Multiple-Choice Questions:

1. T or F? Coupling and cohesion are closely linked in that as one increases, so does the other.

    a. True

    b. False

2. Given classes *A* and *B*, which of the following is <u>not</u> a common type of coupling in object-oriented software?

    a. *A* is a direct or an indirect subclass of *B*

    b. A method parameter or local variable in *A* references *B*

    c. *A* has an instance variable that refers to *B*

    d. *A* invokes methods of *B*

    e. None of the above

3. What is the typical relationship between coupling and cohesion?

    a. There is no relationship between coupling and cohesion.

    b. As cohesion increases, coupling increases.

    c. As cohesion increases, coupling decreases.

4. [2pts] All else being equal, which is more desirable?

    a. Higher cohesion and higher coupling

    b. Higher cohesion and lower coupling

    c. Lower cohesion and lower coupling

    d. Lower cohesion and higher coupling

    e. None of the above is more desirable than the others.

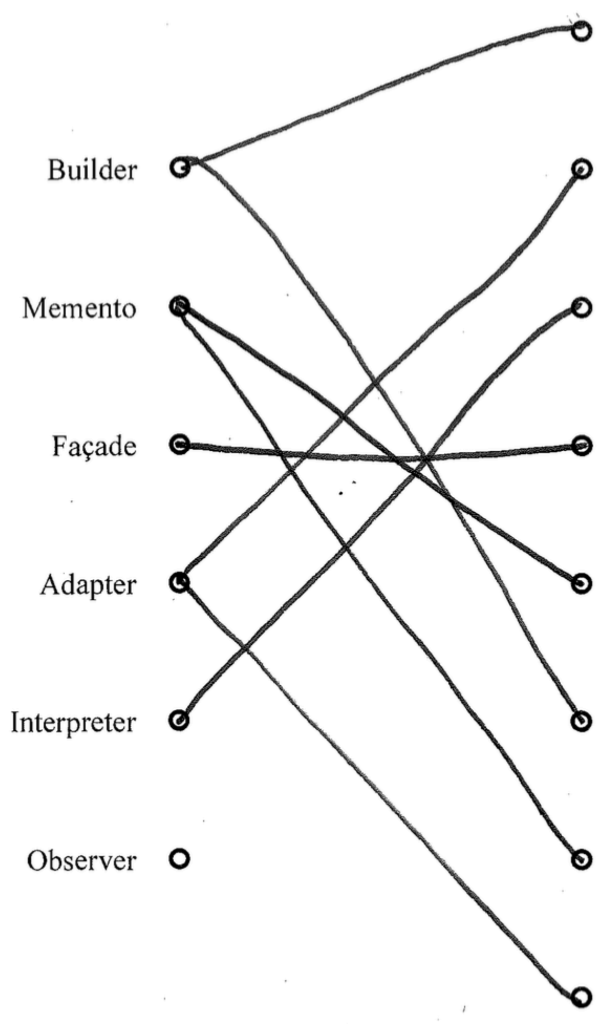
Solutions:

1. b (as coupling increases, cohesion decreases)
2. e
3. c
4. b

Problem: Match the design pattern to the situation to which you should apply it.

	○ Your application needs to generate HTML files (from scratch).
Builder ○	○ Your program must support switching among several different email libraries, but each one has a slightly different interface.
Memento ○	○ You want to let users create and run macros inside your application.
Façade ○	○ Sending an SMS message requires lots of big, ugly code, involving connection, message, and other objects.
Adapter ○	○ You want your application to save its state so that if it crashes, then it can auto-recover.
Interpreter ○	○ Your program has to create and configure some big, ugly record objects before inserting them into a database.
Observer ○	○ Your program has to support replication. You need a way for the program to save its state so the program can be copied to other servers.
	○ Your company already implemented a component that almost implements the interface that you need, but not quite.

Solution:



Your application needs to generate HTML files (from scratch).

Your program must support switching among several different email libraries, but each one has a slightly different interface.

You want to let users create and run macros inside your application.

Sending an SMS message requires lots of big, ugly code, involving connection, message, and other objects.

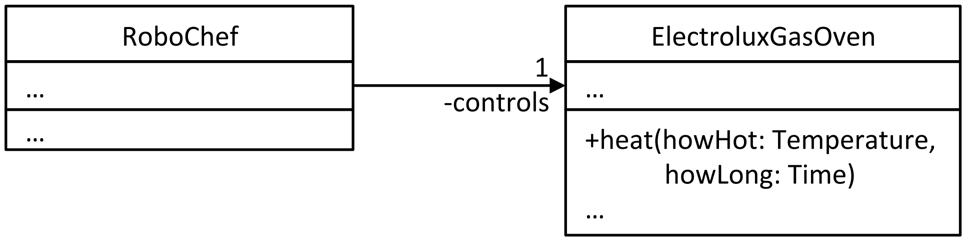
You want your application to save its state so that if it crashes, then it can auto-recover.

Your program has to create and configure some big, ugly record objects before inserting them into a database.

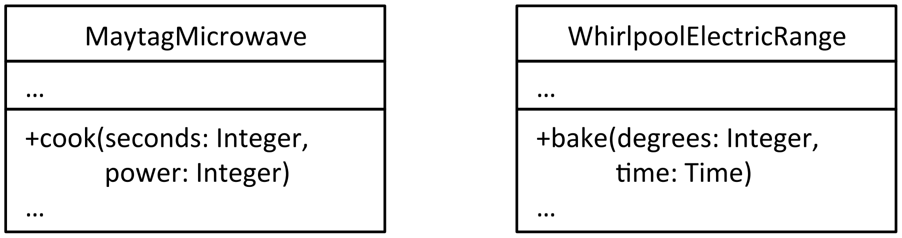
Your program has to support replication. You need a way for the program to save its state so the program can be copied to other servers.

Your company already implemented a component that almost implements the interface that you need, but not quite.

Problem: Imagine that you are the creator of an "intelligent" kitchen system, RoboChef, that can actually control different kitchen appliances (e.g., ovens, choppers) to prepare food. Initially, you implemented RoboChef to use only Electrolux gas ovens. Here is an excerpt of your current software design:



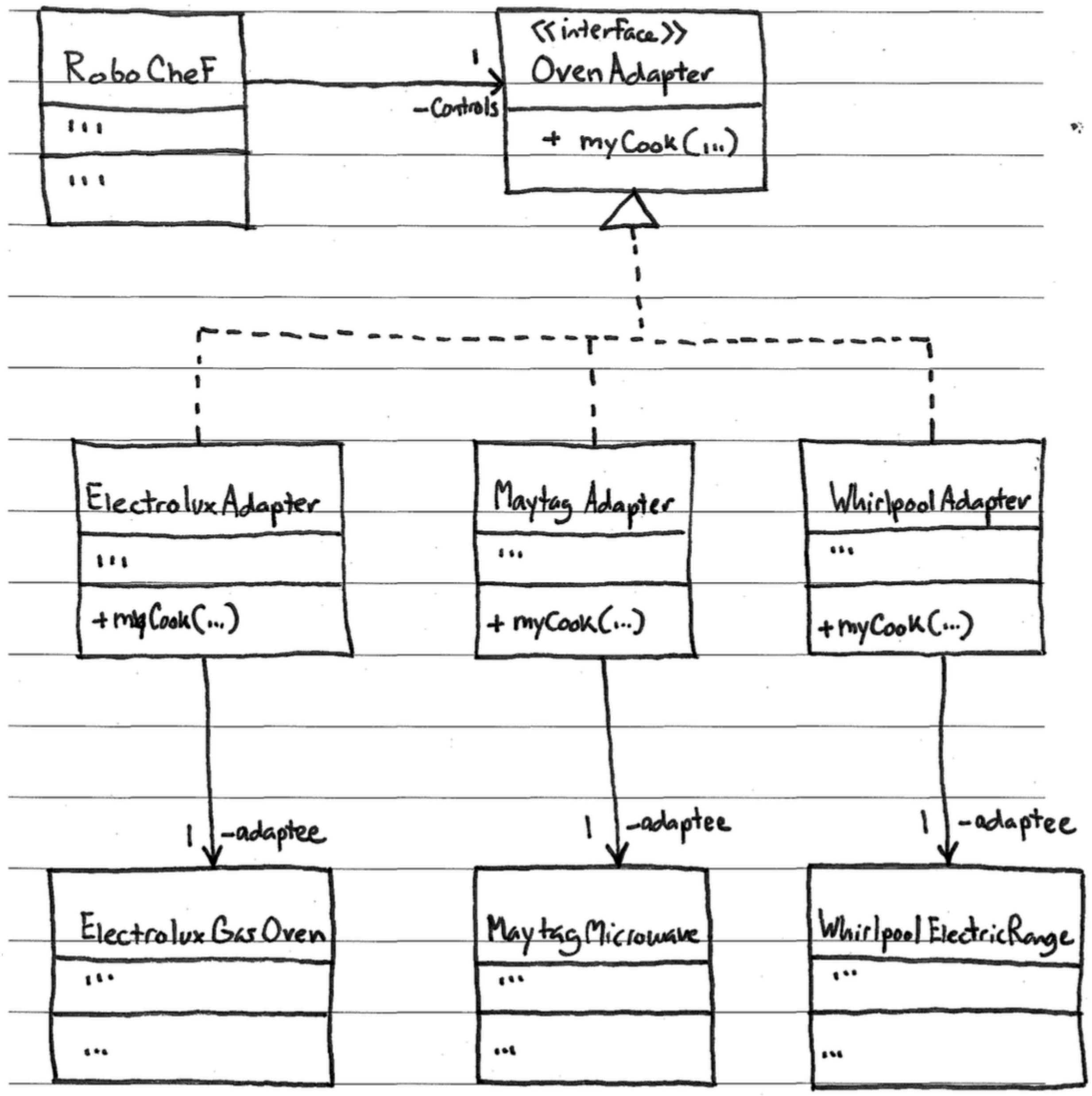
Note that the Electrolux Company provided the software interface for controlling the gas oven (ElectroluxGasOven), and you created the intelligent decision-making part (RoboChef). As your next step, you would like your system to support different types of ovens other than Electolux gas ones. For example, Maytag and Whirlpool each provide their own software interfaces for their ovens:


MaytagMicrowave
…
+cook(seconds: Integer, power: Integer)
…
WhirlpoolElectricRange
…
+bake(degrees: Integer, time: Time)
…

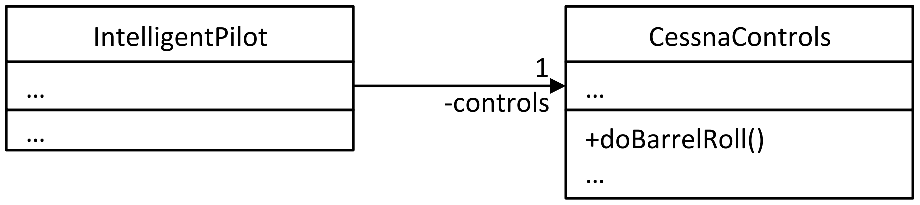
Update your current software design to allow easy switching between oven-control systems. Your design must apply the **adapter pattern**.

Draw a class diagram for your design.

Solution:

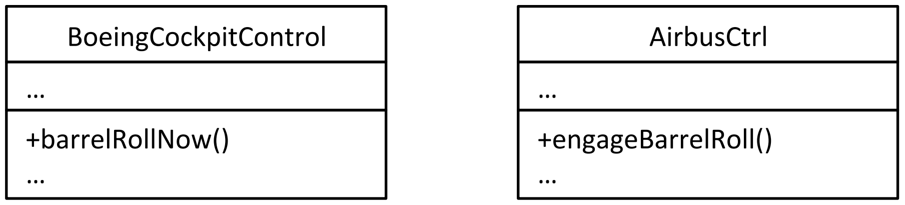

RoboChef
...
...
1
-Controls
<<interface>>
OvenAdapter
+ myCook(...)
ElectroluxAdapter
...
+ myCook(...)
Maytag Adapter
...
+ myCook(...)
WhirlpoolAdapter
...
+ myCook(...)
1 -adaptee
1 -adaptee
1 -adaptee
ElectroluxGasOven
...
...
MaytagMicrowave
...
...
WhirlpoolElectricRange
...
...

Problem: Imagine that you are the creator of an "intelligent" autopilot system that can actually fly and land real airplanes (wow!). Initially, you implemented your system to fly small Cessna airplanes. Here is an excerpt of your current software design:



Note that the Cessna Aircraft Company provided the software interface for controlling the plane (CessnaControls), and you created the intelligent decision-making part (IntelligentPilot).

As your next step, you would like your system to support different types of airplanes other than Cessnas. For example, Boeing and Airbus each provide their own software control interfaces:



Update your current software design to allow easy switching between control systems. Your design must apply the **adapter pattern**.

Draw a class diagram for your design.

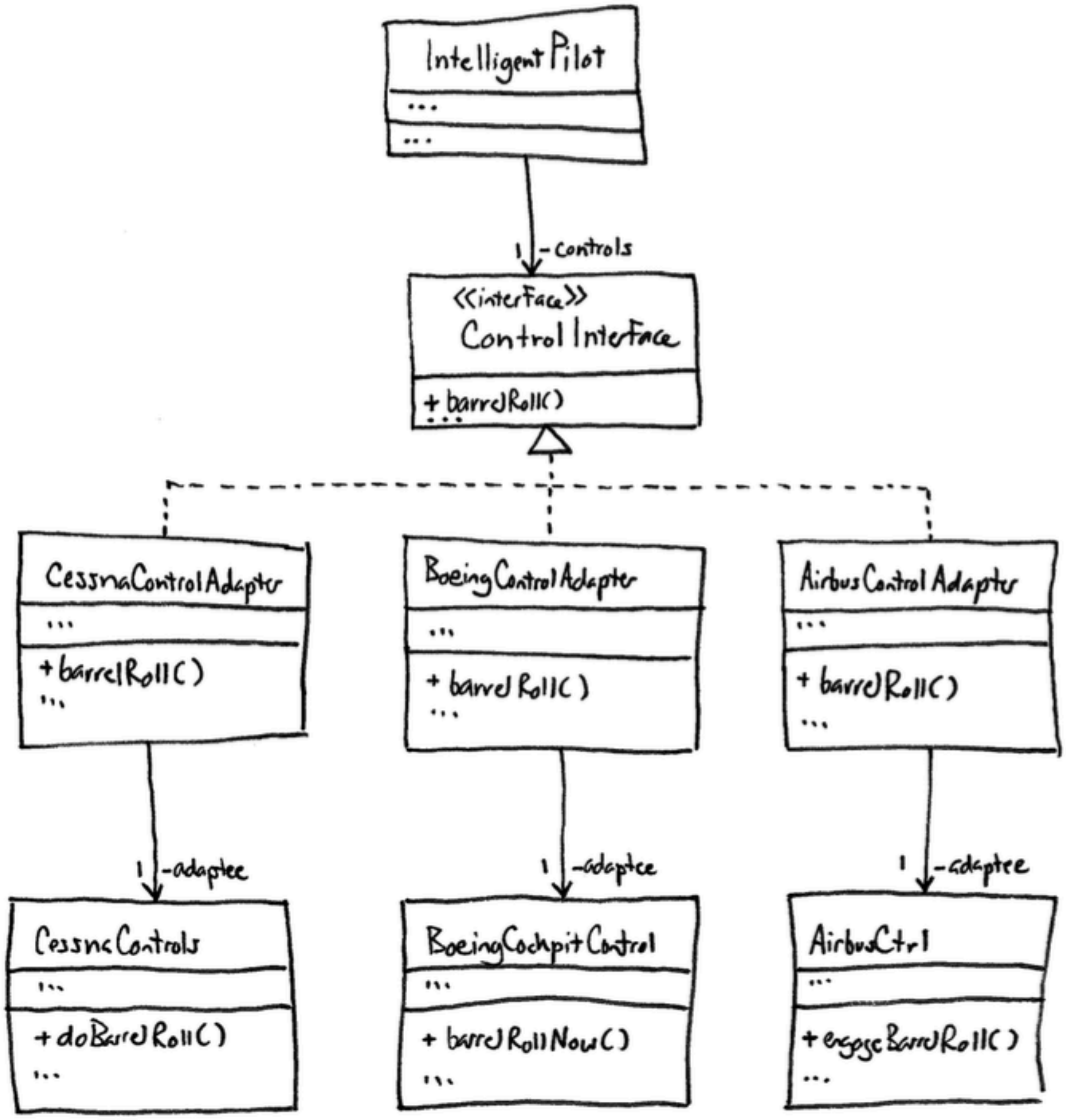
What effect did your new design have on the coupling between class IntelligentPilot and class CessnaControls.

a. Reduced their coupling

b. Increased their coupling

c. Had no effect on their coupling

Solution:

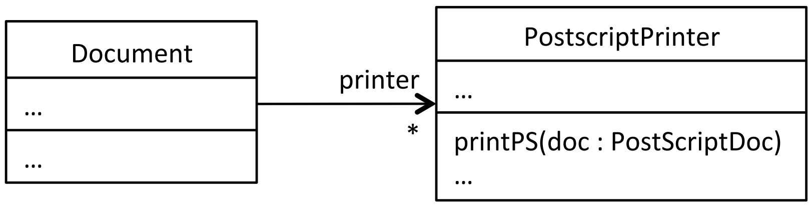


[✓ a.] Reduced their coupling

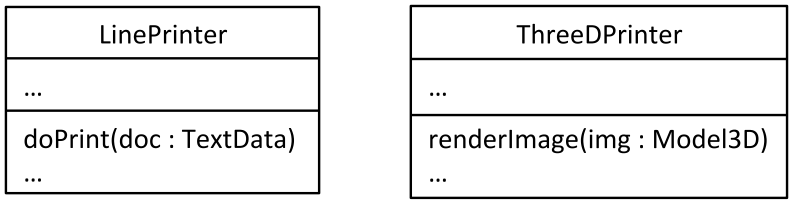
b. Increased their coupling

c. Had no effect on their coupling

Problem: Consider the following design for a document-editing system. The Document class represents a document, and Document objects know how to print themselves using a PostscriptPrinter object.



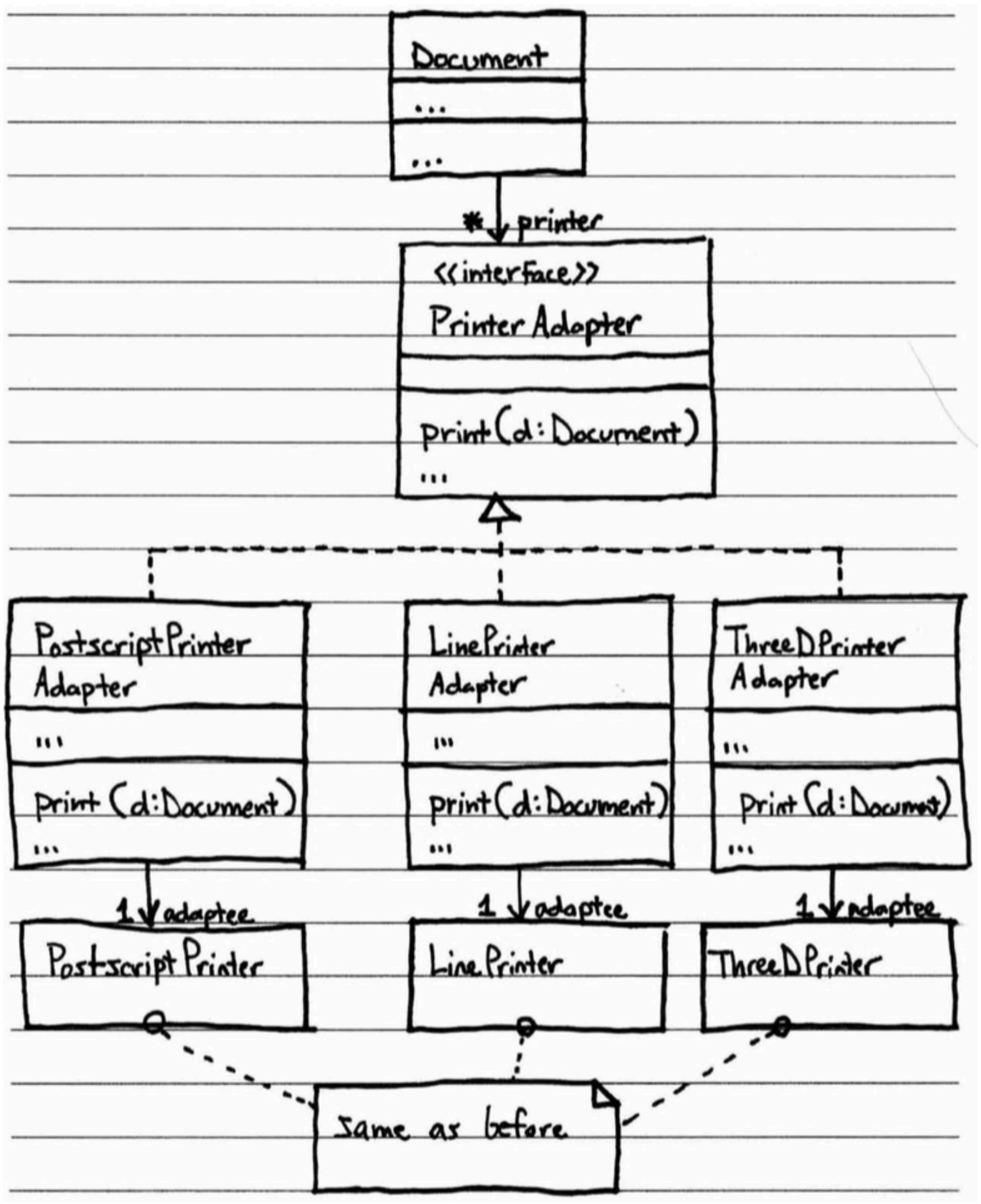
However, there are other types of printers that a document might want to print itself on, but these printers have slightly different interfaces than the Postscript printer, for example:



Using the **adapter pattern**, refactor the design, so that the different types of printers can be easily swapped in and out.

Draw a design class diagram for your design.

Solution:


Document
...
...
* printer
<<interface>>
PrinterAdapter
print(d:Document)
...
PostscriptPrinter Adapter
...
print (d:Document)
...
LinePrinter Adapter
...
print(d:Document)
...
ThreeDPrinter Adapter
...
print (d:Document)
...
1 adaptee
PostscriptPrinter
1 adaptee
LinePrinter
1 adaptee
ThreeDPrinter
Same as before