Multiple-Choice Questions:

1. Which of the following terms does this text best define?

   The extent to which one component depends on other components

   a. Cohesion

   b. Concern

   c. Coupling

   d. Crossover

   e. None of the above

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. All else being equal, which is more desirable?

   a. Higher/tighter coupling

   b. Lower/looser coupling

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

4. Class `Gear` has which of the following dependencies (i.e., things that if changed force a change in class `Gear`)?

```
class Gear
…
  def gear_inches
    ratio * Wheel.new(rim, tire).diameter
  end
…
end
```

   a. A class named `Wheel` must exist

   b. `Wheel.new` must take two parameters, `rim` and `tire`

   c. The first argument for `Wheel.new` must be `rim`, and the second must be `tire`

   d. All of the above

   e. None of the above

5. Which of the following is true about design patterns?

   a. Represent the best practices used by experienced object-oriented software developers

   b. Solutions to general problems that developers commonly face during software development

   c. Obtained by trial and error of numerous software developers over a substantial period of time

   d. All of the above

   e. None of the above

6. Which pattern automatically notifies dependent objects when a subject object is modified?

   a. Adapter

   b. Observer

   c. Mediator

   d. Memento

   e. None of the above

7. Which pattern encapsulates how a set of objects interact?

   a. Adapter

   b. Observer

   c. Mediator

   d. Memento

   e. None of the above

8. Which of the following are true about the Mediator Pattern? Circle all that apply.

   a. Reduces interdependencies by spreading interaction logic throughout objects

   b. Promotes loose coupling by keeping objects from referring to each other explicitly

   c. Allows you to vary the interaction between objects independently

   d. Tightly couples objects together to make them more maintainable

   e. Uses indirection to keep objects from directly referring to each other

Solutions:

1. c
2. e
3. b
4. d
5. d
6. b
7. c
8. b, c, e

Consider these figures when answering the following question.

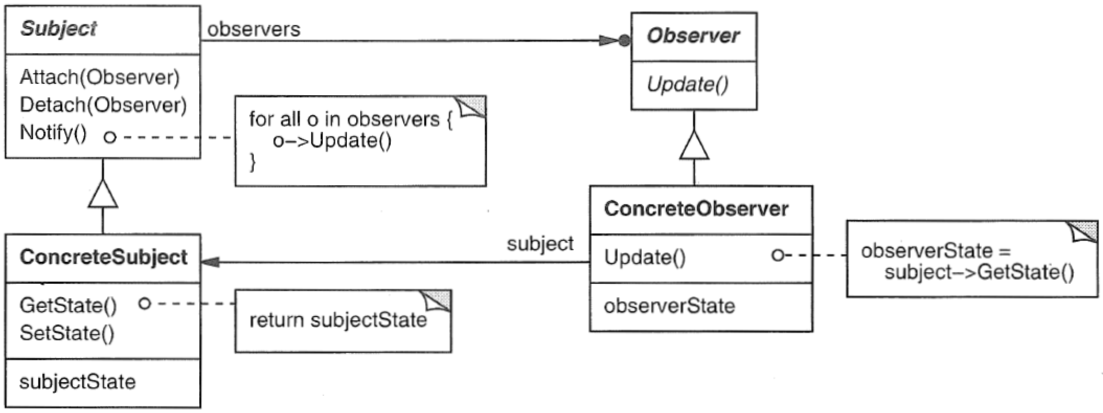


Figure 1. Observer Pattern from the "Gang of Four" book. (Note that the book uses an outdated class diagram notation.)

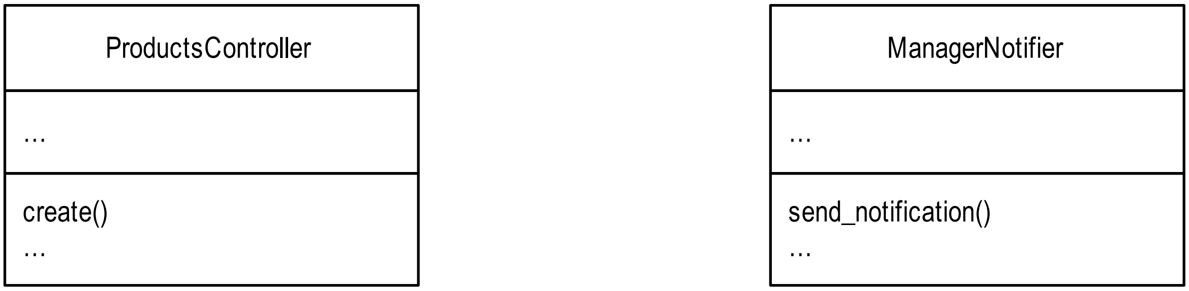
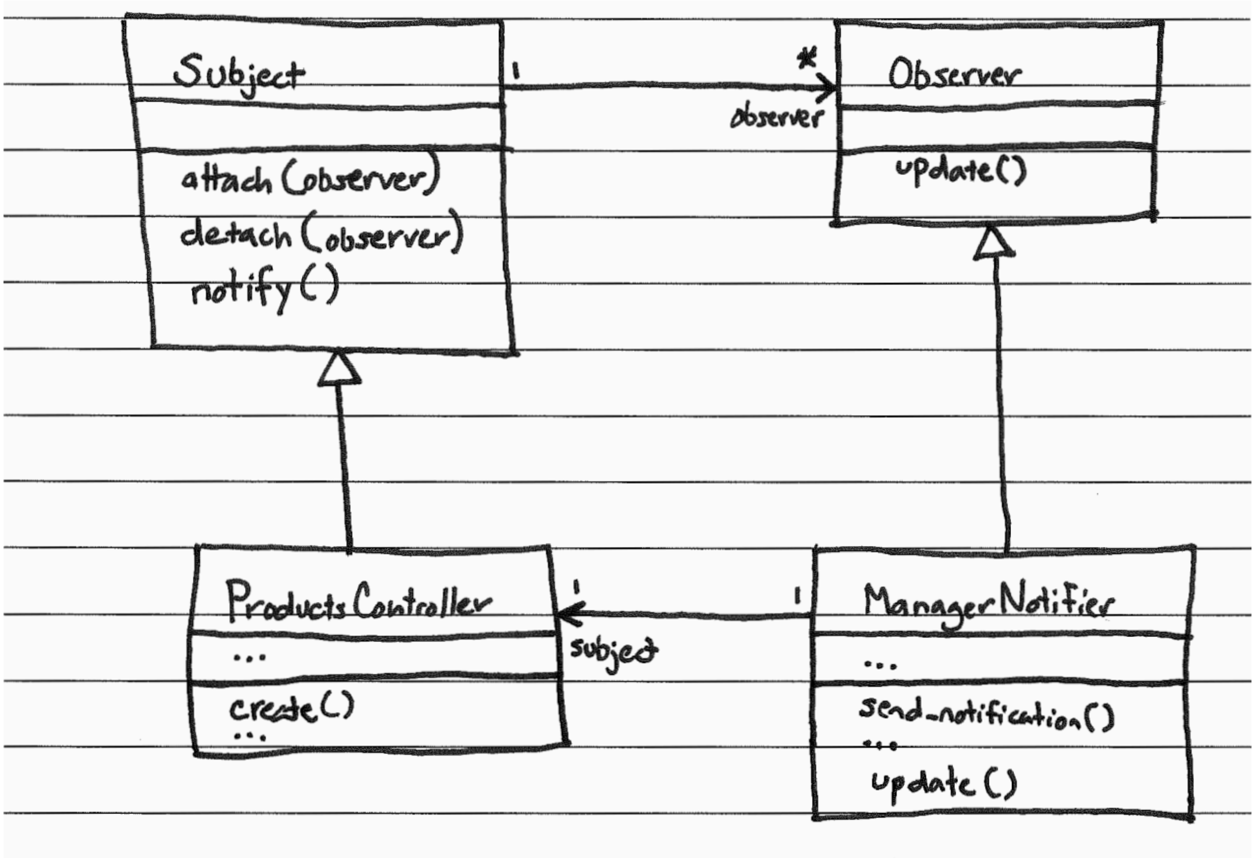


Figure 2. Classes for product-supply system.

Problem:

Recall the Observer Design Pattern depicted in Figure 1. Imagine that you are designing a web app for product-supply business. Figure 2 depicts the classes that you have so far. In particular, you have designed a ProductsController class that records product information. As part of this controller's responsibilities, it must create new product entries when new products are added. You have also designed a ManagerNotifier that is capable of sending notification messages to managers at the company. The design problem you need to solve is how to make a ManagerNotifier "listen" for when a ProductsController creates a new product, and to send a notification to a manager whenever that happens. Draw a class diagram that applies the Observer Design Pattern to solve this problem. Use the same names used in the design pattern as much as possible (except make Ruby style). You must include all the classes from Figure 2 in your diagram (i.e., your changes should be additive). In particular, I expect that you will be adding classes, operations, inheritance relationships, and associations.

Solution:


Subject
attach(observer)
detach(observer)
notify()
1
*
observer
Observer
update()
ProductsController
...
create()
...
1
subject
1
ManagerNotifier
...
send_notification()
...
update()

Consider these figures when answering the following question.

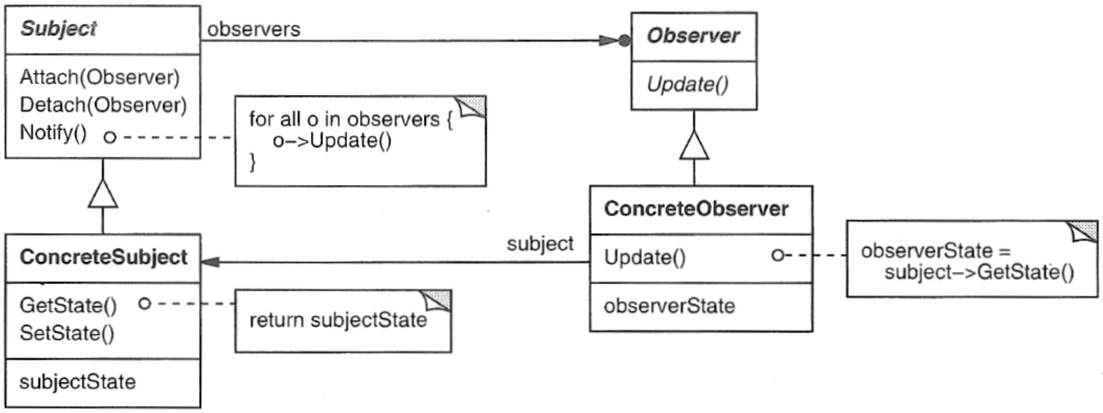


Figure 3. Observer Pattern from the "Gang of Four" book. (Note that the book uses an outdated class diagram notation.)

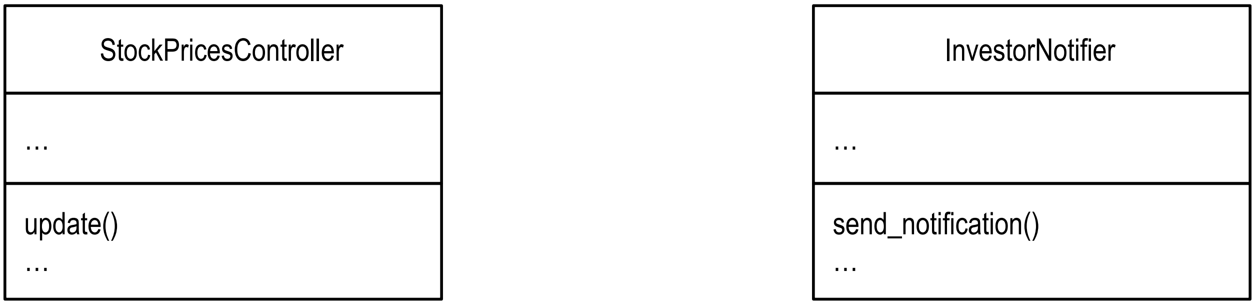
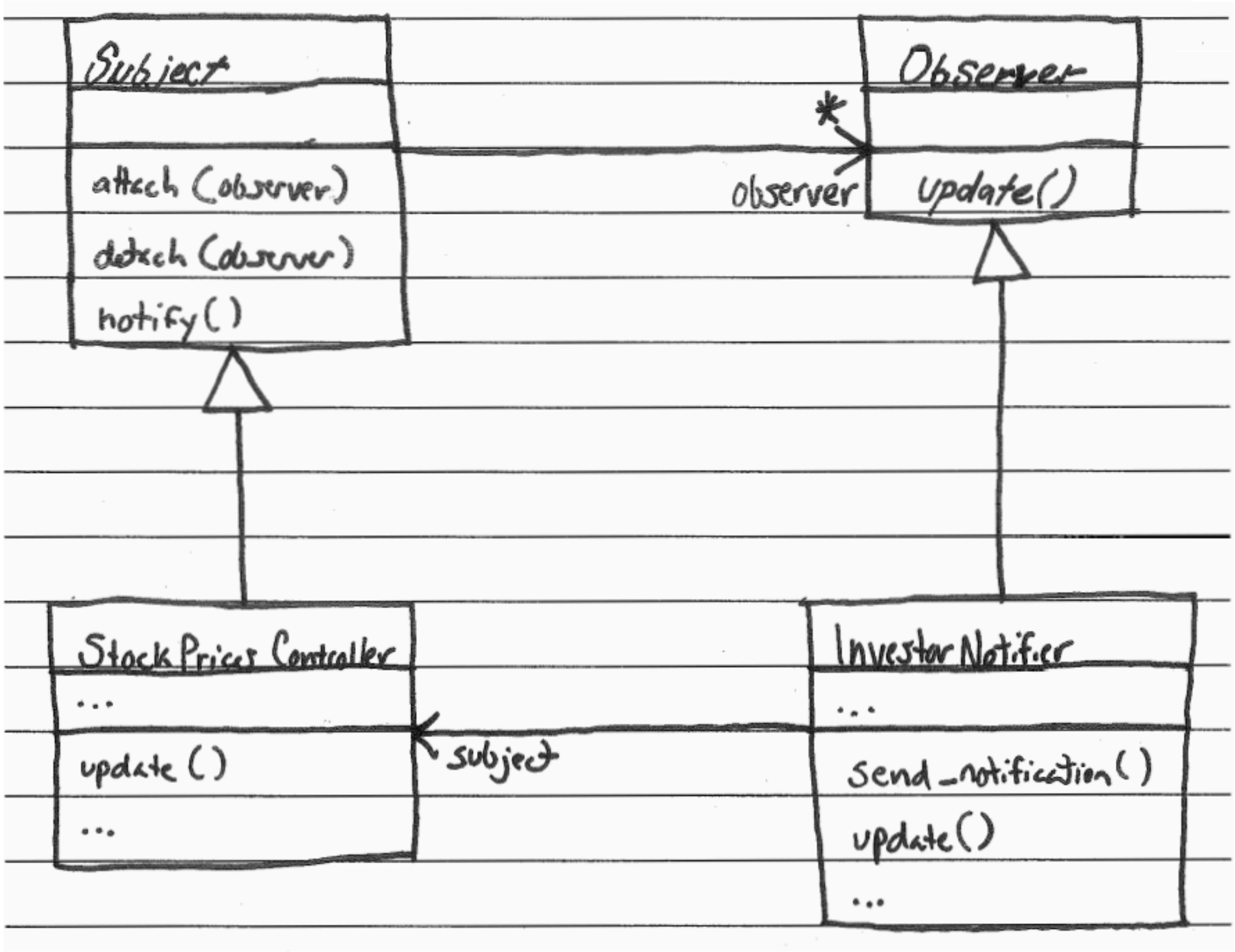


Figure 4. Classes for investment company web app.

Problem:

Recall the Observer Design Pattern depicted in Figure 3. Imagine that you are designing a web app for an investment company. Figure 4 depicts the classes that you have so far. In particular, you have designed a StockPricesController class that records price changes to stocks. As part of this controller's responsibilities, it must update stock price entries as they change. You have also designed a InvestorNotifier that is capable of sending notification messages to investors. The design problem you need to solve is how to make a InvestorNotifier "listen" for when a StockPricesController updates a stock price, and to send a notification to affected investors whenever that happens. Draw a class diagram that applies the Observer Design Pattern to solve this problem. Use the same names used in the design pattern as much as possible (except make Ruby style). You must include all the classes from Figure 4 in your diagram (i.e., your changes should be additive). In particular, I expect that you will be adding classes, operations, inheritance relationships, and associations.

Solution:


Subject
attach (observer)
detach (observer)
notify()
Observer
*
observer
Update()
StockPrices Controller
...
update ()
...
subject
InvestorNotifier
...
send_notification()
update()
...

In answering the next question, consider this application of the Observer Pattern. In the application, there is a products controller that can create new products in the system. Manager notifiers observe the products controller, and send notifications to managers when new products are created.

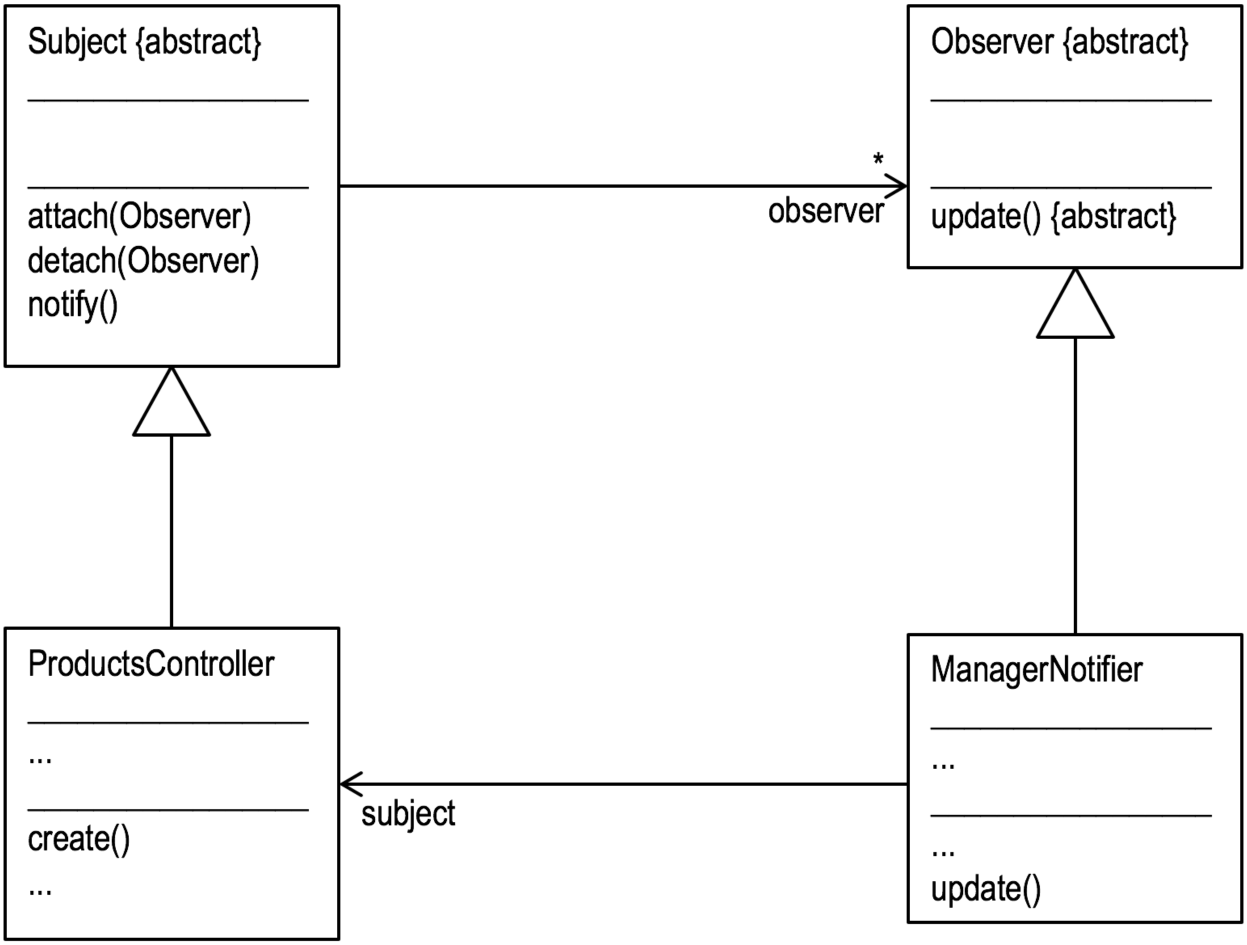
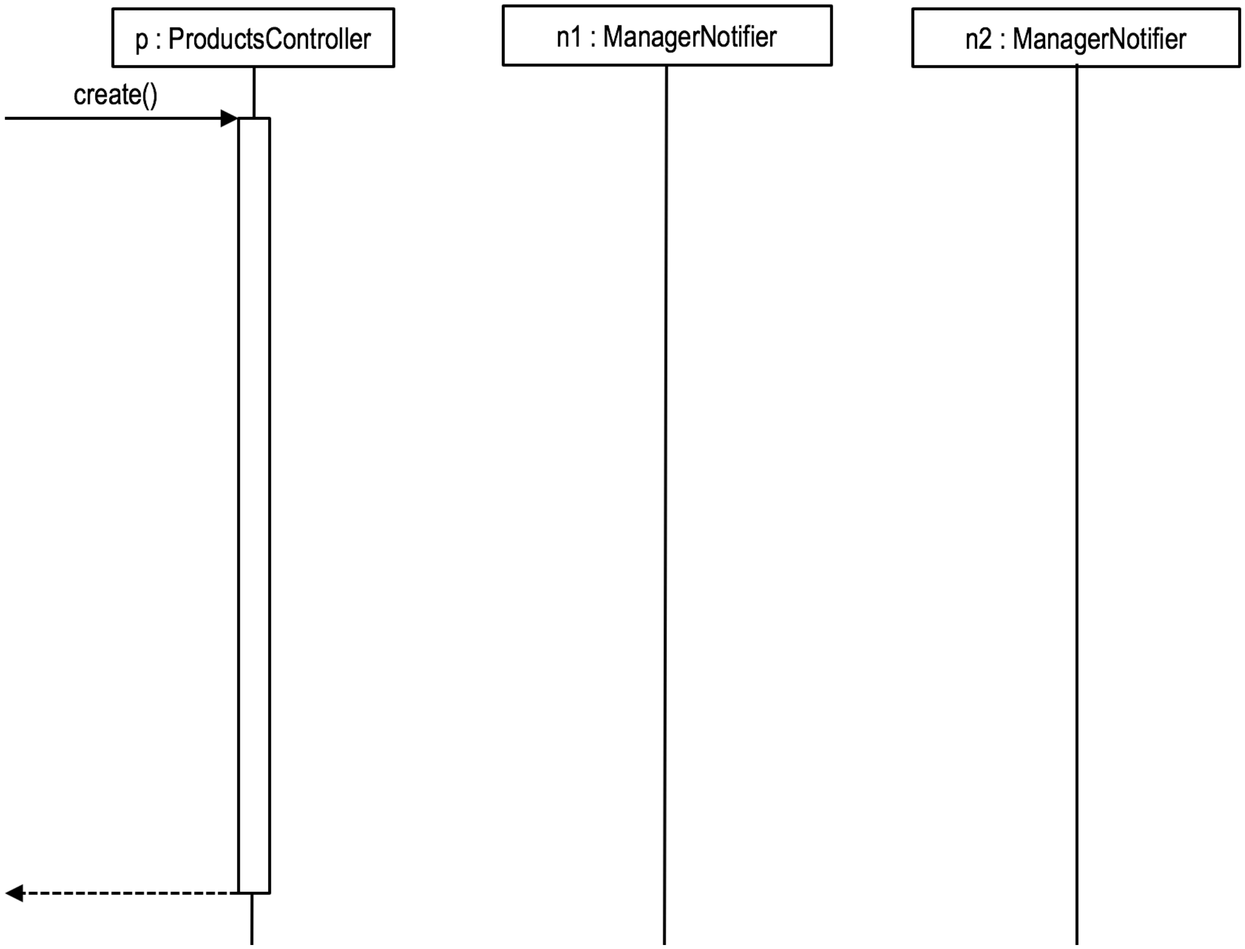


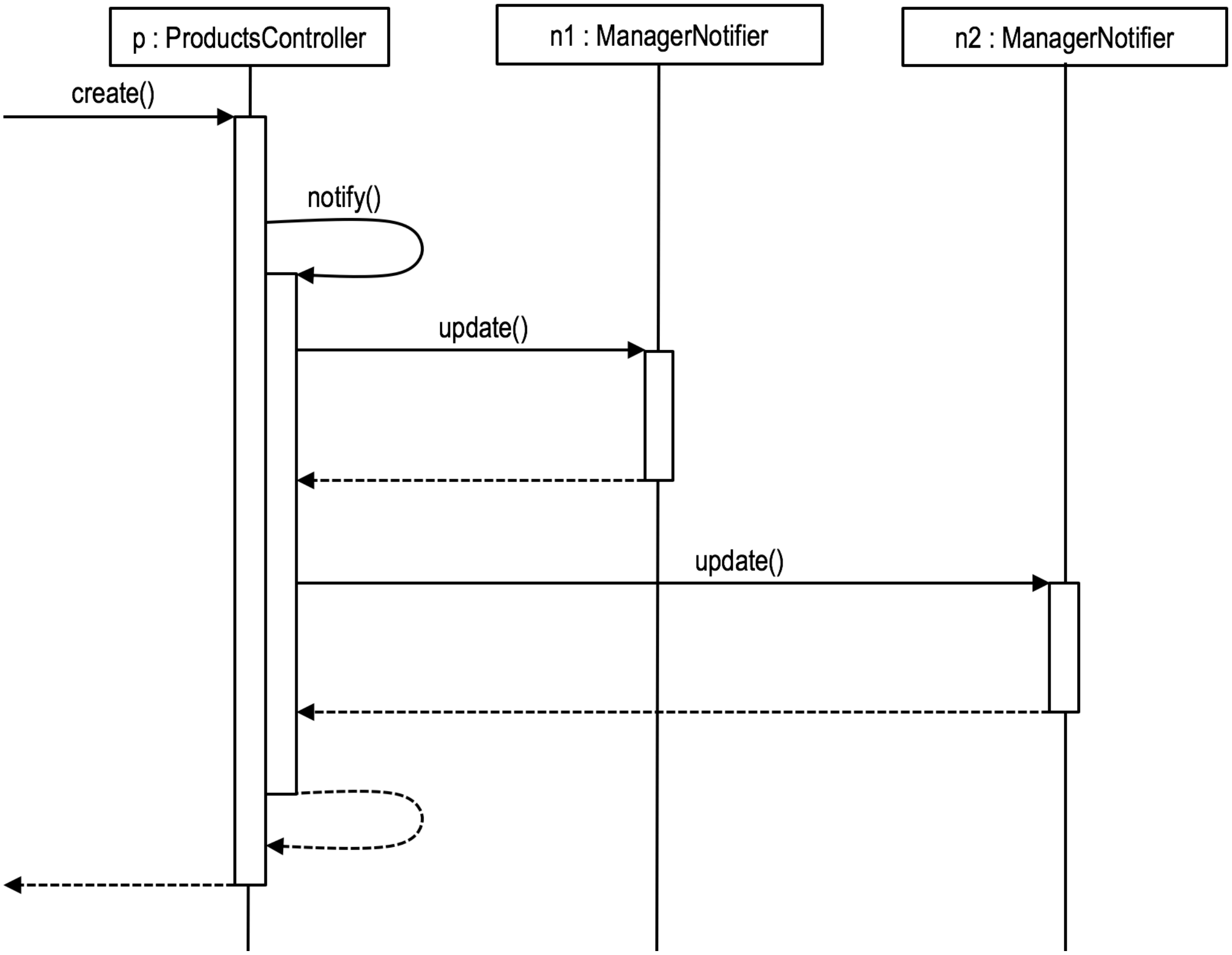
Figure 5. Application of Observer Pattern for a product management application that notifies managers whenever a new product is created in the system.

Problem:

The partially completed sequence diagram below depicts a ProductsController object (*p*) and two ManagerNotifier objects (*n1* and *n2*). The ManagerNotifier objects are already attached to the ProductsController object (although it is not depicted explicitly in the sequence diagram). Complete the sequence diagram such that, as per the Observer Pattern, it shows the method calls and returns triggered by the products controller creating a new product. Show only calls to methods that are depicted in the class diagram.



Solution:


p : ProductsController
n1 : ManagerNotifier
n2 : ManagerNotifier
create()
notify()
update()
update()