Chapter 7
Creational Design Pattern

- □ Factory
- □ Singleton
- □Abstract Factory
- □ Prototype

7.1 - Factory Design Pattern

Factory

Design Purpose

Create individual objects in situations where the constructor alone is inadequate.

Design Pattern Summary

Use methods to return required objects.

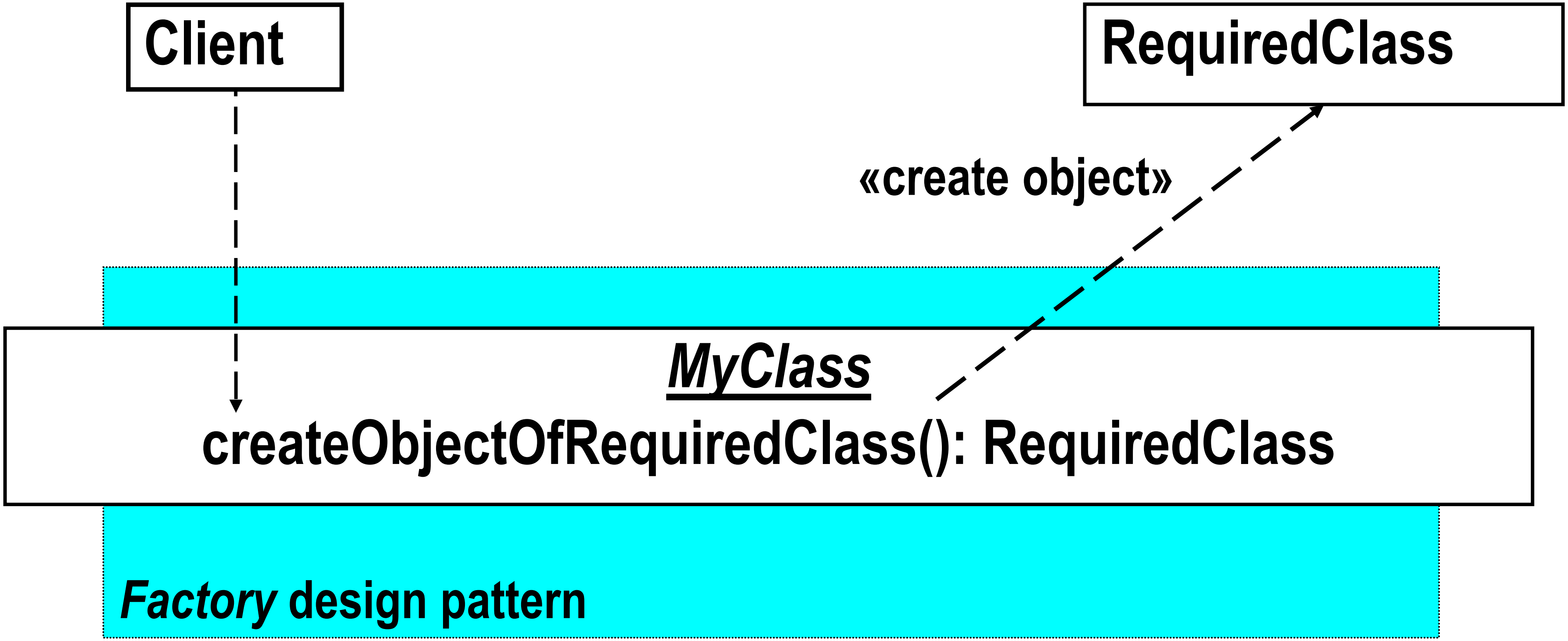
Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Factory Interface for Clients

Demonstrates the use of a static method to create an instance of a class

```
public static void main(String[] args)
{
RequiredClass instanceOfRequiredClass = MyClass.getNewInstanceOfRequiredClass();

} // End main
```

Factory Class Model



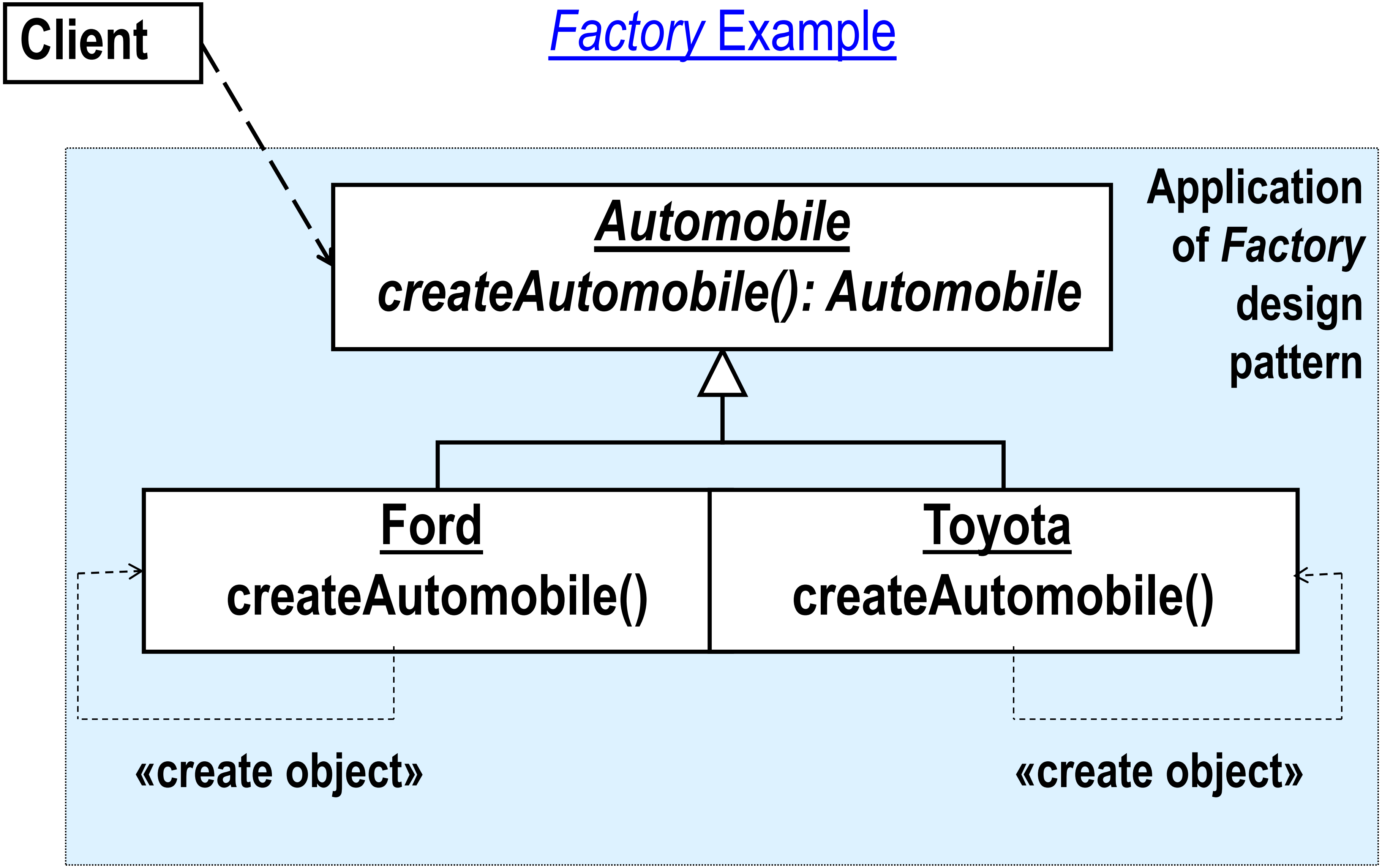
Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Design Goal At Work: → *Reusability* and *Corrrectness* ←

We want to write code about automobiles in general: Code that applies to any make, exercised repeatedly (thus reliably).

Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Factory Example

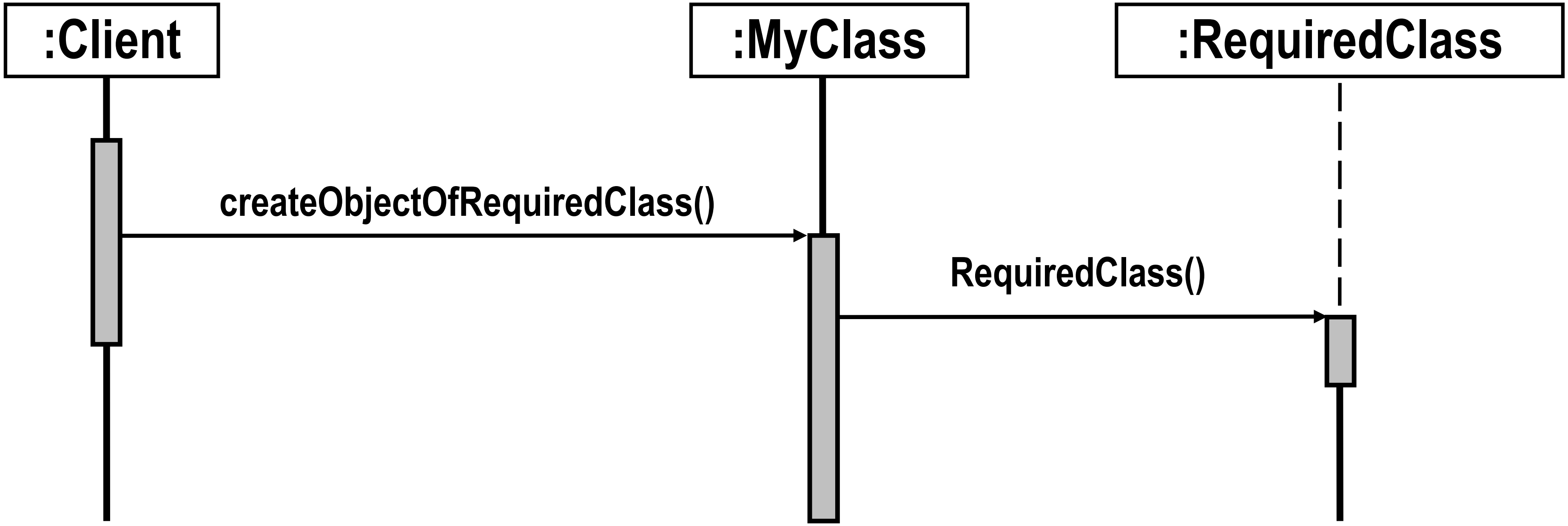


Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Example Code

```
class Ford extends Automobile
{

   static Automobile createAutomobile()
   {
   return new Ford();
   } // End createAutomobile

} // End class
```

Sequence Diagram for *Factory*



:Client

:MyClass

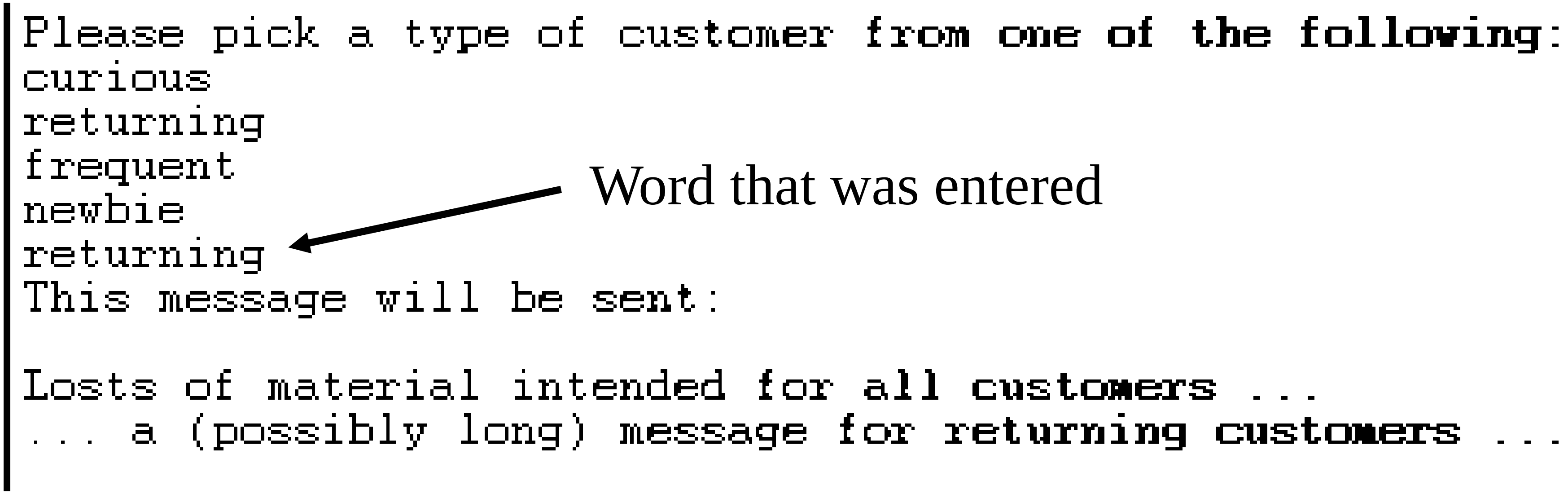
:RequiredClass

createObjectOfRequiredClass()

RequiredClass()

Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Typical Output of E-Mail Generation Example



```
Please pick a type of customer from one of the following:
curious
returning
frequent
newbie
returning
This message will be sent:

Losts of material intended for all customers ...
... a (possibly long) message for returning customers ...
```

Word that was entered

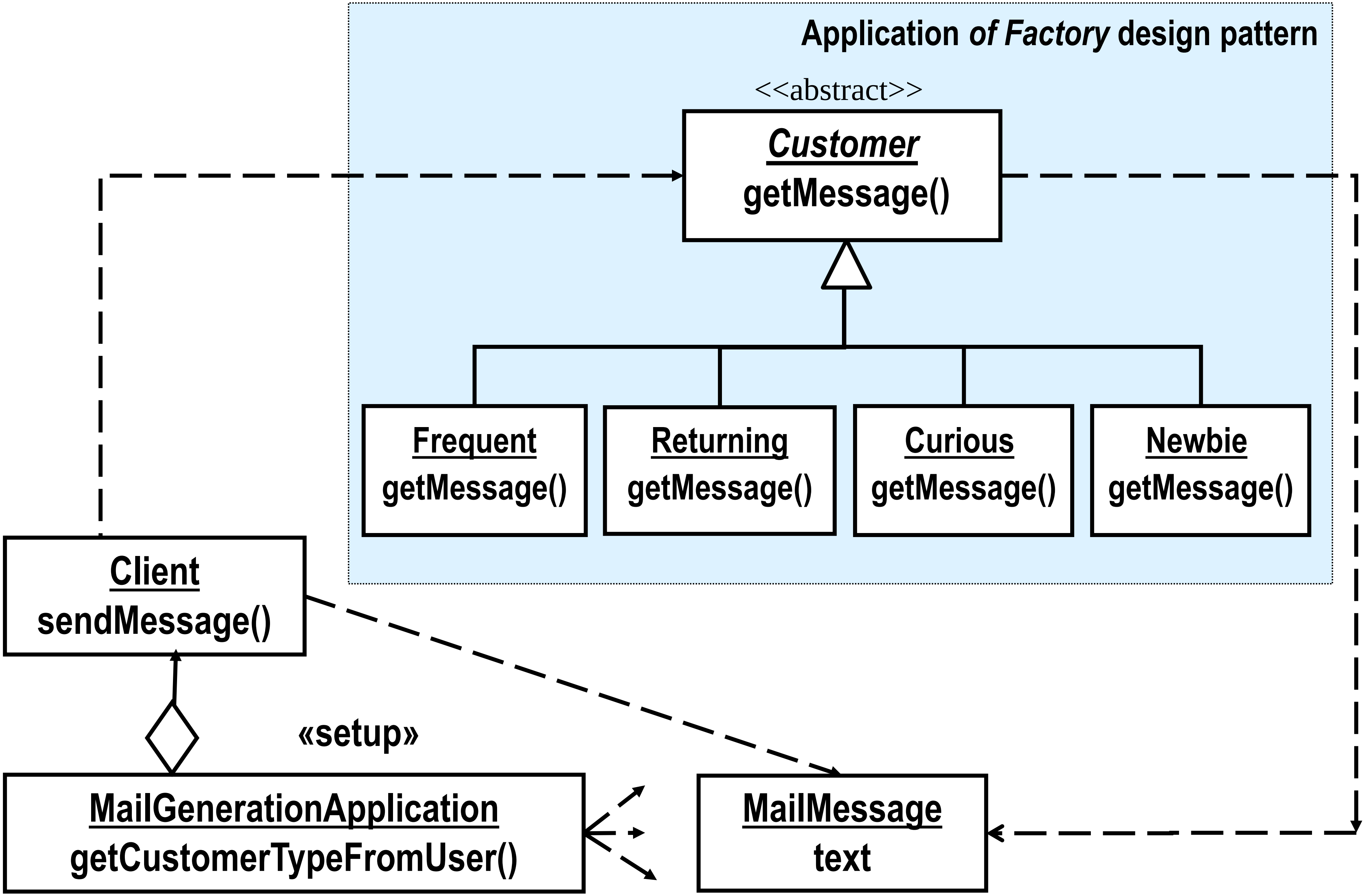
Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Design Goals At Work: → *Correctness and Reusability* ←

We want to separate the code common to all types of customers. We want to separate the specialized code that generates e-mail for each type of customer. This makes it easier to check for correctness and to reuse parts.

Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Factory: Email Generation Example



Application *of Factory* design pattern

<<abstract>>

Customer

getMessage()

Frequent

getMessage()

Returning

getMessage()

Curious

getMessage()

Newbie

getMessage()

Client

sendMessage()

«setup»

MailGenerationApplication

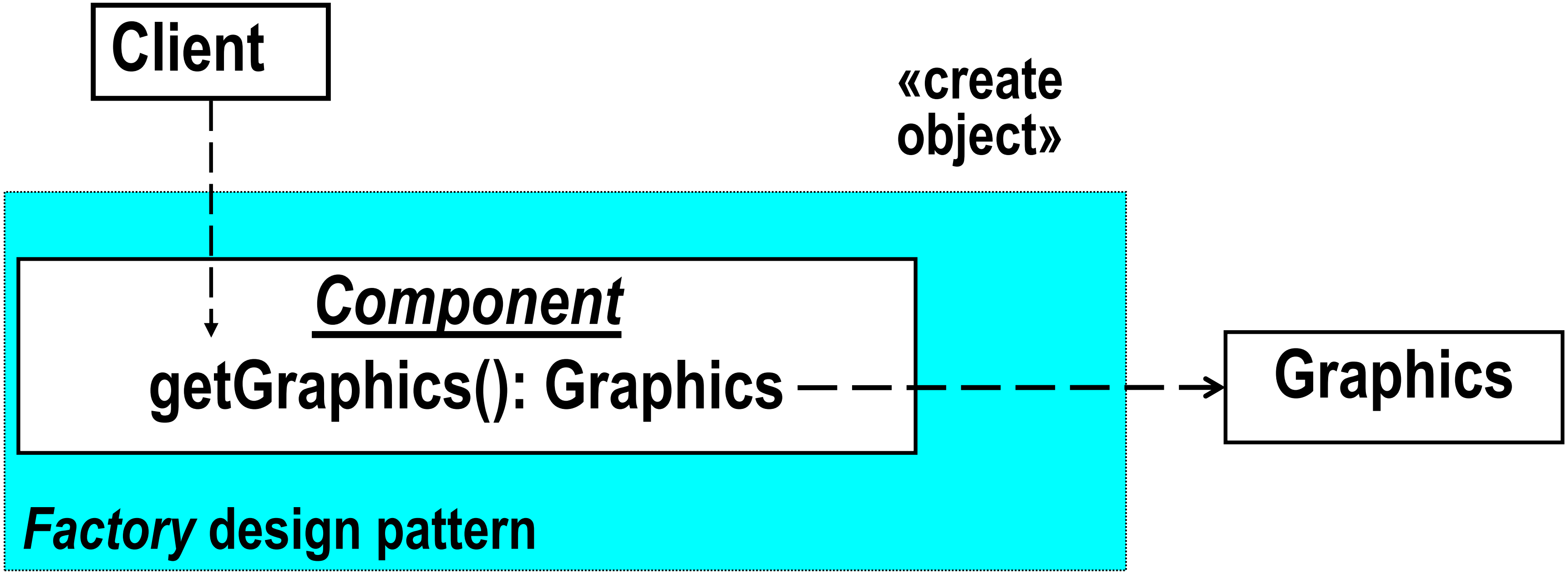
getCustomerTypeFromUser()

MailMessage

text

Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Factory Applied to *getGraphics()* in Java



```
public static Box createVerticalBox()

public static Box createHorizontalBox()
```

Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

7.2 - Singleton Design Pattern

Key Concept: → Singleton Design Pattern ←

-- when a class has exactly one instance.

Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Singleton Design Purpose

Ensure that there is exactly one instance of a class *S*. Be able to obtain the instance from anywhere in the application.

Design Pattern Summary

Make the constructor of *S* private; define a private static attribute for *S* of type *S;* define a public accessor for it.

Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Design Goal At Work: → *Correctness* ←

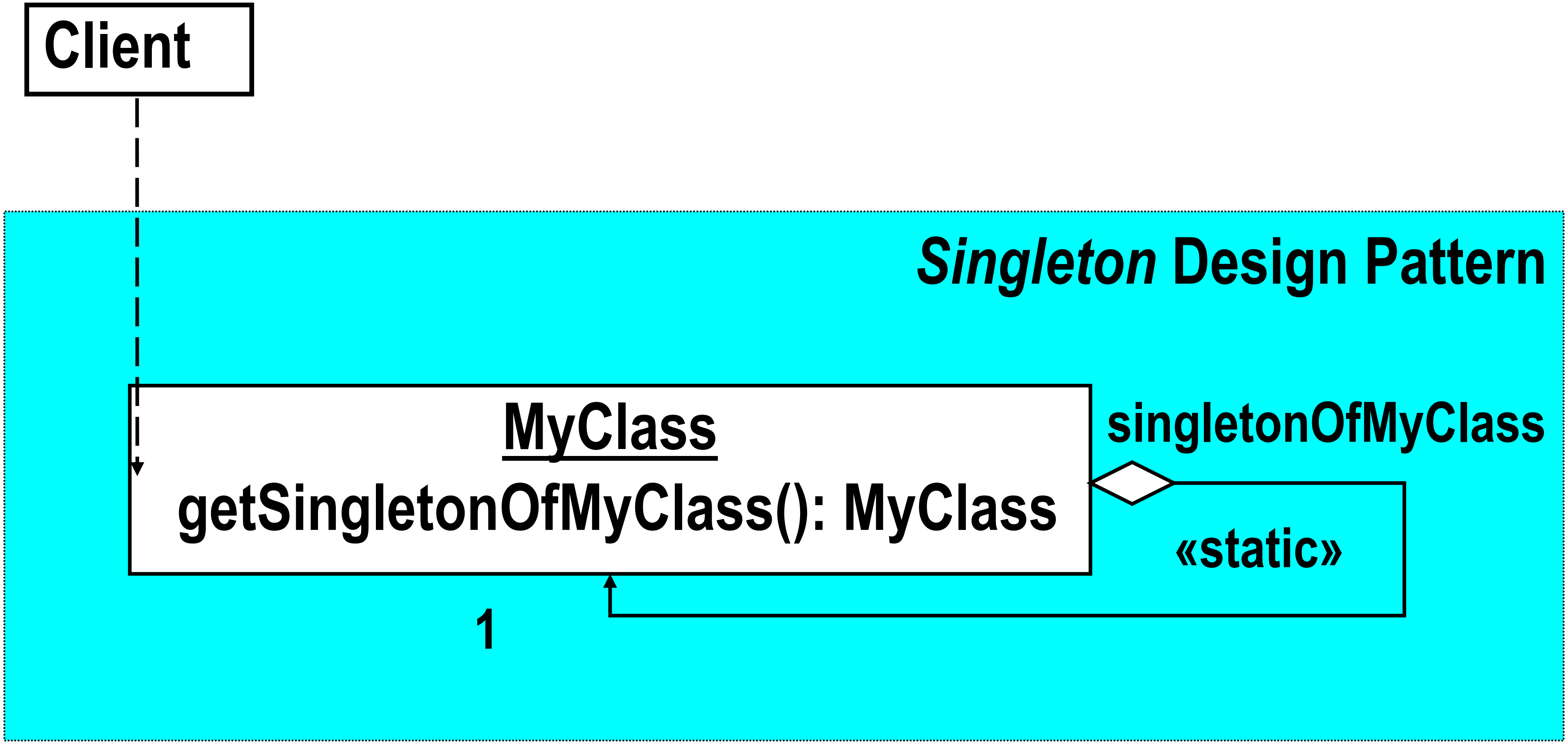
Singleton enforces the intention that only one *User* object exists, safeguarding the application from unanticipated *User* instance creation.

Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

The Singleton Interface for Clients

```
User mainUser = User.getTheUser();
```

Singleton: Class Model



Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

The *Singleton* Design Pattern -- applied to *MyClass*

1. **Define a private static member variable of MyClass of type MyClass**

```
private static MyClass singletonOfMyClass = new MyClass();
```

1. **Make the constructor of MyClass private**

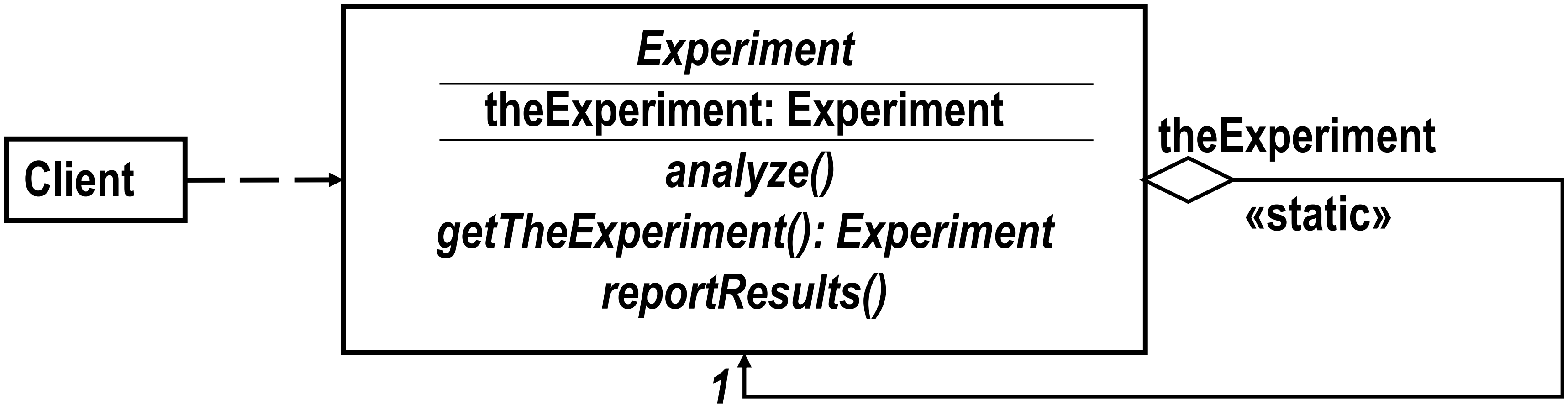
```
private MyClass() { /* …. constructor code …. */ };
```

1. **Define a public static method to access the member**

```
public static MyClass getSingletonOfMyClass()
{
    return singletonOfMyClass;
}
```

Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Application of *Singleton* to *Experiment* Example



Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Key Concept: → Singleton Design Pattern ←

When a class must have exactly one instance, make the constructor private and the instance a private static variable with a public accessor.

Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Example Code

```
public class Runtime
{
private static Runtime currentRuntime = new Runtime();

// Returns the runtime object associated with the current
// Java application.

   public static Runtime getRuntime()
   {
   return currentRuntime;
   }

private Runtime() { }

}
```

7.3 - Abstract Factory Design Pattern

Abstract Factory

Design Purpose

“Provide an interface for creating families of related or dependent objects without specifying their concrete classes.”*

Design Pattern

Capture family creation in a class containing a factory method for each class in the family.

* Gamma *et al*

Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Word Processor Interaction *1 of 2*

```
---> Enter title:
My Life

---> Enter Heading or “-done”:
Birth

---> Enter text:
I was born in a small mountain hut
….

---> Enter Heading or “-done”:
Youth
```

```
---> Enter text:
I grew up playing in the woods …

---> Enter Heading or “-done”:
Adulthood
….

---> Enter Heading or “-done”:
-done
```

(*continued*)

Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Word Processor Interaction *2 of 2*: Output Options

```
….
>> Enter the style you want displayed:
big
```

Option 1

```
        ----- Title: MY LIFE -----

Section 1. --- BIRTH ---
I was born in a mountain hut ….

Section 2. --- YOUTH ---
I grew up sturdy …

Section 3. --- ADULTHOOD ---
….
```

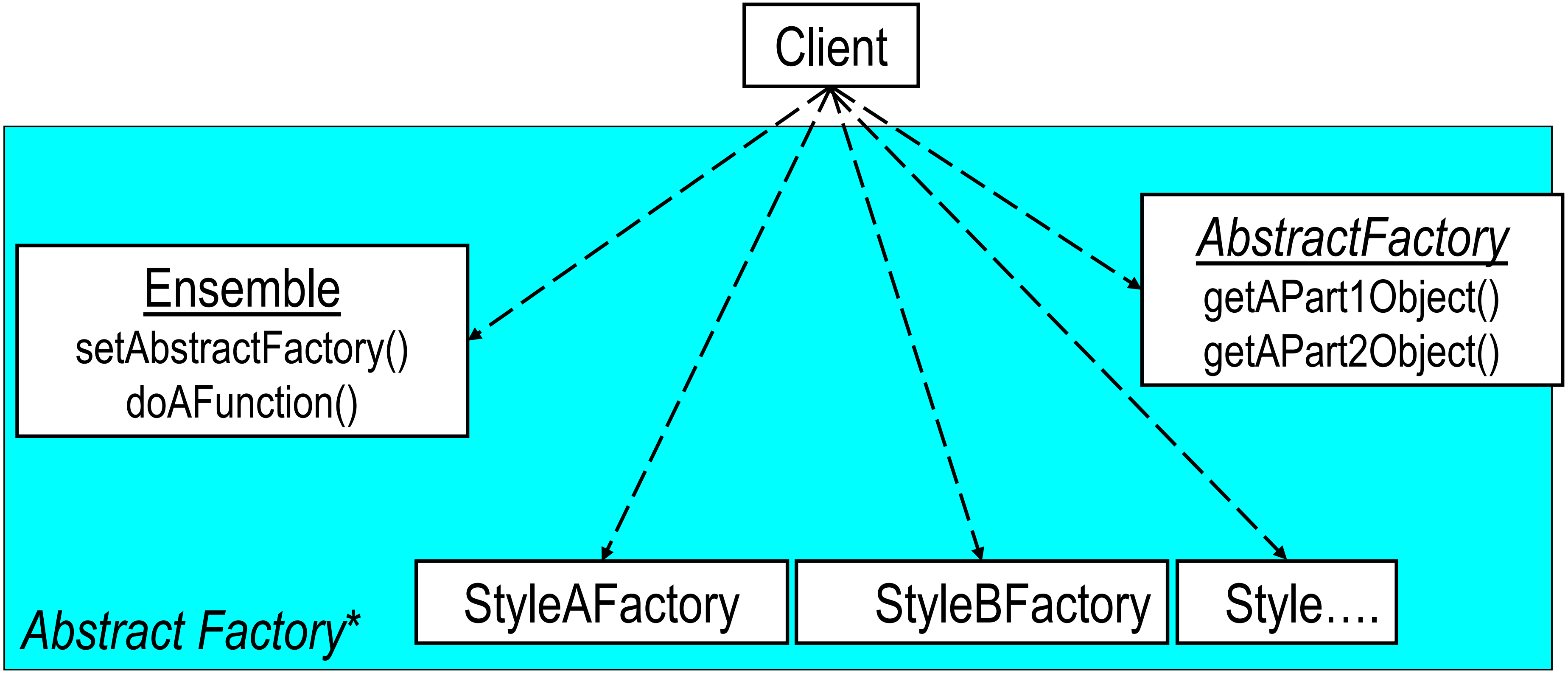
```
….
>> Enter the style you want displayed:
small
```

Option 2

```
My Life

Birth
I was born in a mountain hut ….

Youth
I grew up sturdy …

Adulthood
```

Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Abstract Factory Interface



Client

Ensemble
setAbstractFactory()
doAFunction()

AbstractFactory
getAPart1Object()
getAPart2Object()

StyleAFactory

StyleBFactory

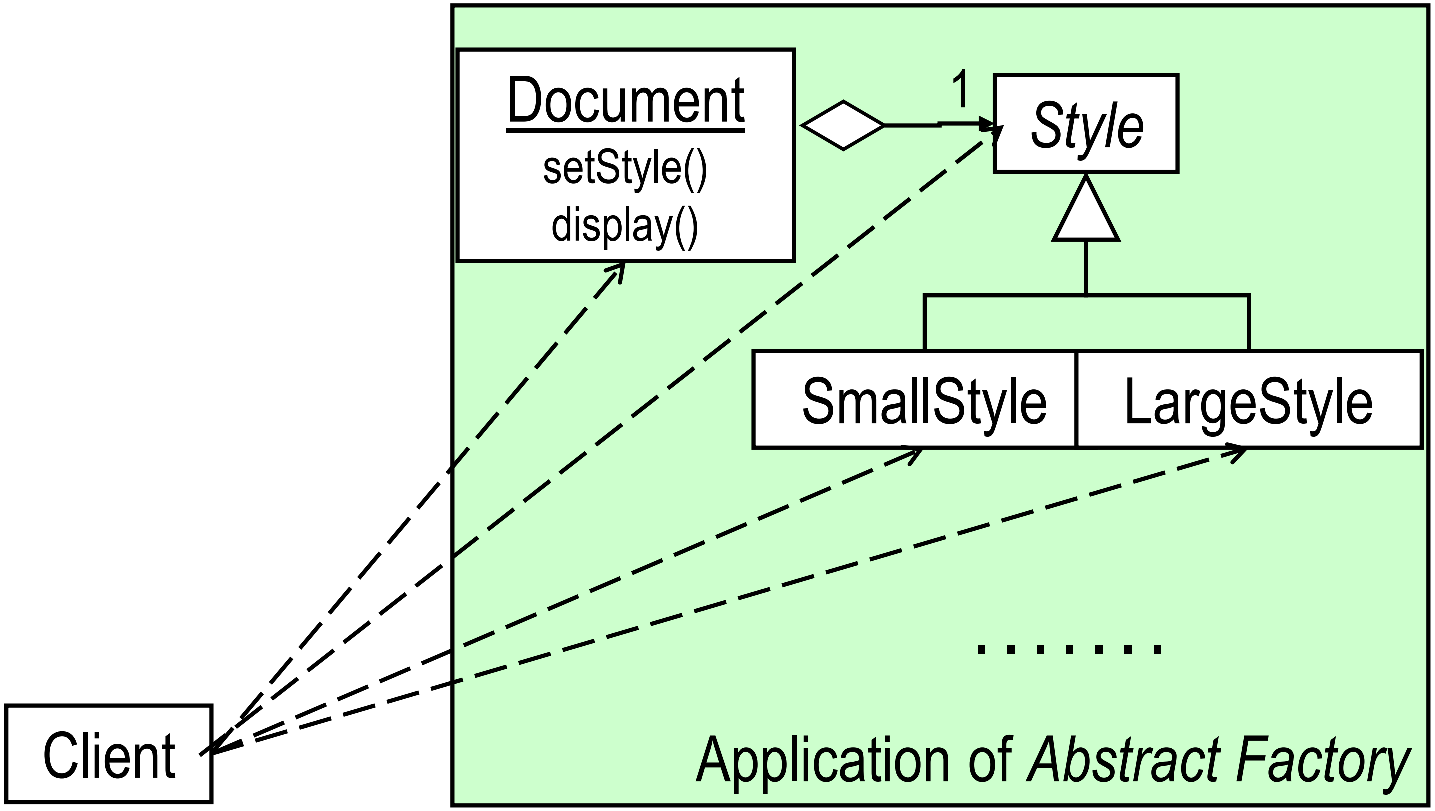
Style….

*Abstract Factory**

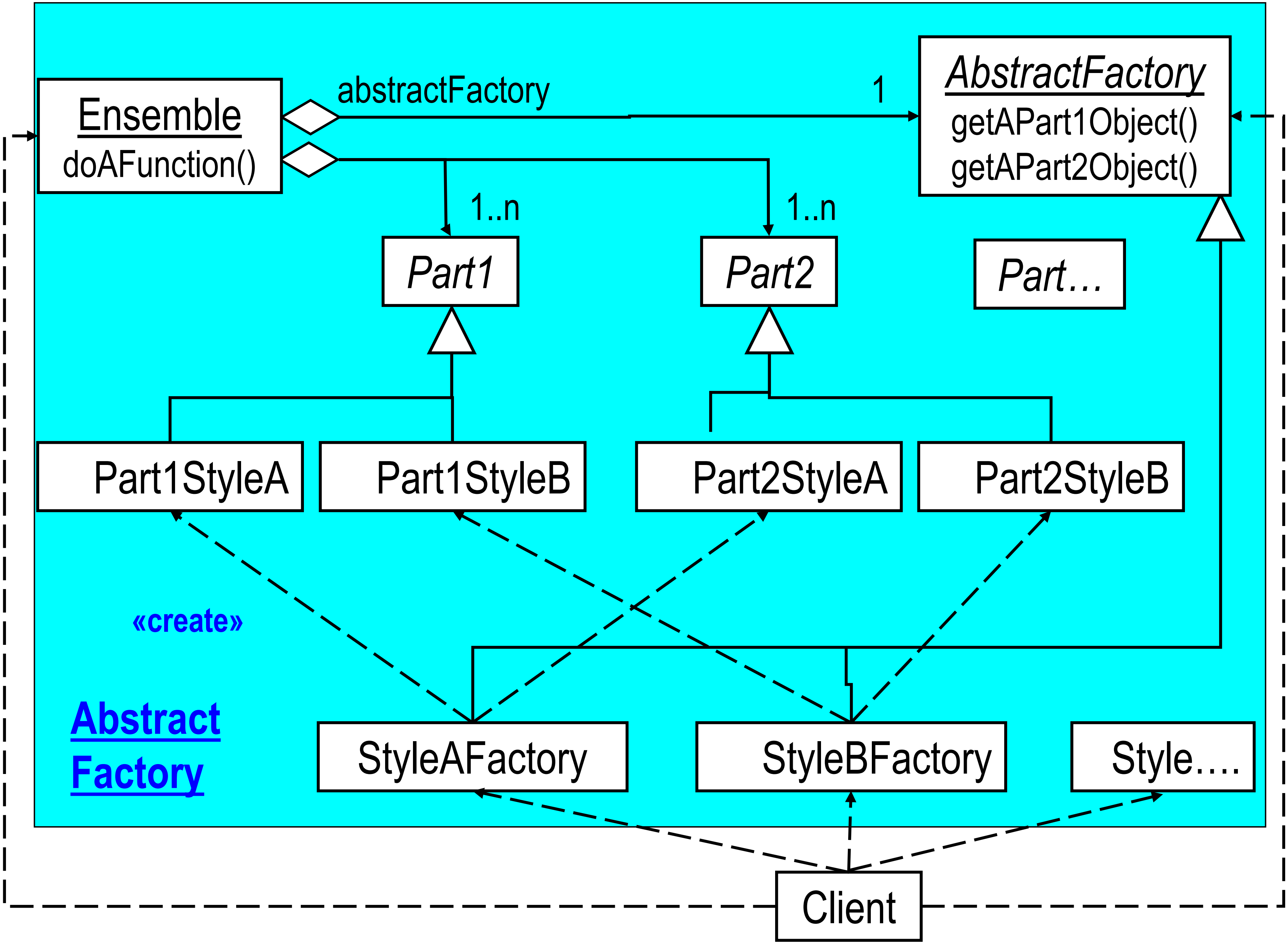
* relationships within pattern application not shown

Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Interface of *Abstract Factory* Applied to Word Processor

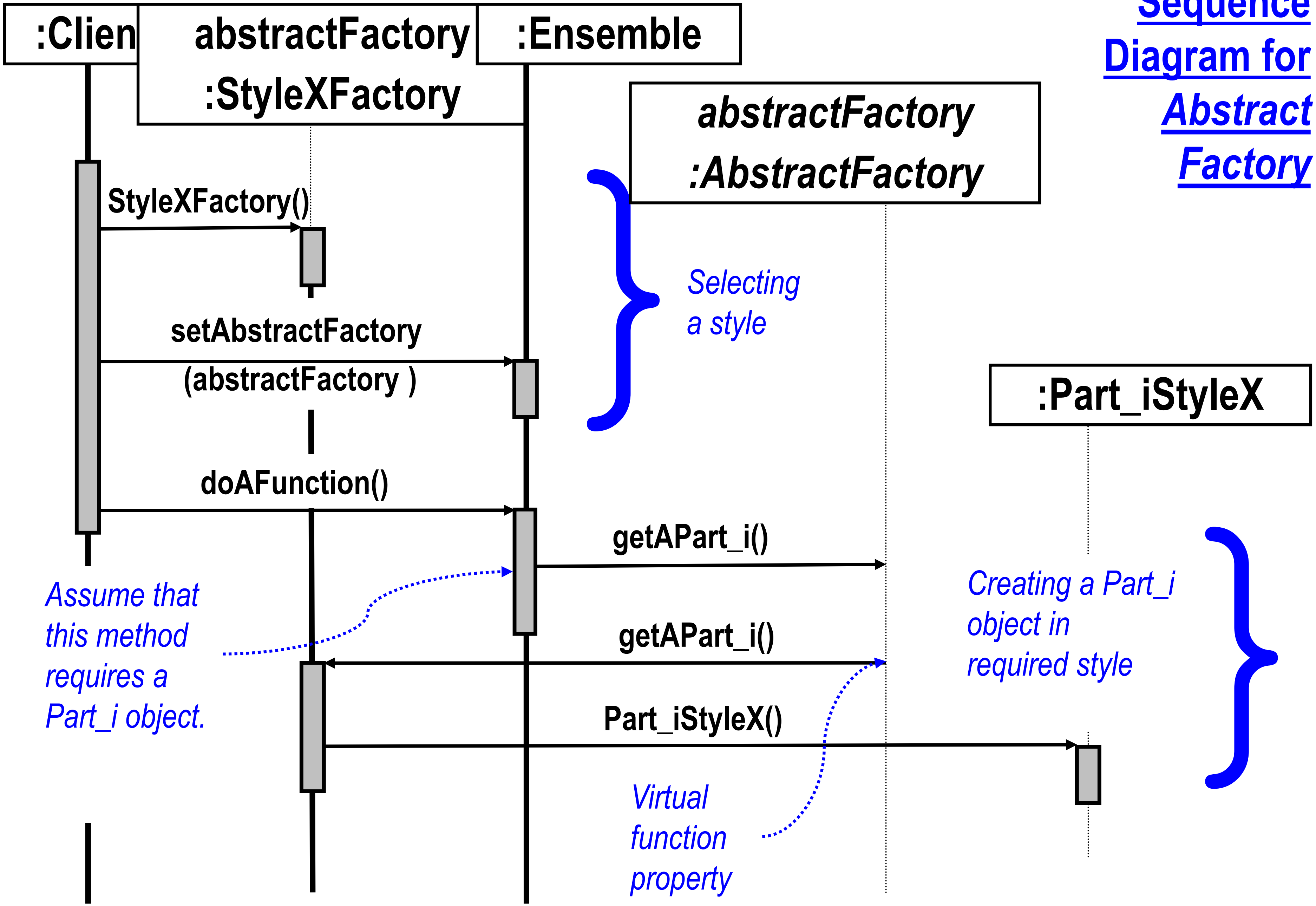


Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.



Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Sequence Diagram for *Abstract Factory*


:Clien
abstractFactory :StyleXFactory
:Ensemble
abstractFactory :AbstractFactory
:Part_iStyleX
StyleXFactory()
setAbstractFactory (abstractFactory)
doAFunction()
getAPart_i()
getAPart_i()
Part_iStyleX()

Selecting a style

Assume that this method requires a Part_i object.

Creating a Part_i object in required style

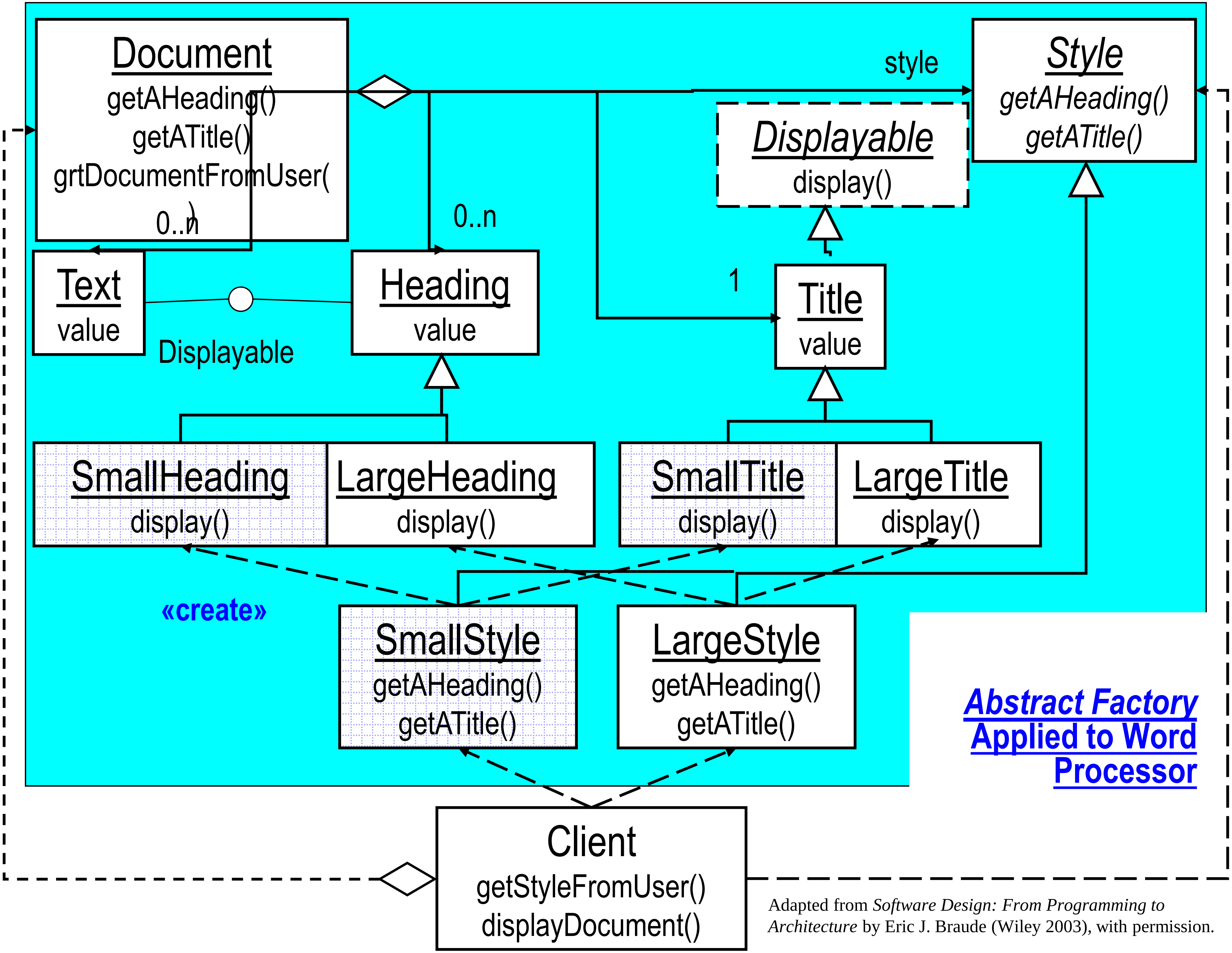
Virtual function property

Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Design Goals At Work: → *Correctness and Reusability* ←

We want to separate the code parts that format the document in each style. We also want to separate the common document generation code. This facilitates reusing parts and checking for correctness.

Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.



Document
getAHeading()
getATitle()
grtDocumentFromUser()
0..n

style

Style
getAHeading()
getATitle()

Displayable
display()

0..n

1

Text
value

Displayable

Heading
value

Title
value

SmallHeading
display()

LargeHeading
display()

SmallTitle
display()

LargeTitle
display()

«create»

SmallStyle
getAHeading()
getATitle()

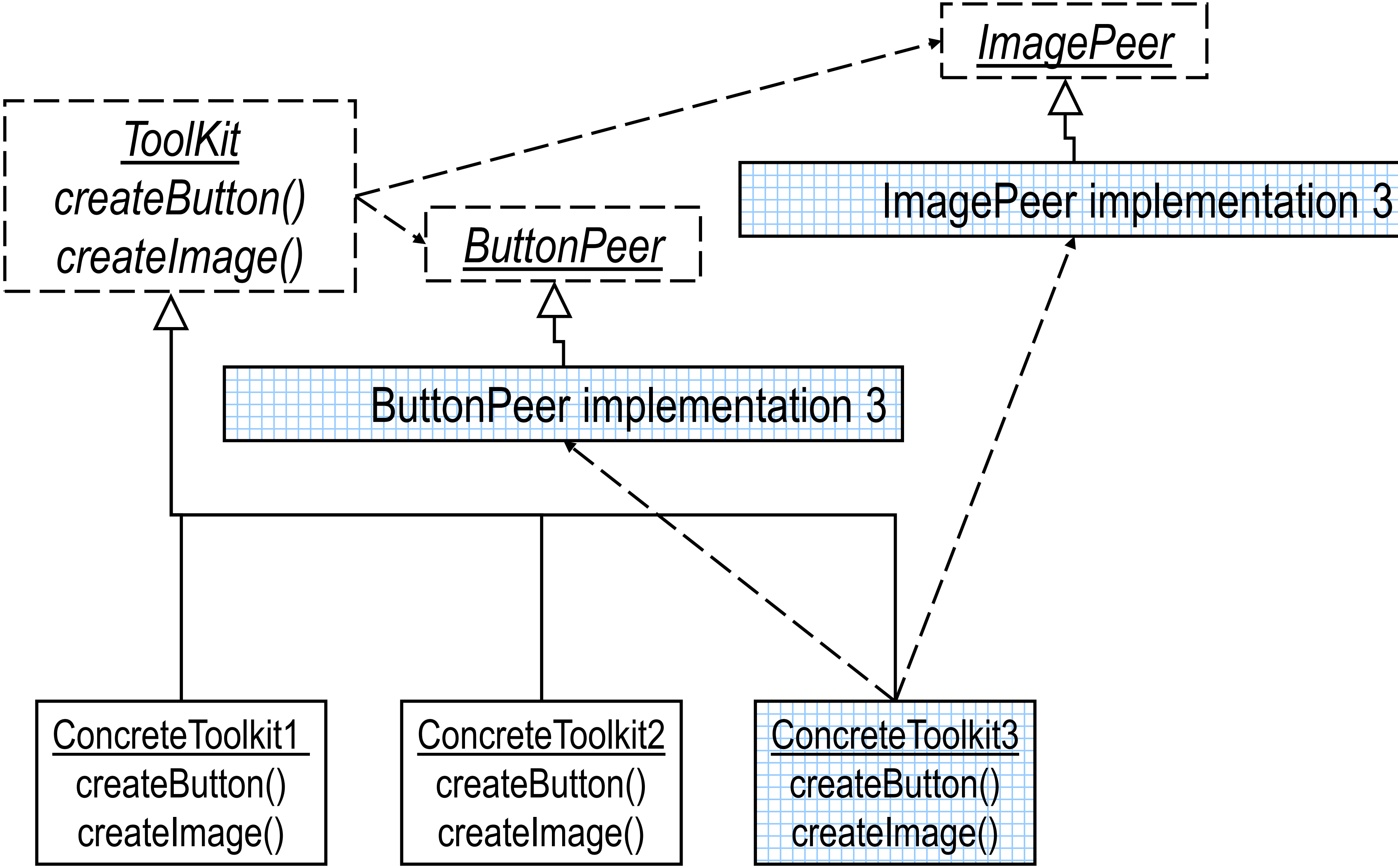
LargeStyle
getAHeading()
getATitle()

Client
getStyleFromUser()
displayDocument()

Abstract Factory **Applied to Word Processor**

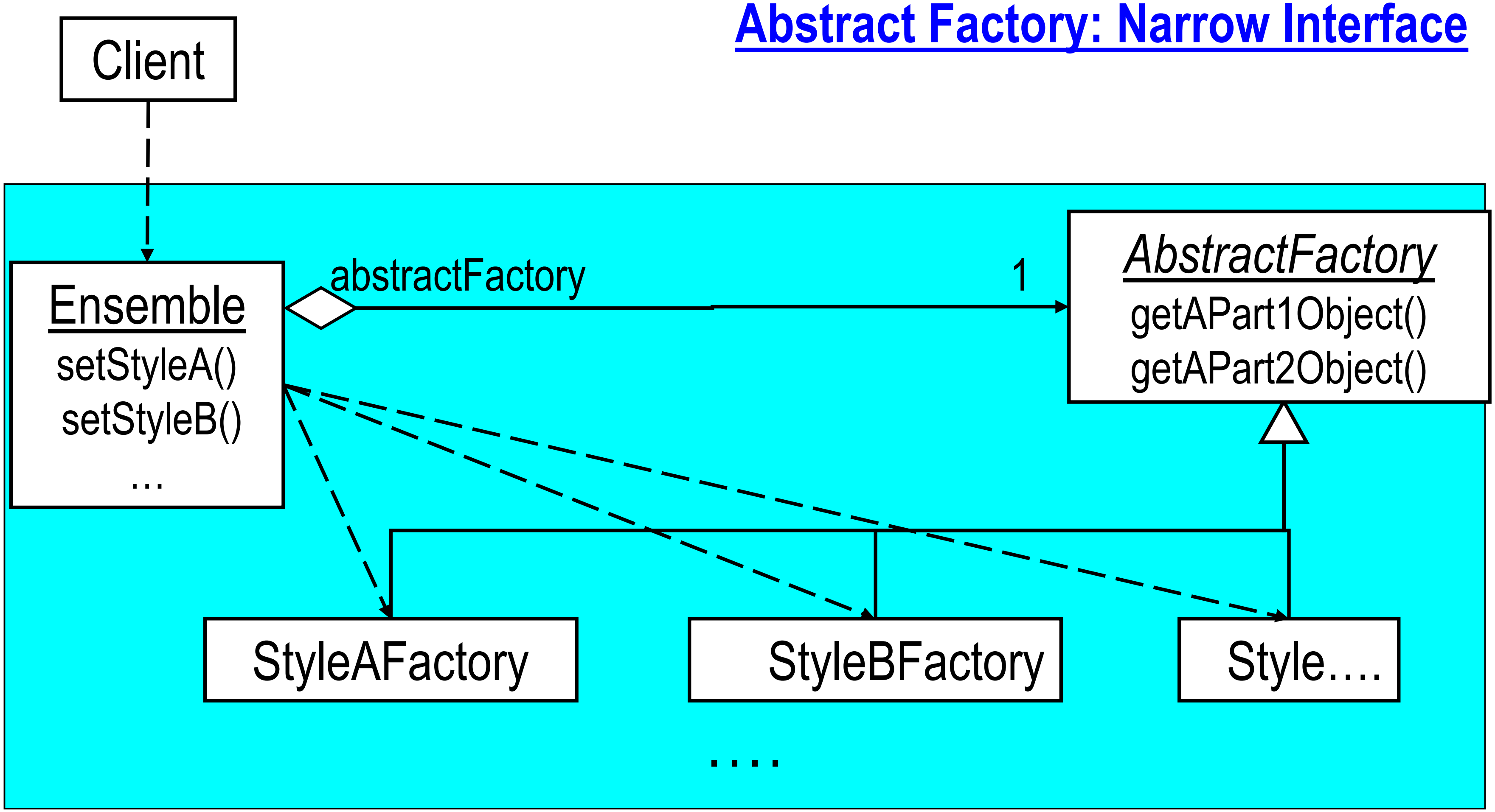
Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

An Abstract Factory Application: Java *ToolKit*



Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Abstract Factory: Narrow Interface



Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Key Concept: → Abstract Factory Design Pattern ←

To design an application in which there are several possible styles for a collection of objects, capture styles as classes with coordinated factory methods.

Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

7.4 - Prototype Design Pattern

Prototype

Design Purpose

Create a set of almost identical objects whose type is determined at runtime.

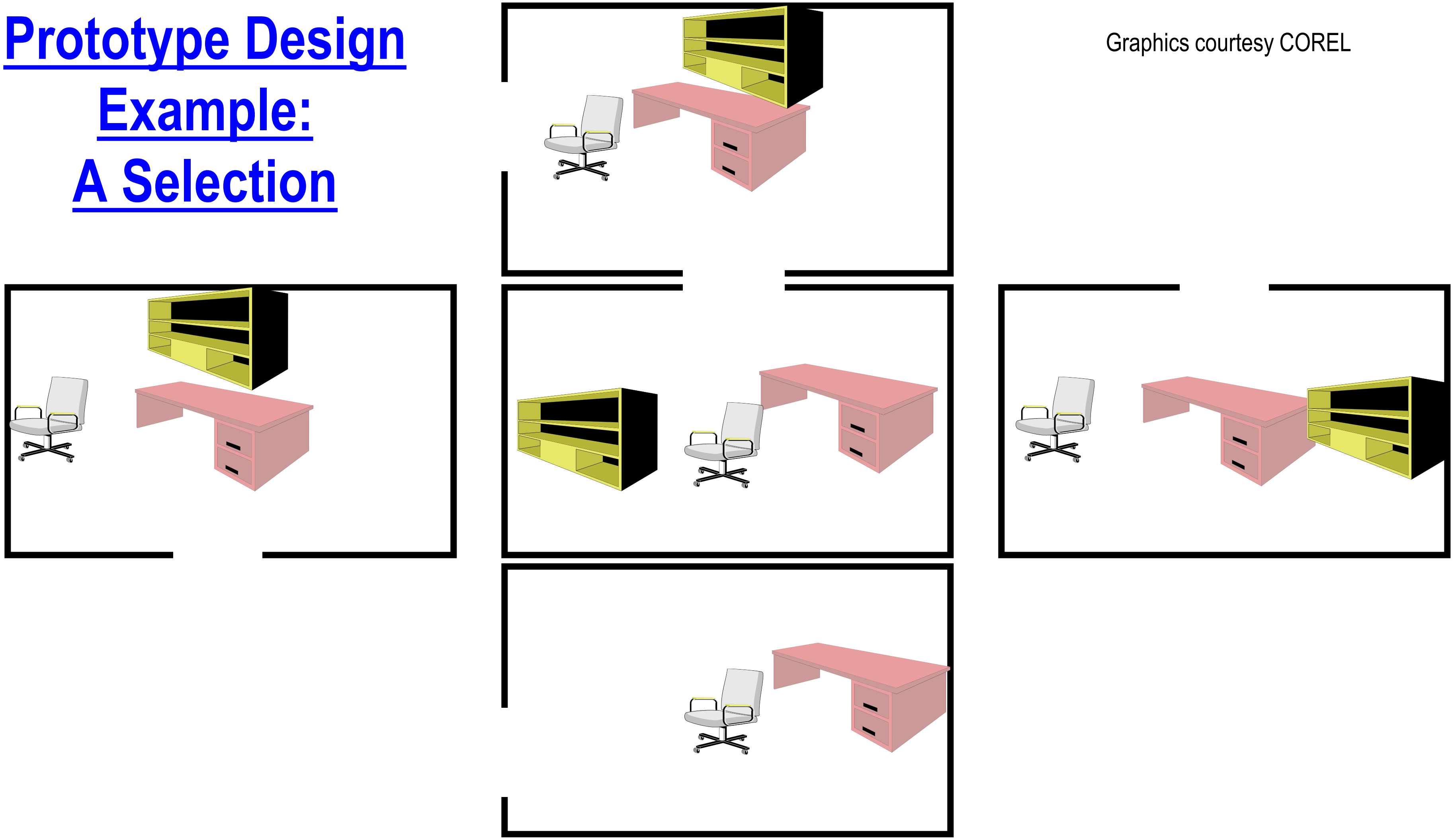
Design Pattern

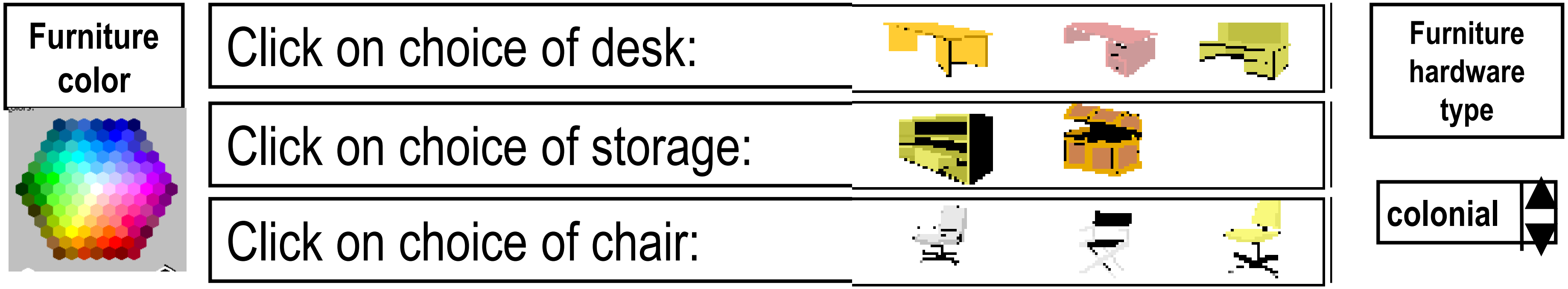
Assume that a prototype instance is known; clone it whenever a new instance is needed.

Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Prototype Design Example: A Selection

Graphics courtesy COREL





Furniture color

Click on choice of desk:

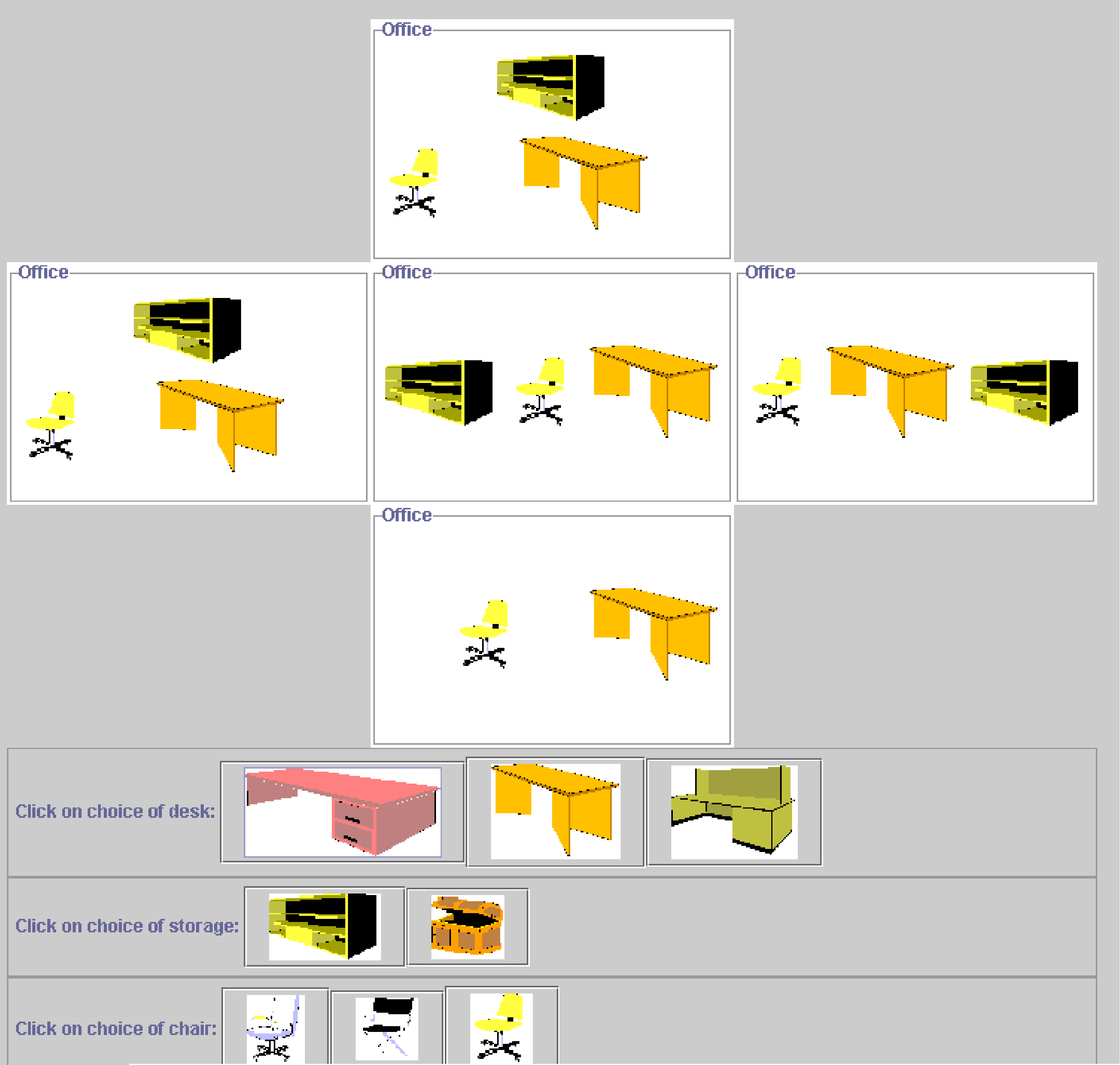
Click on choice of storage:

Click on choice of chair:

Furniture hardware type

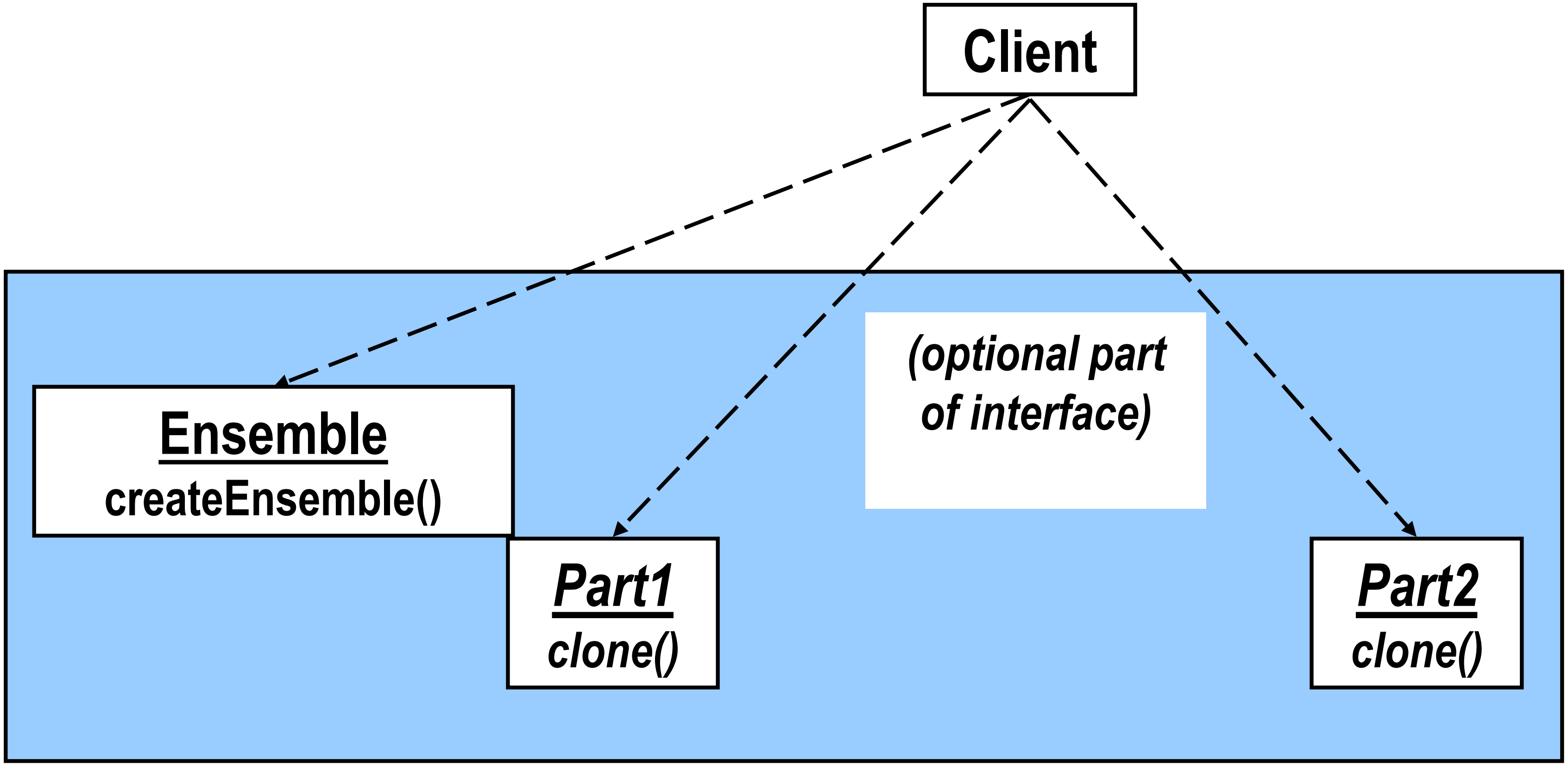
colonial ▲▼

A Simplified Prototype Example



Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Prototype Interface With Clients

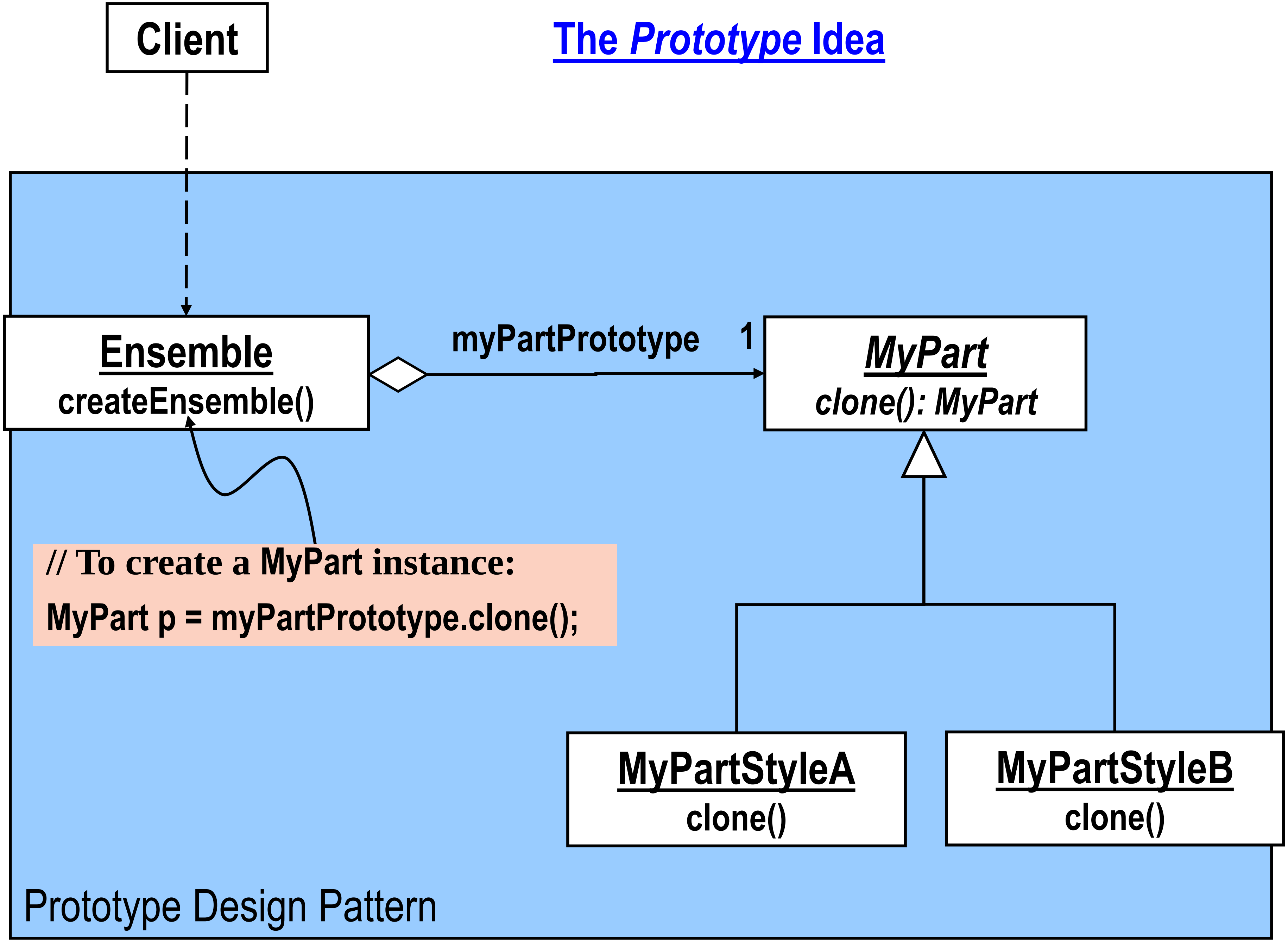


Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Code Example

```
OfficeSuite myOfficeSuite =
   OfficeSuite.createOfficeSuite( myDesk, myChair, myStorage);

myGUI.add(myOfficeSuite);
myOfficeSuite.setBackground(“pink”);
```

The *Prototype* Idea



```
// To create a MyPart instance:
MyPart p = myPartPrototype.clone();
```

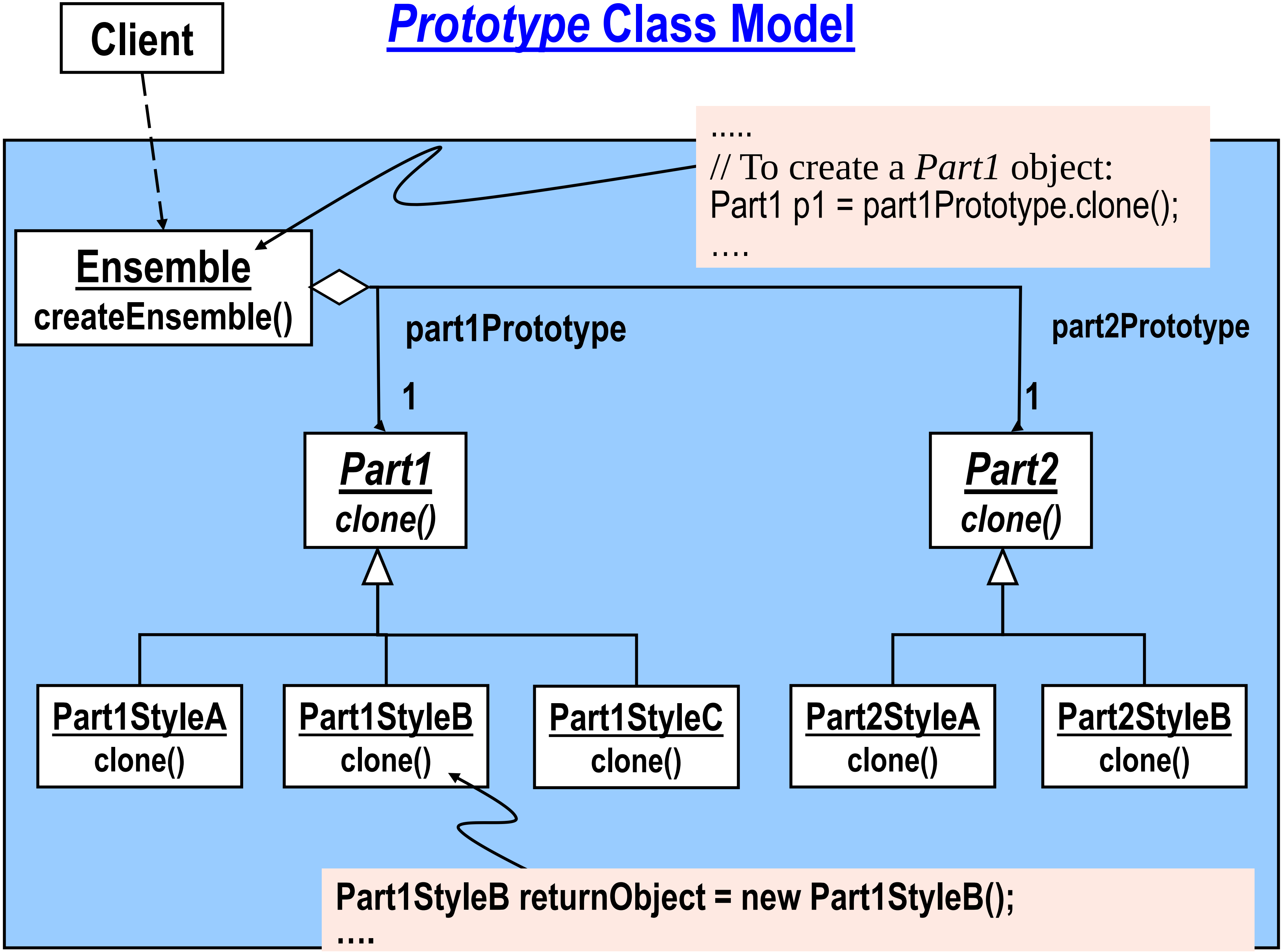
Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Code Example

```
Ensemble EnsembleA Ensemble.createEnsemble(. . .);

Ensemble EnsembleB Ensemble.createEnsemble();
```

```
// This code is inside the Ensemble class
MyPart anotherMyPart = MyPartPrototype.clone();
MyPart yetAnotherMyPart = MyPartPrototype.clone();
```

Prototype Class Model

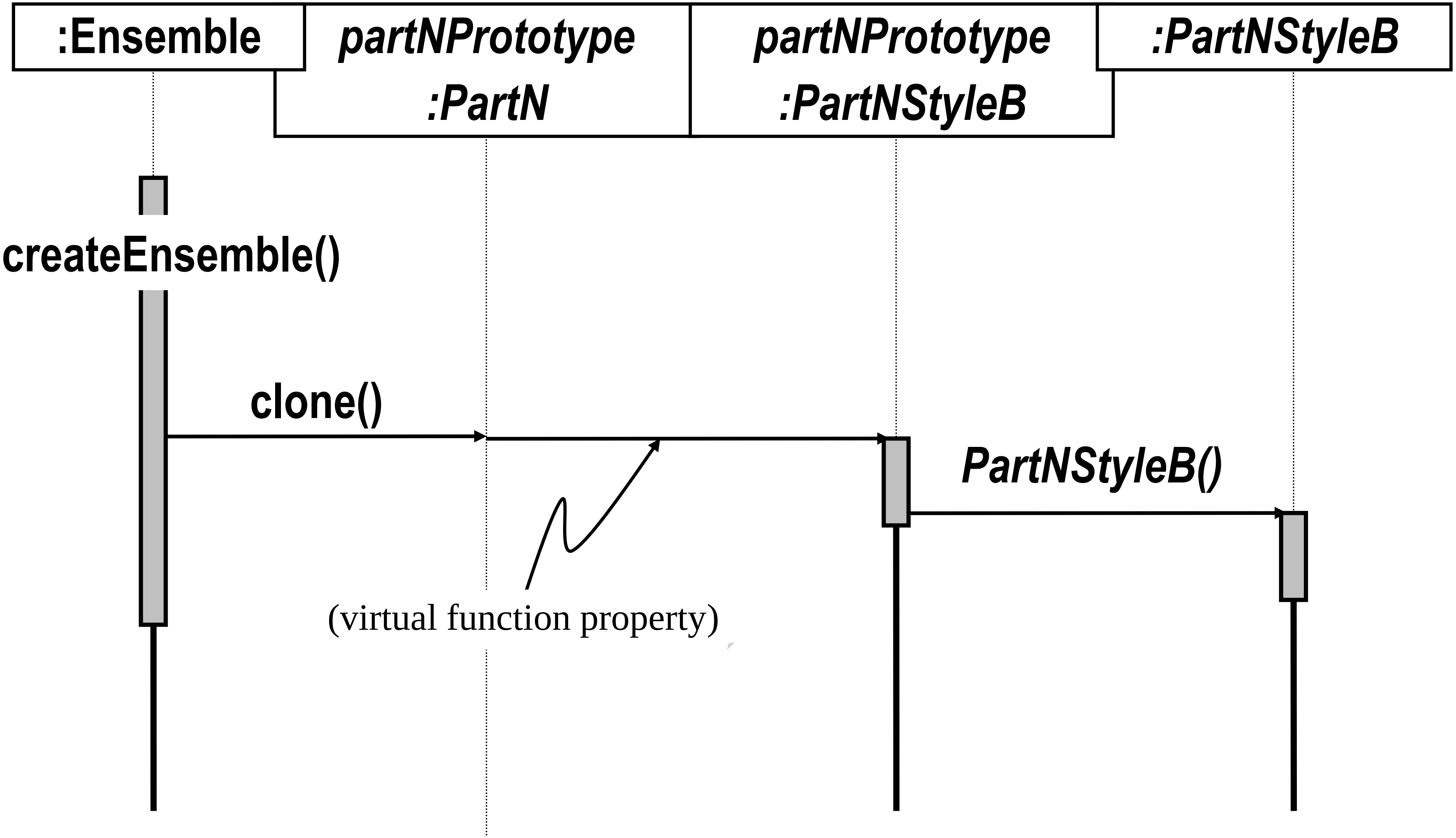


```
.....
// To create a Part1 object:
Part1 p1 = part1Prototype.clone();
….
```

```
Part1StyleB returnObject = new Part1StyleB();
….
```

Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Sequence Diagram for *Prototype*



Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Contrasting Abstract Factory and Prototype

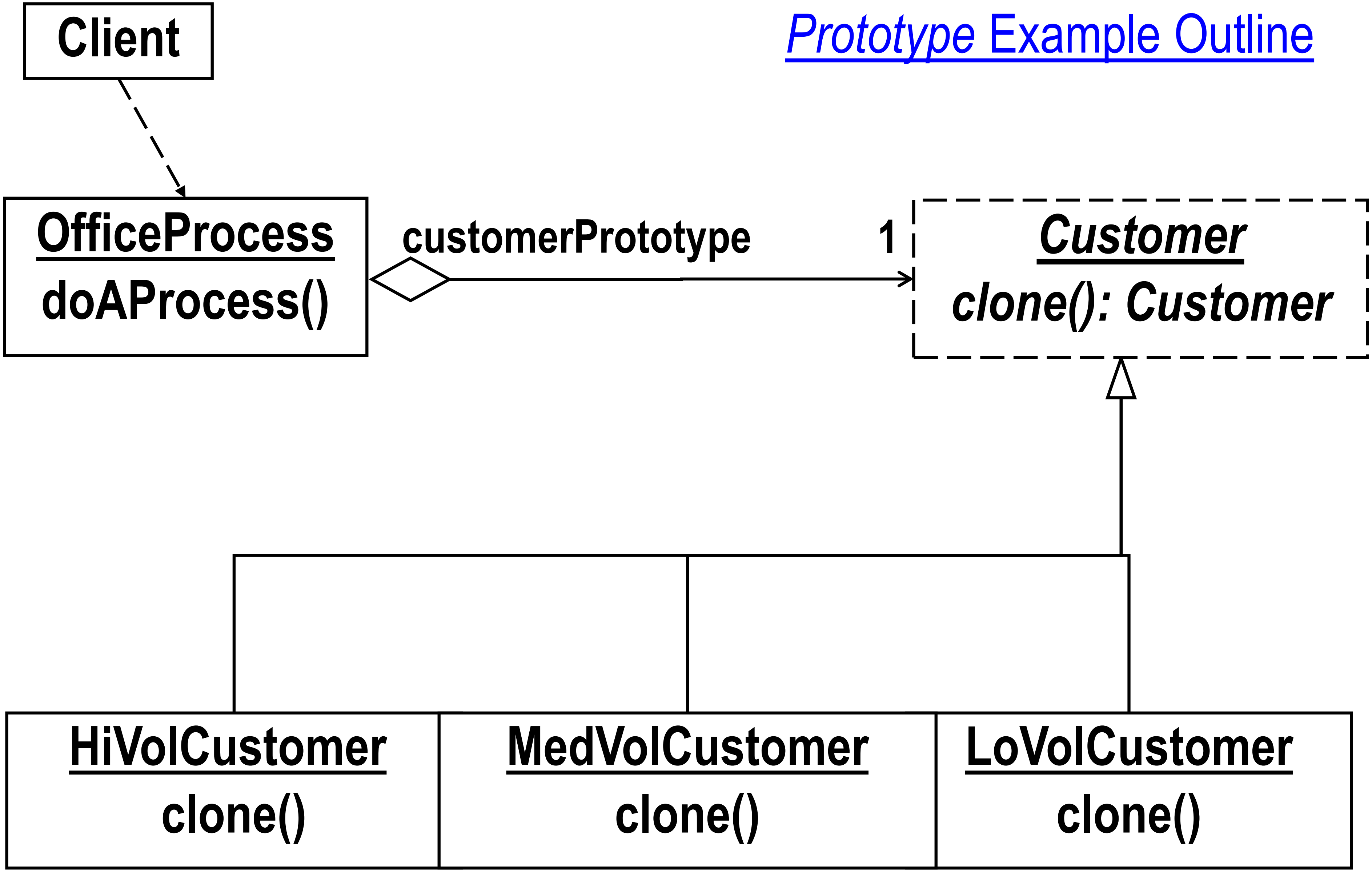
- Prototype allows the client to select any chair style, any desk style, and any cabinet style
- This is all done separately rather than have to select an overall office style
- Nevertheless, the client wants to keep a single style of chair and a single style of desk throughout the office suite

Design Goals At Work: → *Correctness and Reusability* ←

We want to isolate the parts pertaining to each type of customer. We also want to isolate the common customer code. This makes it easier to check the design and implementation for correctness, and to reuse the parts.

Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Prototype Example Outline



Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Requirement for (Deep) Cloning

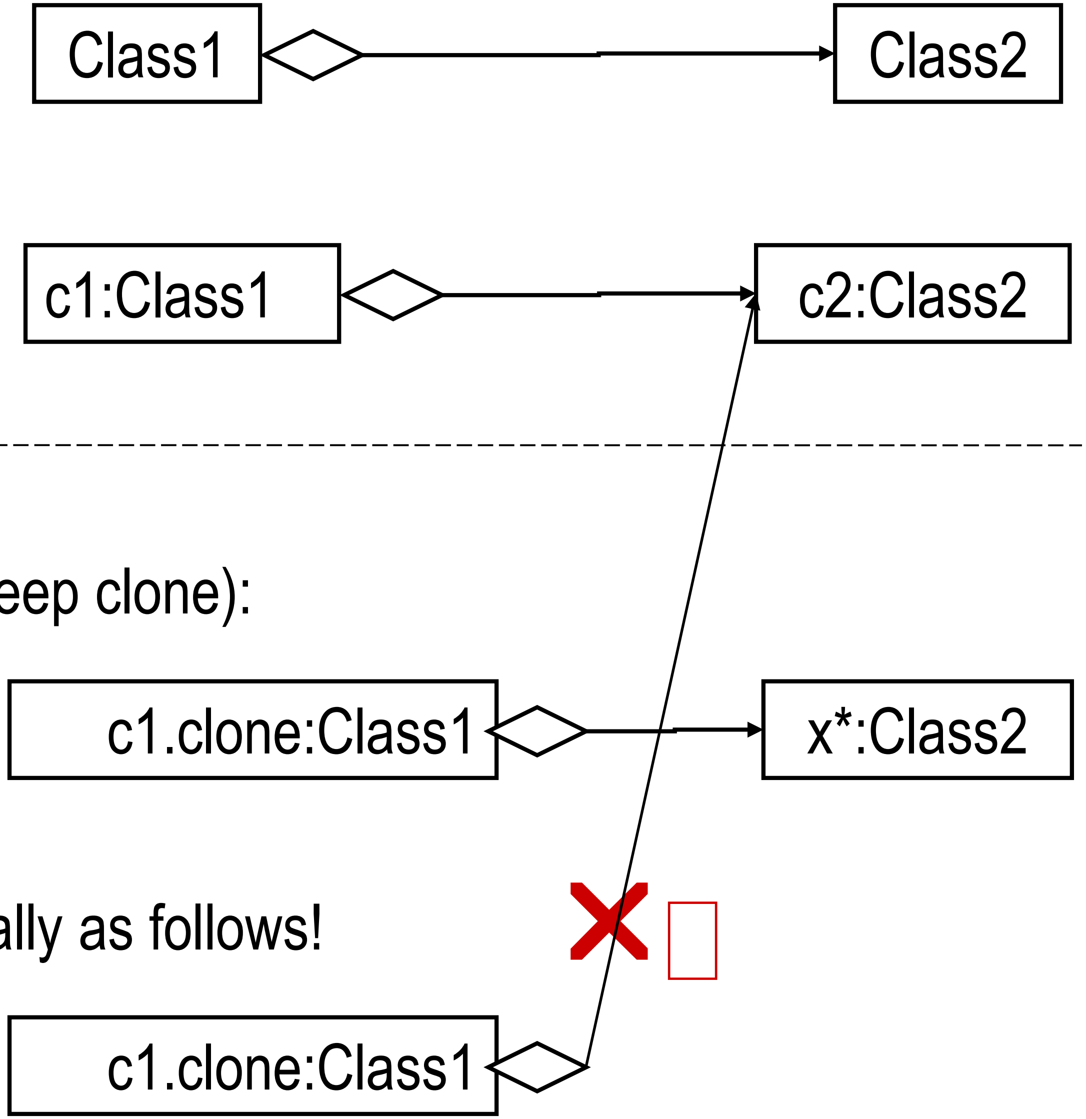
Given:

(1) Class model:

(2) *c1* an instance of *Class1*:

c1.clone should be as follows (deep clone):

In shallow cloning, *c1.clone* actually as follows!



* a clone of c2

Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Key Concept: → Prototype Pattern ←

-- when designing for multiple instances which are the same in key respects, create them by cloning a prototype.

Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Summary of *Creational Design Patterns*

Use *Creational Design Patterns* when creating complex objects

- *Factory* when creating individuals
- *Singleton* for exactly one individual
- *Abstract Factory* when creating families
- *Prototype* to “mix & match”

Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.