of books borrowed by a member, or an anxious user asking the librarian when he can expect the book on which he placed a hold. In all these situations we have operations related to some `Member` and `Book` objects.

4. **Holds** Unlike `Borrows`, this class denotes a many-to-many relationship between the `Member` and `Book` classes. *In typical many-to-many relationships, implementation of the association without using an additional class is unlikely to be clean and efficient.* To attempt to do this without an additional class in the case of holds, we would need to maintain within each `Member` object references to all `Book` instances for which there is a hold, and keep 'reverse' references from the `Book` objects to the `Member` objects. This is, however, incomplete because we also need to maintain for each hold the number of days for which it is valid.But there is no satisfactory way of associating this attribute with the references. We could have queries like a user wanting a list of all of his holds that expire within 30 days. The reader can verify that implementations without involving an additional class will be messy and inefficient.

   It is, therefore, appropriate that we have a class for this relationship and make the `Hold` object accessible to the instances of `Member` and `Book`.

As we look at ways to implement the use cases, it often happens that we eliminate some of these classes, discover more, and determine the attributes and methods for all of the concrete classes.

7.1.3 Assigning responsibilities to the classes

Having decided on an adequate set of software classes, our next task is to assign responsibilities to these. Since the ultimate purpose of these classes is to enable the system to meet the responsibilities specified in the use case, we shall work with these system responsibilities to find the class responsibilities. The next step is, therefore, to spell out the details of how the system meets its responsibilities by devolving these down to the software classes, and the UML tool that we employ to describe this devolution is the sequence diagram.

It should be noted that the sequence diagram is only a concise, visual way of *representing* the devolution, and we need to make our design choices *before* we start drawing our arrows. For each system response listed in the right-hand column of the use case tables, we need to specify the following:

- The sequence in which the operations will occur.
- How each operation will be carried out.

For the first item above, we need a complete algorithm; the second item describes which classes will be involved in each step of the algorithm and how the classes will be engaged.