1. The class diagram below shows an outline of what the software might look like, however, it is likely that many acceptable answers may differ in many important respects, for example the relationship between Program List, its subclasses and Program might vary. It is also possible to move methods between classes.
2. There are six classes shown in the diagram. They are:

(a) WebPurchasingApplication

(b) Order - which provides four operations:
   i. add_item which takes an argument of type Item
   ii. remove_item which also takes an argument of type Item
   iii. print_delivery_order
   iv. charge_credit_card

(c) Customer - which has three attributes (name, address and credit_card_no) and provides the operation web_form_parser.

(d) Shopping basket which provides four operations:
   i. add_item which takes an argument of type Item
   ii. remove_item which also takes an argument of type Item
   iii. checkout
   iv. confirm_purchase

(e) Item which has the attribute num_ordered and provides two operations, description and price.

(f) Book which provides three operations, num_available, title and price.

There are seven relationships shown in the UML diagram. These show that

(a) The Web Purchasing Application contains 0 or more Customers. The filled diamond indicates ownership of the Customers by the Application.

(b) The Web Purchasing Application also contains 0 or more Books. Again the solid diamond indicates ownership and shows that Books exist for the lifetime of the application (ie the books form a static catalogue).

(c) The Web Purchasing Application also contains Orders, but the open diamond shows that these have lifetimes that are independent of that of the application.

(d) the arrow between Order and Customer shows navigability between objects of these classes (i.e. each Order has a reference (or pointer) to exactly one Customer and each Customer has a container of references (or pointers) to 0 or more Orders).

(e) Each Customer has exactly one Shopping basket.

(f) Each Shopping basket has 0 or more items, furthermore, these items are transient (items may be added or removed from the shopping basket).

(g) Each Item refers to exactly one Book.
4. This requires creating a class hierarchy to replace Book which contains all the various types of product. In the diagram shown below, the title() function has been replaced by the more generic description().
5. The decorator pattern should be applied to Item rather than Book because books are permanent items in the database of the application whereas Items are created on demand. Applying the decorator to Book would mean doubling the number of items in the catalog which would have to contain both “War and Peace” and “War and Peace (wrapped)”. By contrast, applying the pattern to Item means that wrapped items can be created on the fly as needed. However, one instance where decorating book might be better is if we only wished to permit wrapping of particular books - in which case, only wrapped versions of these items would appear in the catalogue.

The pattern could also be implemented using the more traditional form of decorator as shown below - but this would permit doubly wrapping a book which would only really be useful for games of pass the parcel...
6. In this case the composite pattern should be applied to Product Line (which is the abstract class that replaced Book when we added CDs etc) because collections of books are long lived entities in the system. The fact that the company offers a discount on the complete set of “The Lord of the Rings” has a longer life than a customer’s decision to purchase the collection. Furthermore, the company only wishes to offer the discount on particular collections and the discount rate may vary between collections.

7. Two realisations are needed to solve this problem. The first is that Customer has to become a class hierarchy to support different behaviours for Personal Customers and Corporate Customers. Secondly, the charge credit card function is no longer appropriate within Order and has to be moved to Personal Customer.

The function within Order can be changed to charge customer - but this must operate by calling a virtual (or abstract) function within Customer (which is now an abstract base class). This function is then implemented differently within Corporate Customer and Personal Customer either by charging the credit card as before in the case of Personal Customer or by adding the cost to the monthly invoice for the corporate customer.