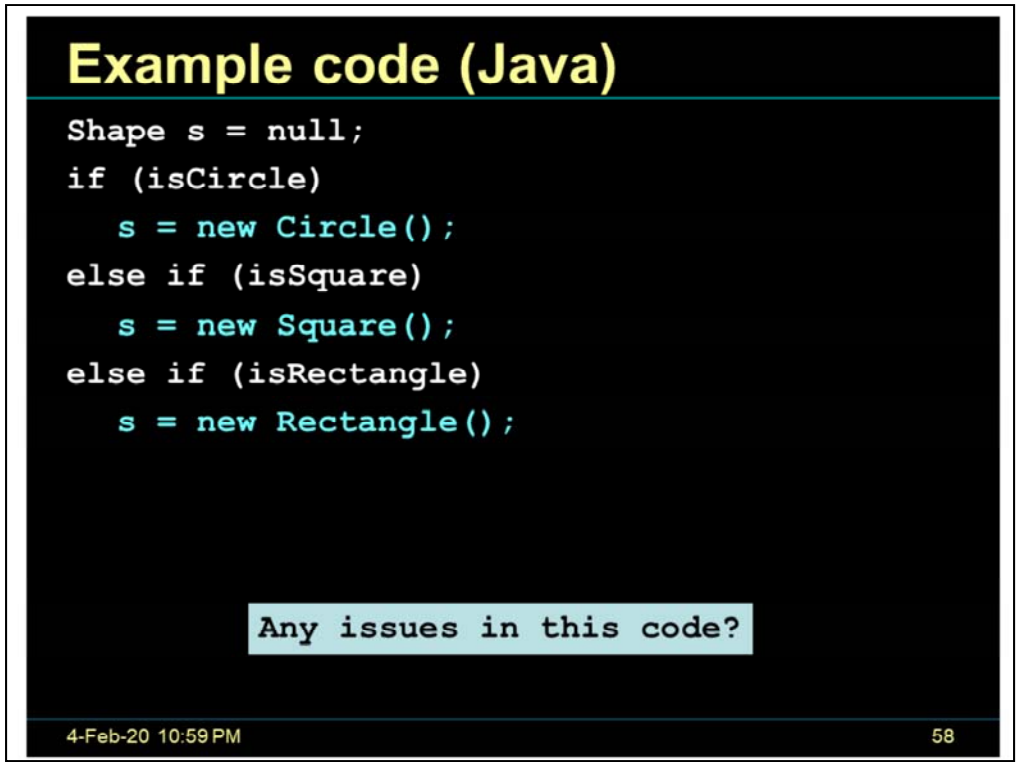

Factory and Abstract
Factory DP
4-Feb-20 10:59 PM
57



Problems in this code:

Are we coding for an implementation or for an interface?

Is the part that changes, isolated from the part that remains constant?

What happens if there is a new kind of Shape?

Are we open for extension but closed for modification?

Example code (C++)

```
Shape *s = 0;
if (isCircle)
   s = new Circle();
else if (isSquare)
   s = new Square();
else if (isRectangle)
   s = new Rectangle();
```

Any issues in this code?

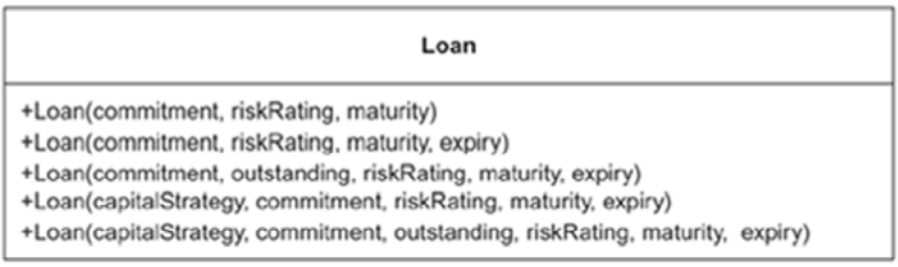
Simple Factory Pattern code (Java)

```
//Shape is an interface implemented by
//   Circle, Square and Rectangle
Shape createShape(ShapeTypes type) {
  Shape s = null;
  if (type == ShapeTypes.CIRCLE)
     s = new Circle();
  else if (type == ShapeTypes.SQUARE)
     s = new Square();
  else if (type == ShapeTypes.RECTANGLE)
     s = new Rectangle();
   return s;
}
```

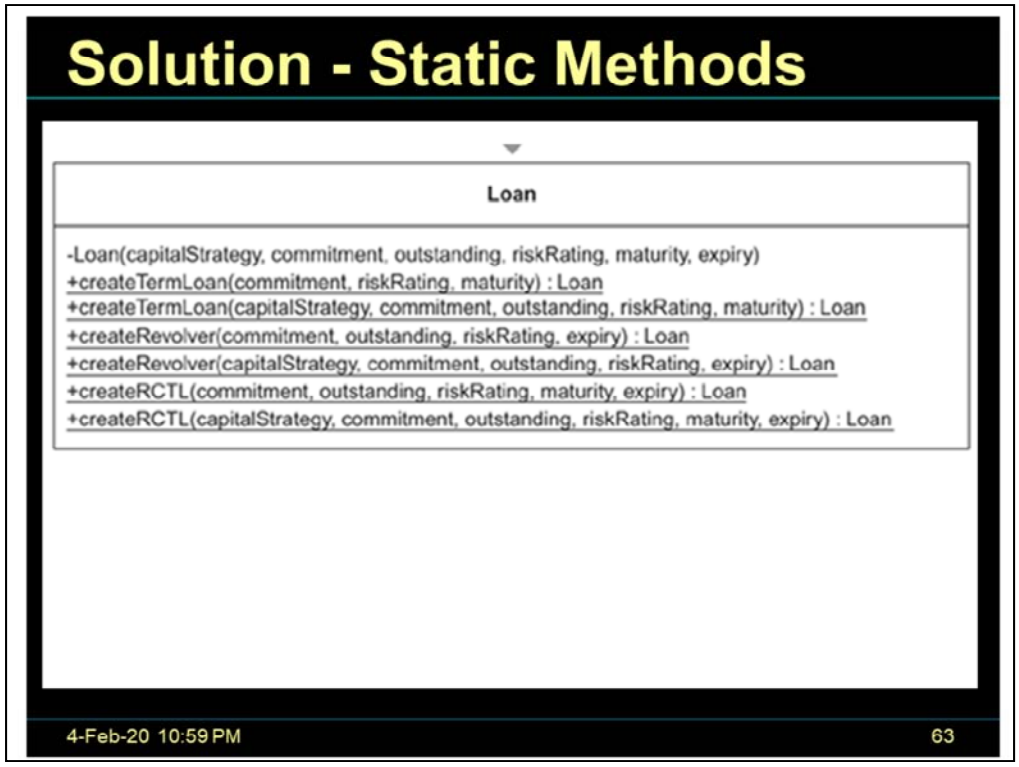
Simple Factory Pattern code (C++)

```
//Shape has only pure virtual functions
//implemented by Circle, Square & Rectangle
Shape *createShape(ShapeTypes type) {
  Shape *s = 0;
  if (type == ShapeTypes.CIRCLE)
    s = new Circle();
  else if (type == ShapeTypes.SQUARE)
    s = new Square();
  else if (type == ShapeTypes.RECTANGLE)
    s = new Rectangle();
   return s;
}
```

Improve


Loan
+Loan(commitment, riskRating, maturity)
+Loan(commitment, riskRating, maturity, expiry)
+Loan(commitment, outstanding, riskRating, maturity, expiry)
+Loan(capitalStrategy, commitment, riskRating, maturity, expiry)
+Loan(capitalStrategy, commitment, outstanding, riskRating, maturity, expiry)

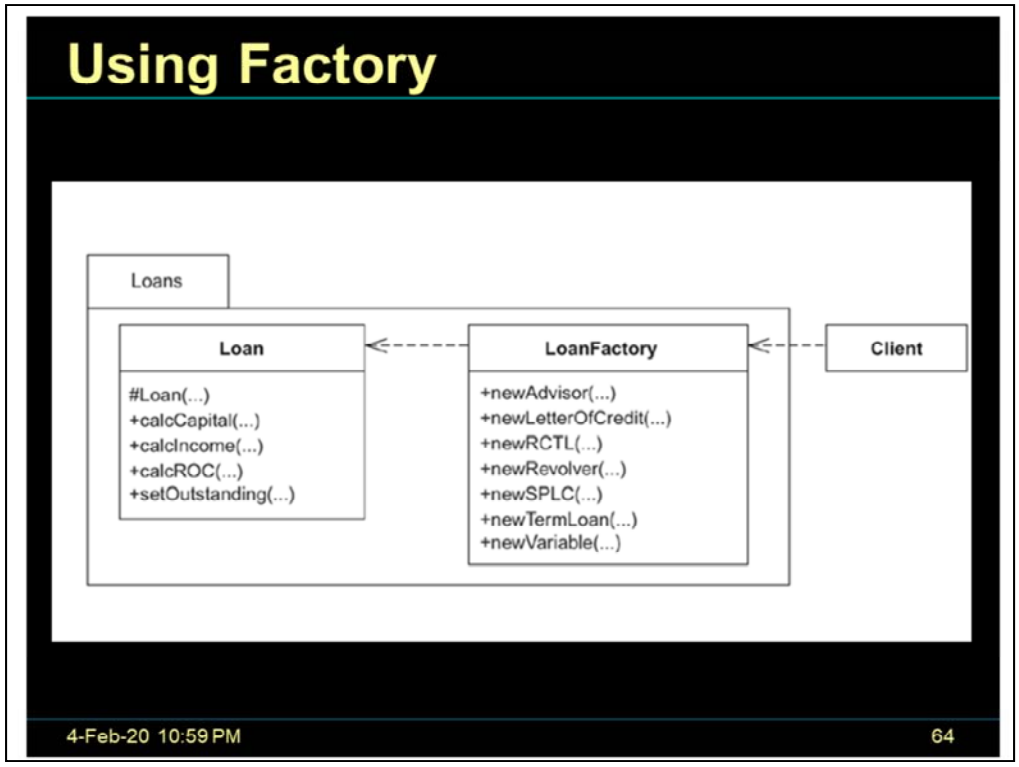
?



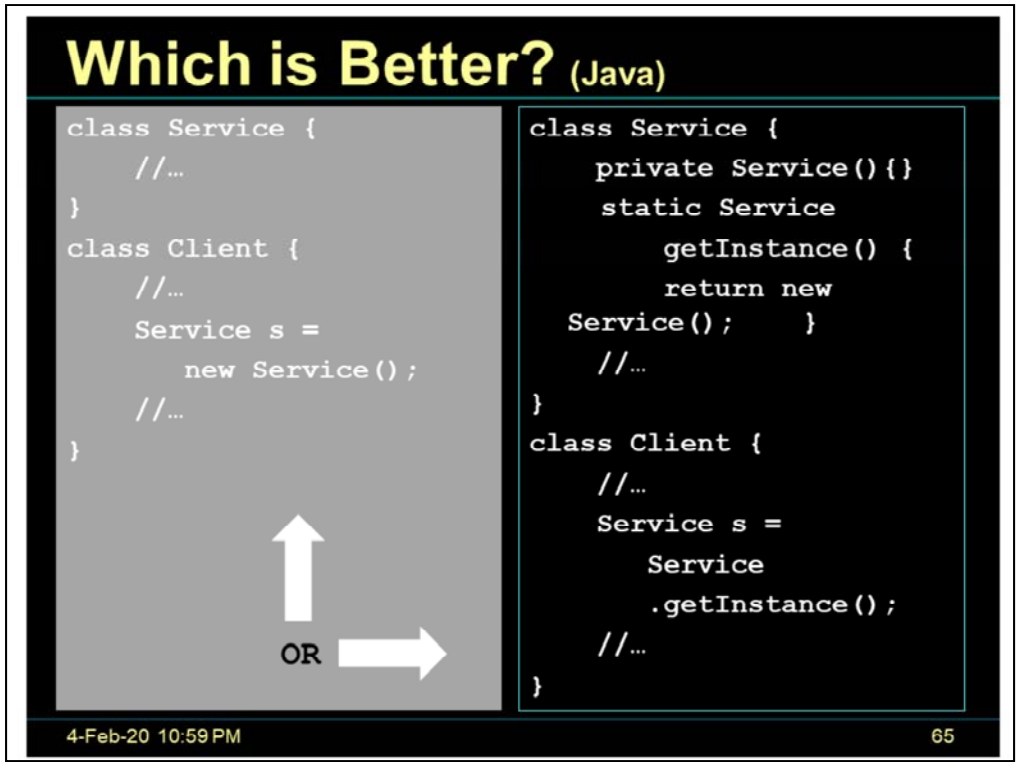
+Communicates what kinds of instances are available better than constructors.

+Bypasses constructor limitations, such as the inability to have two constructors with the same number and type of arguments.

+Makes it easier to find unused creation code.



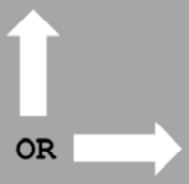
If there are too many methods, then



The right side is better. Service class can be made abstract tomorrow.

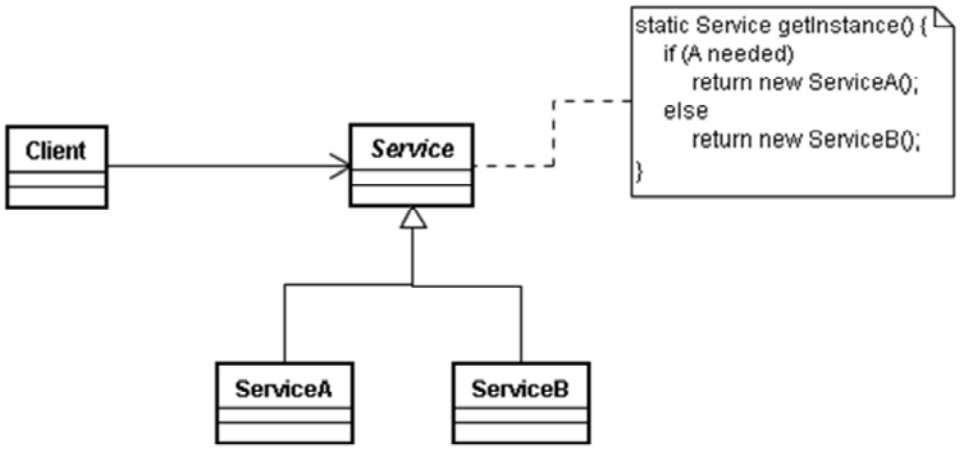
Which is Better? (C++)

```
class Service {
    //…
}
class Client {
    //…
    Service *s =
       new Service();
    //…
}
```

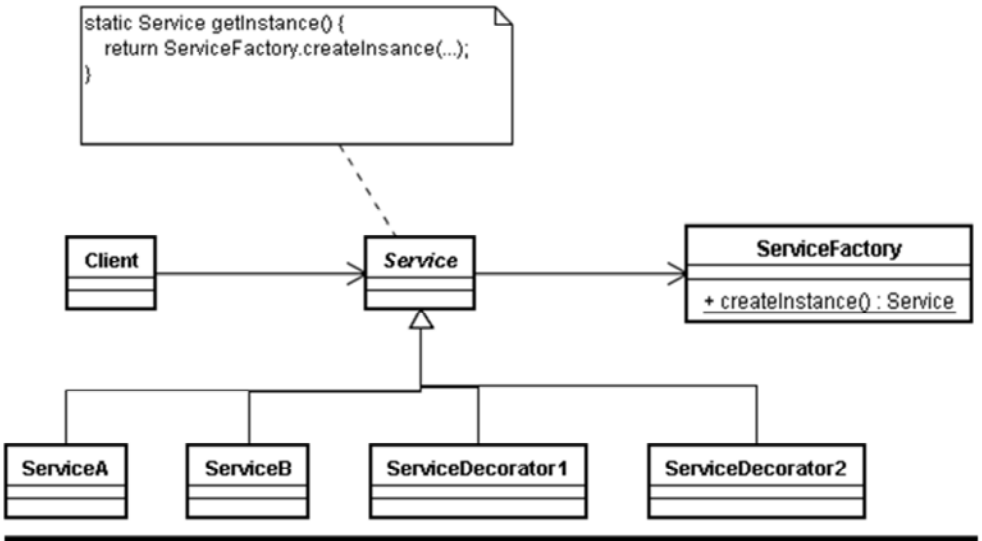


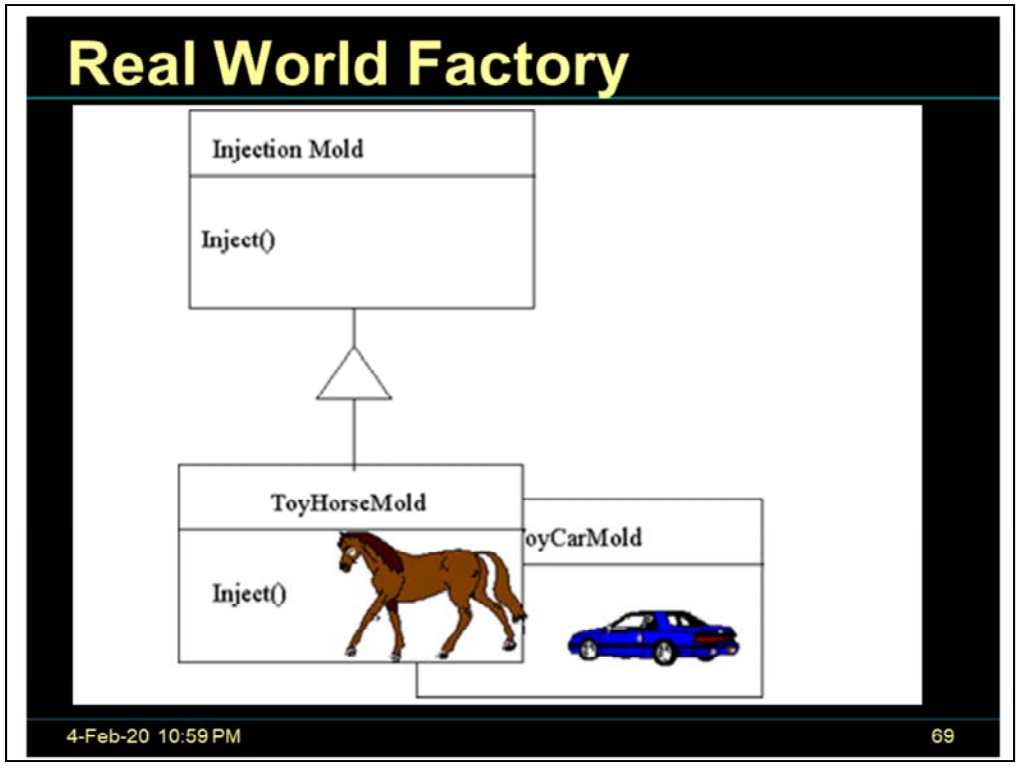

```
class Service {
    Service(){}
    public: static
  Service *getInstance(){
    return new Service();
  } //…
};
class Client { //…
    Service *s =
  Service::getInstance();
    //…
};
```

Service can be Abstract now!

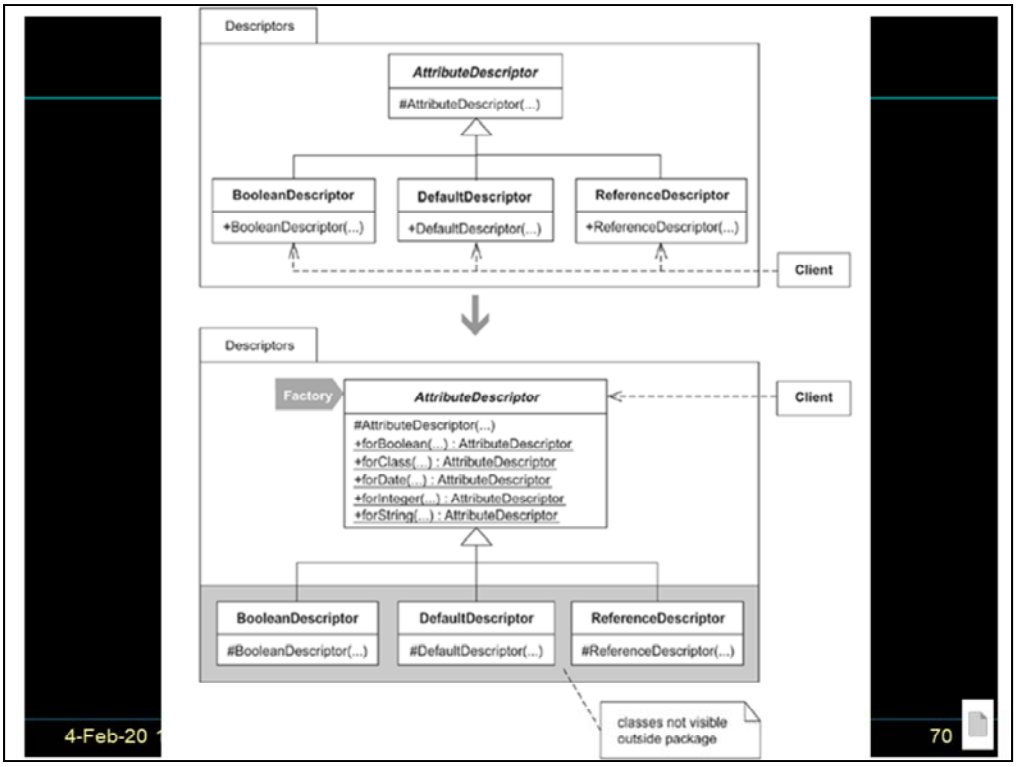


Complex Method to Factory!





The *Factory Method* defines an interface for creating objects, but lets subclasses decide which classes to instantiate. Injection molding presses demonstrate this pattern. Manufacturers of plastic toys process plastic molding powder, and inject the plastic into molds of the desired shapes. The class of toy (car, action figure, etc.) is determined by the mold.



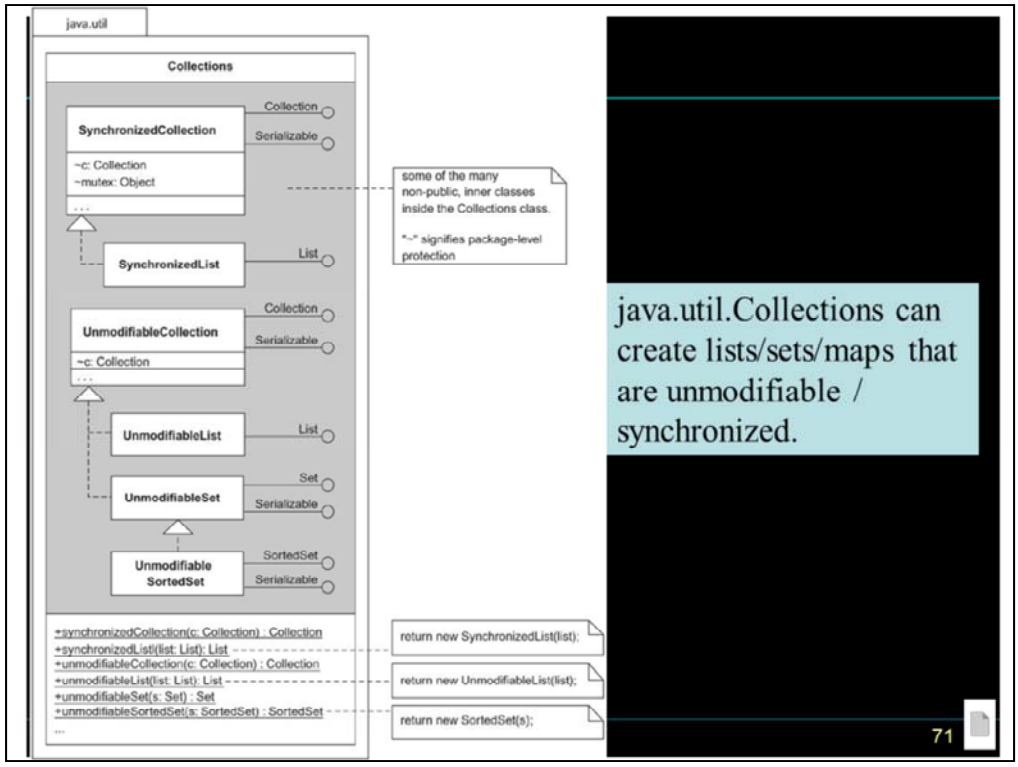
+Simplifies the creation of kinds of instances by making the set available through intention-revealing methods.

+Reduces the "conceptual weight" [Bloch] of a package by hiding classes that don't need to be public.

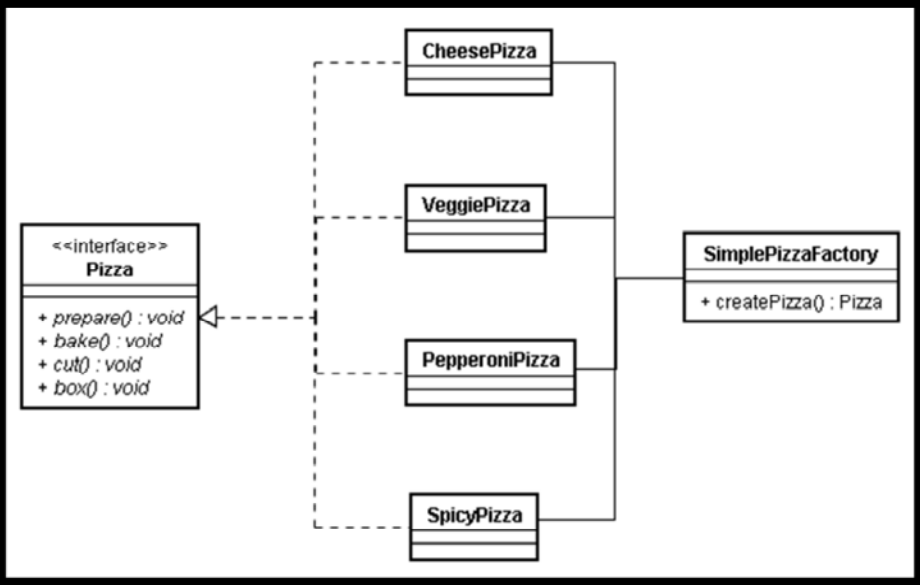
+Helps enforce the mantra "program to an interface, not an implementation"

–Requires new/updated Creation Methods when new kinds of instances must be created.

–Limits customization when clients can only access a Factory's binary code, not its source code.


java.util
Collections
SynchronizedCollection
Collection
Serializable
~c: Collection
~mutex: Object
...
SynchronizedList
List
UnmodifiableCollection
Collection
Serializable
~c: Collection
...
UnmodifiableList
List
UnmodifiableSet
Set
Serializable
Unmodifiable SortedSet
SortedSet
Serializable
some of the many non-public, inner classes inside the Collections class.
"~" signifies package-level protection
+synchronizedCollection(c: Collection) : Collection
+synchronizedList(list: List): List
+unmodifiableCollection(c: Collection) : Collection
+unmodifiableList(list: List): List
+unmodifiableSet(s: Set) : Set
+unmodifiableSortedSet(s: SortedSet) : SortedSet
...
return new SynchronizedList(list);
return new UnmodifiableList(list);
return new SortedSet(s);
java.util.Collections can create lists/sets/maps that are unmodifiable / synchronized.
71

Simple Pizza Factory



Simple Factory Pattern code(Java)

```
Pizza createPizza(PizzaTypes type) {
  Pizza pizza = null;
  if (type == PizzaTypes.CHEESE)
    pizza = new CheesePizza();
  else if (type == PizzaTypes.GREEK)
    pizza = new GreekPizza();
  else if (type == PizzaTypes.PEPPERONI)
    pizza = new PepperoniPizza();
   return pizza;
}
```

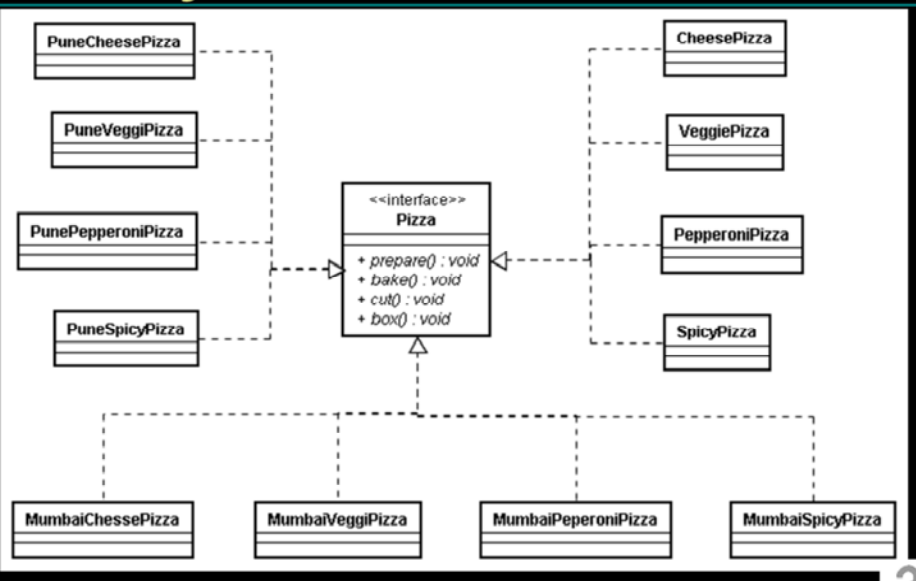
Simple Factory Pattern code(C++)

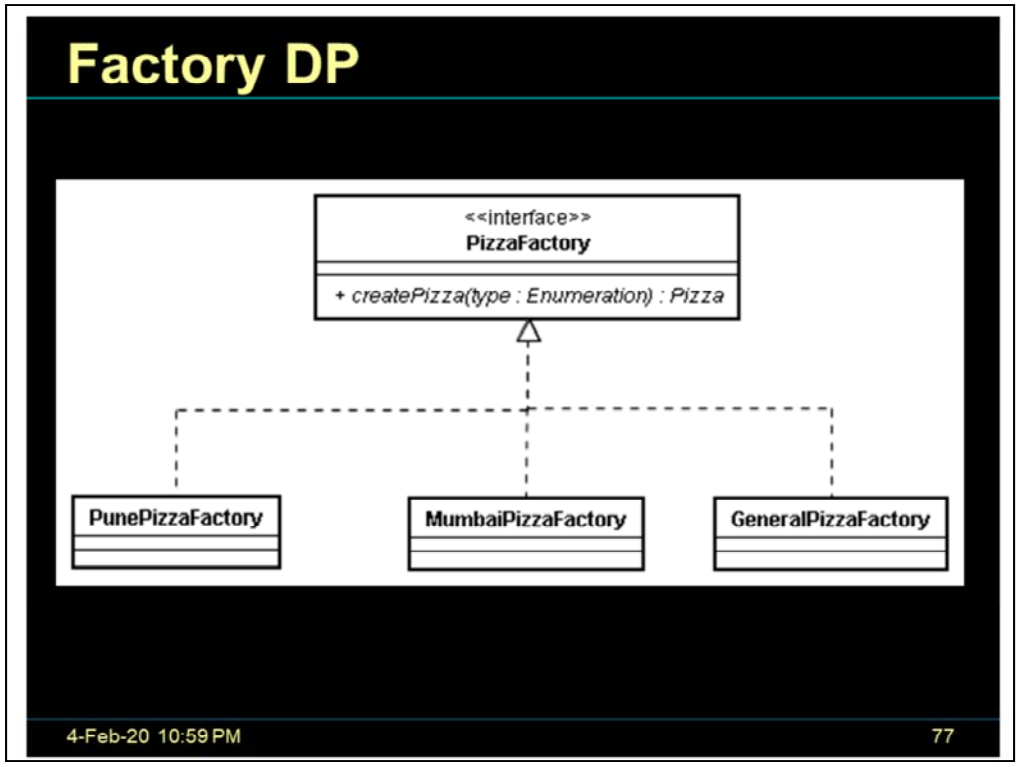
```
Pizza *createPizza(PizzaTypes type) {
  Pizza *pizza = 0;
  if (type == PizzaTypes.CHEESE)
    pizza = new CheesePizza();
  else if (type == PizzaTypes.GREEK)
    pizza = new GreekPizza();
  else if (type == PizzaTypes.PEPPERONI)
    pizza = new PepperoniPizza();
   return pizza;
}
```

Factory DP

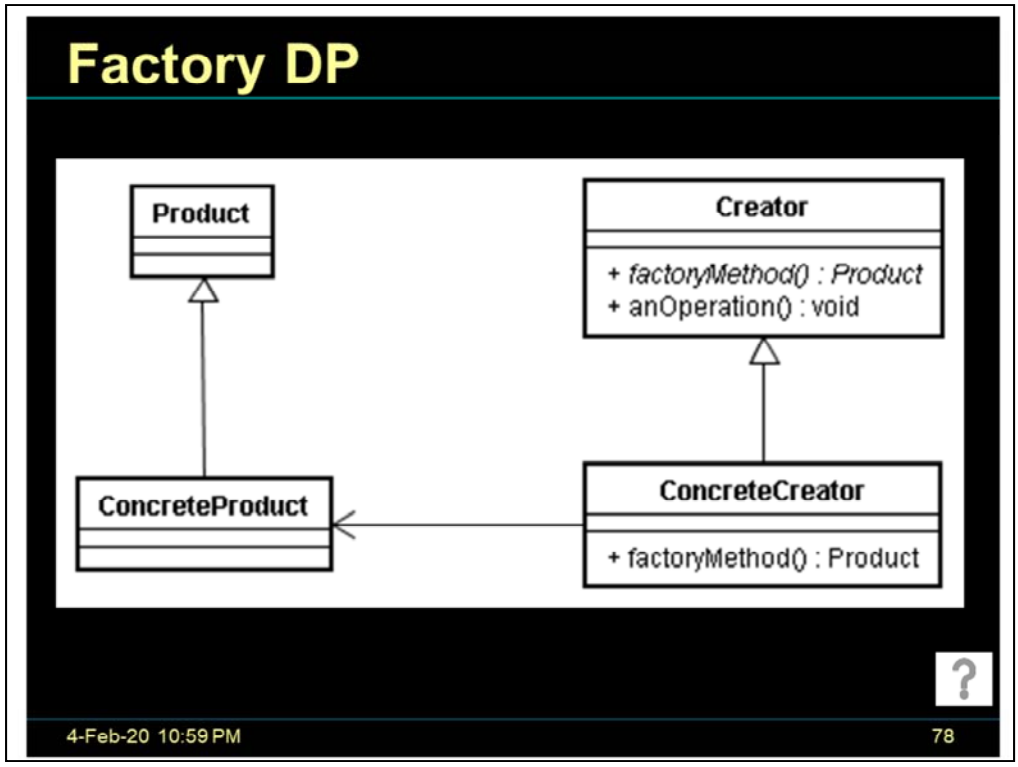
- Now
  - Mumbai has its own set of customized pizzas.
  - Pune has its own set of customized pizza.
  - Other metros do not yet have any customized pizzas.
- How does the design change?

Factory DP


PuneCheesePizza
PuneVeggiPizza
PunePepperoniPizza
PuneSpicyPizza
<<interface>>
Pizza
+ prepare() : void
+ bake() : void
+ cut() : void
+ box() : void
CheesePizza
VeggiePizza
PepperoniPizza
SpicyPizza
MumbaiChessePizza
MumbaiVeggiPizza
MumbaiPeperoniPizza
MumbaiSpicyPizza



Code example



It defines an interface for creating an object but lets subclasses/implantations decide which class to instantiate.

Factory method lets a class defer instantiation to the subclasses.

Advantages

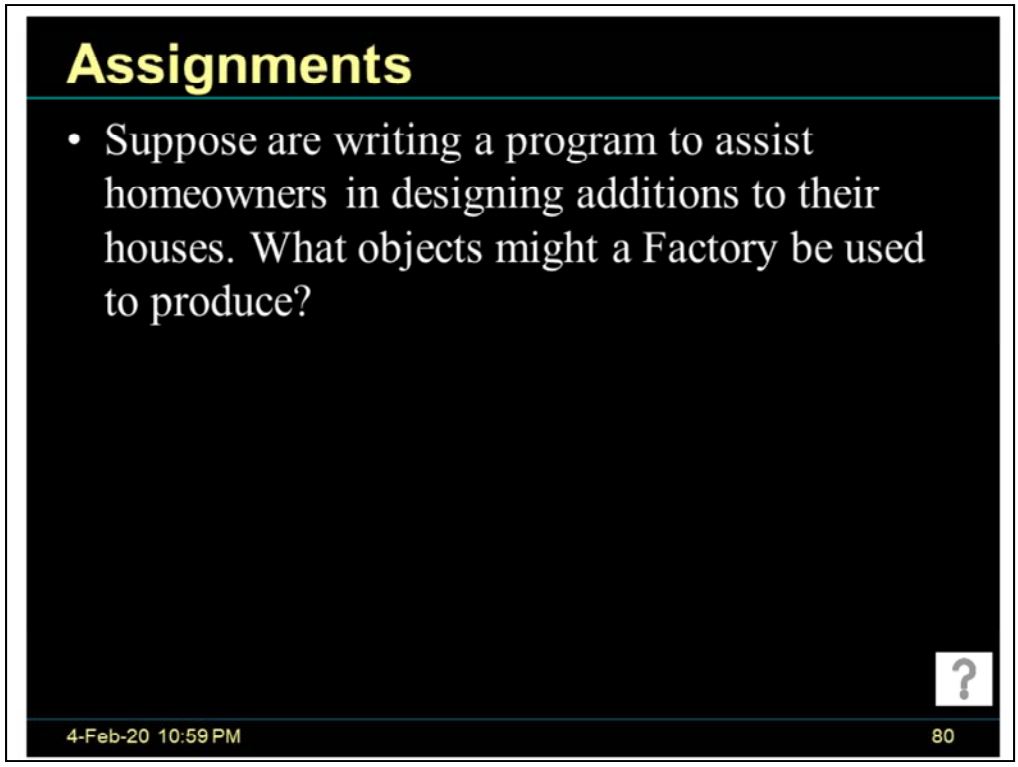
Avoids duplication of code.

One place for maintenance of code.

We depend upon interfaces and not implementations.

Usage of Factory DP

- We consider using a Factory pattern when
  - We can't anticipate the class, whose object must be create.
  - We want to localize the knowledge of which class gets created.
- In Java, Iterator method on Collection uses a Factory to generate the actual object.

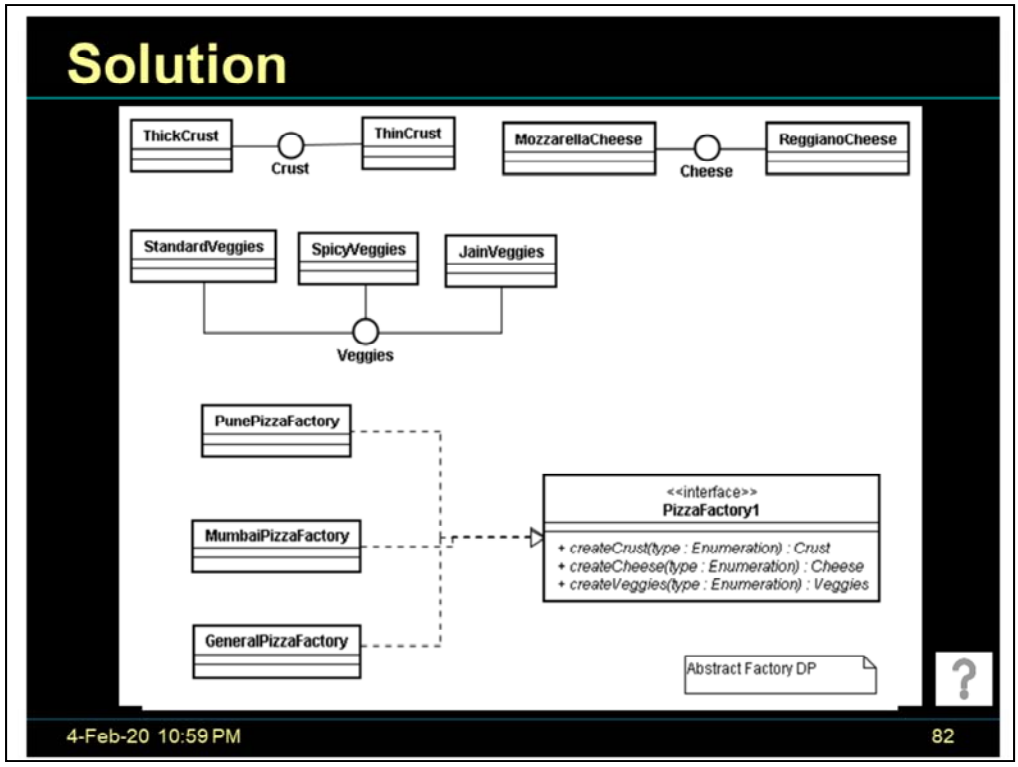


Assignment

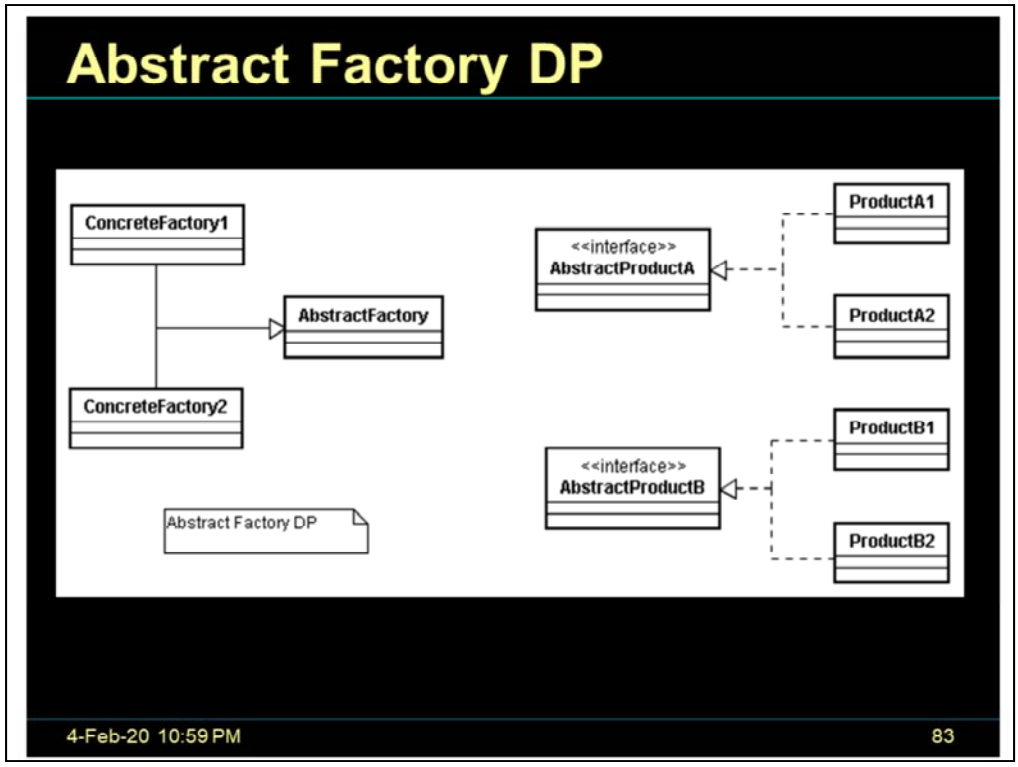
Consider a personal checkbook management program like Quicken. It manages several bank accounts and investments and can handle our bill paying. Where could we use a Factory pattern in designing a program like that?

Problem

- Now, each store wants to use different crust, veggies and cheese. How should our factory handle it?



Code example present



The abstract factory pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes.

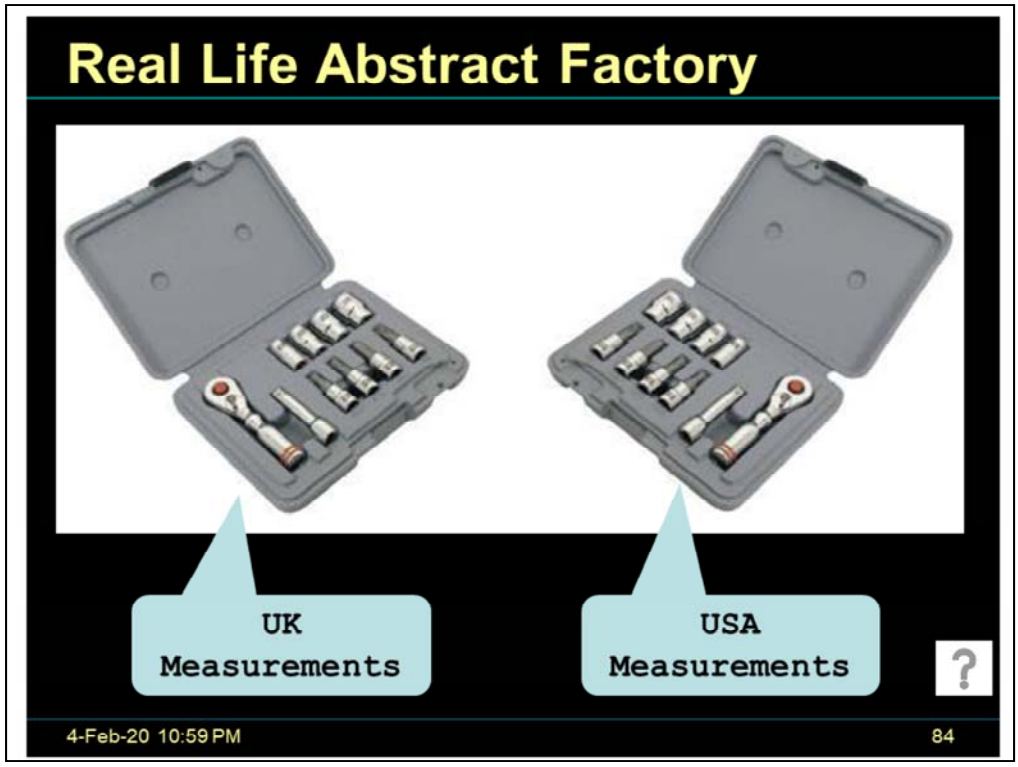
It is used when

a system should be independent of how its products are created, composed and represented.

a system should be configured with one of multiple families of products.

a family of related product objects is designed to be used together and we need to enforce this constraint.

we want to provide a class library of products and we want to reveal the interfaces and not their implementations.



One contains a set of wrenches (box-end wrenches, socket wrenches, closed-end wrenches) that are designed to work on a car that comes from overseas, and thus has metric measurements (centimeters, millimeters, etc...)

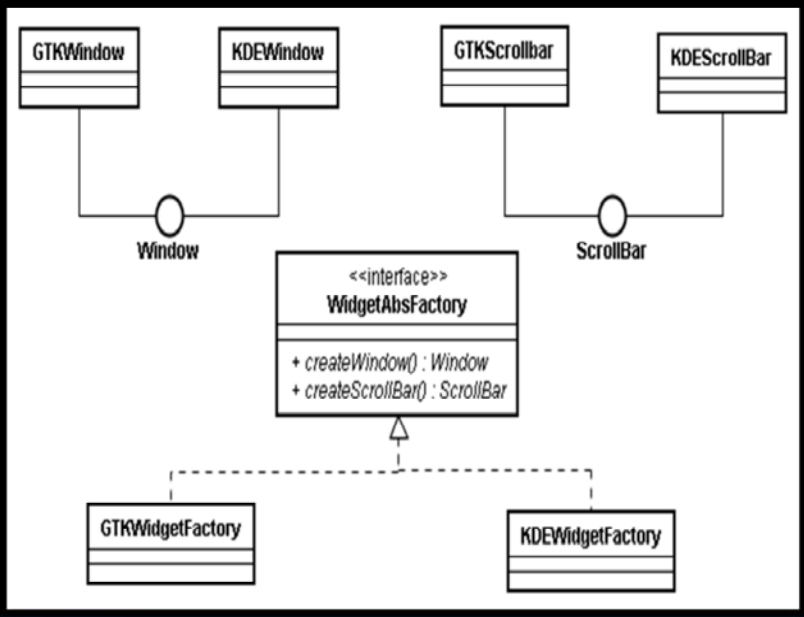
The other contains an identical set (box-end, socket, closed-end) that are designed instead for an engine with "English" measurements (inches, quarter-inches, and so forth)

I do not have engines that I work on which have some bolts that are metric and others which are English, and therefore I only use one set of wrenches or the other. The two toolkits encapsulate the difference: I choose the toolkit that applies to a given engine, and I know that all the wrenches I pull from it will be appropriate, making my maintenance process much simpler and allowing me to concentrate on the tune-up or other process I am trying to accomplish.

Example of Abstract DP

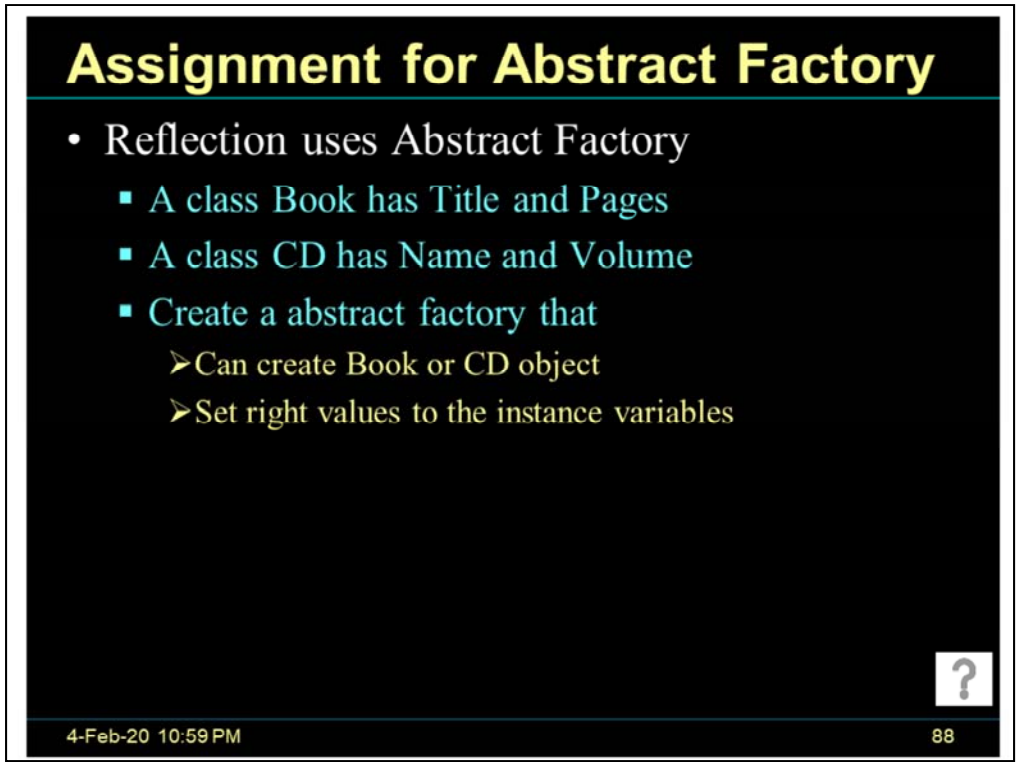
- We want portability across various GUIs.
  - We create an abstract factory object appropriate to the GUI that we are working with,
  - When we ask it for a menu, button, slider, etc. it will automatically create the appropriate version of that item for the GUI.
- Thus we're able to isolate, in one place, the effect of changing from one GUI to another.
  - Java API uses this pattern in java.awt.Toolkit to create objects that work with the native windowing system. The concrete factory class it uses is determined by initialization code, and the singleton concrete factory object is returned by its getDefaultToolkit method.

Example of Abstract Factory



Example of Abstract Factory

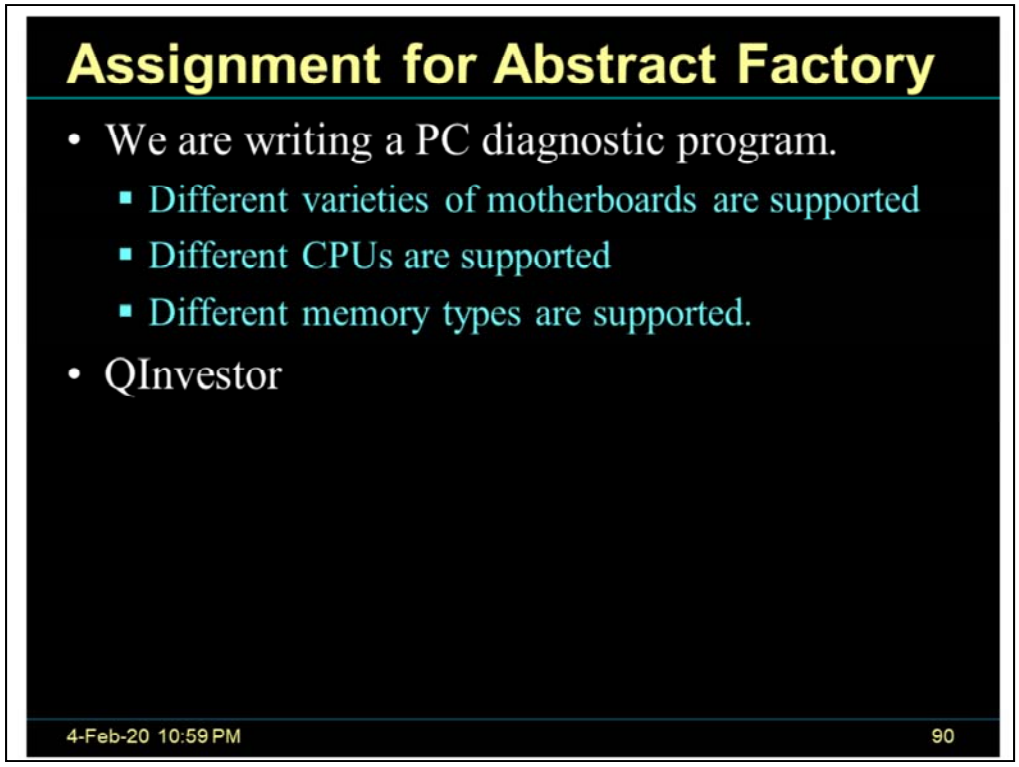
- JDBC drivers use this pattern.
  - Connection, Statement and ResultSet are different interfaces.
  - Each JDBC driver creates the appropriate family of objects for the interfaces.



This example is NOT possible in C++ because it uses Reflection.

Assignment for Abstract Factory

- We are writing a program to plan the layout of gardens.
  - A garden may be a fruit garden, vegetable garden or flower garden.
  - For each garden, we have certain recommended border plants, center plants and shade plants.



We have different abstract factories for each vendor e.g. Dell, Compaq, etc.